

# Chapitre 4 : Le langage C

## Construction et maintenance de logiciels

Guy Francoeur

basé sur les travaux d'Alexandre Blondin Massé, professeur

5 septembre 2019

**UQÀM** | **Département d'informatique**

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

- ▶ Ce matériel n'est pas officiel il est en bêta pour 2020!

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

- ▶ Collection de données de **même type**;

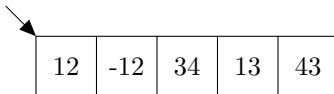
- ▶ **Déclaration** :

```
1 int donnees[10];  
2 // Réserve 10 "cases" de type "int" en mémoire  
3 int donnees[taille];  
4 // Seulement avec C99 et allocation sur la pile
```

- ▶ **Définition** et **initialisation** :

```
1 int toto[] = {12,-12,34,13,43};
```

- ▶ Stockées de façon **contiguë** en mémoire;



- ▶ À l'aide de l'opérateur `[]` :

```
1 // exo8.c
2 #include <stdio.h>
3 int main() {
4     int donnees[] = {12,-12,34,13,43};
5     int a, b;
6     a = donnees[2];
7     b = donnees[5];
8     printf("%d %d\n", a, b); /* que vaut a et b ? */
9     return 0;
10 }
```

- ▶ Le **premier** élément est à l'indice **0**;
- ▶ S'il y a **dépassement** de borne, **aucune erreur** ou un **avertissement** (**warning**).
- ▶ Source fréquente de **segfault**.

# Chaînes de caractères

- ▶ Les **chaînes de caractères** sont représentées par des **tableaux de caractères**;
- ▶ Les chaînes **constantes** sont délimitées par les symboles de guillemets " ".
- ▶ Les deux déclarations suivantes sont **équivalentes** :

```
1 char chaîne[] = "tomate";  
2 char chaîne[] = { 't', 'o', 'm', 'a', 't', 'e', '\0' };
```

- ▶ Termine par le caractère `\0`;
  - ▶ Longueur de la chaîne "tomate" : **6**;
  - ▶ Taille du tableau de la chaîne "tomate" : **7**.

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données



- ▶ Une adresse est un emplacement **précis** en mémoire.
- ▶ Un pointeur est une variable qui contient l'adresse d'une autre variable en mémoire;
- ▶ On déclare un pointeur en utilisant le symbole **\***;
- ▶ L'opérateur **&** retourne l'adresse d'une variable en mémoire.

# Exemple

```
1 //pointeur1.c
2 #include <stdio.h>
3
4 int main() {
5     int *p; //un pointeur vers un entier
6
7     printf("La variable p pointe vers l'adresse %p.\n", p);
8
9     return 0;
10 }
```

# Exemple

```
1 // pointeur2.c
2 #include <stdio.h>
3
4 int main() {
5     int *pi, x = 104;
6     pi = &x;
7     printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
8     printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);
9
10    *pi = 350;
11    printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
12    printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);
13    return 0;
14 }
```

## Affiche :

x vaut 104 et se trouve à l'adresse 0x7fff5fbff73c

pi vaut 0x7fff5fbff73c et pointe sur la valeur 104

x vaut 350 et se trouve à l'adresse 0x7fff5fbff73c

pi vaut 0x7fff5fbff73c et pointe sur la valeur 350

# Affectation

- ▶ Impossible d'affecter directement une **adresse** à un pointeur :

```
1 int *pi;  
2 pi = 0xdff1;          /* interdit */
```

- ▶ Par contre, avec une conversion **explicite**, c'est possible :

```
1 int *pi;  
2 pi = (int*)0xdff1; /* permis, mais à éviter */
```

- ▶ On peut aussi utiliser une conversion pour associer une **même adresse** à des pointeurs de **types différents** :

```
1 int *pi;  
2 char *pc;  
3 pi = (int*)0xdff1;  
4 pc = (char*)pi;
```

# Lien entre tableaux et pointeurs

- ▶ Un tableau d'éléments de type  $t$  peut être vu comme un **pointeur constant** vers des valeurs de type  $t$ ;
- ▶ **Exemple** : `int a[3]` définit un pointeur `a` vers des entiers;
- ▶ De plus, `a` pointe vers le **premier** élément du tableau :

```
1 //pointeur3.c
2 #include <stdio.h>
3 int main() {
4     int a[3] = {1,2,3}, *pi;
5     pi = a;           /* initialisation de pi */
6     printf("%p, %p, %d, %d, %d, %d\n",
7           a, pi, a[0], a[1], a[2], *pi);
8     return 0;
9 }
```

- ▶ **Affiche** : `0x7fff5fbff720 0x7fff5fbff720 1 2 3 1`
- ▶ `pi = a` est valide, mais `a = pi` n'est pas **valide**.

# Un extra sur les pointeurs

```
1 //pointeur5.c
2 #include <stdio.h>
3
4 int main() {
5     int a;
6     int *b;
7     int *c=NULL;
8
9     printf("%p, %d\n", a, a);
10
11     *b = 100000;
12     printf("%p, %d\n", b, *b);
13
14     *c = 100000000;
15     printf("%p, %d\n", c, *c);
16
17     return 0;
18 }
```

► Que donne le programme?

# Un extra sur les pointeurs

```
1 //sizeof.c
2 #include <stdio.h>
3
4 struct une_s {
5     unsigned long a;
6     unsigned long b;
7 };
8
9 int main(void) {
10
11     int a[3]={0,1,2};
12     struct une_s b;
13     unsigned __int128 c;
14
15     int *a2=a;
16     struct une_s *b2=&b;
17     unsigned __int128 *c2;
18 //sizeof:
19     printf("var a %lu, pointeur a2 %lu\n", sizeof a, sizeof a2);
20     printf("var b %lu, pointeur b2 %lu\n", sizeof b, sizeof b2);
21     printf("var c %lu, pointeur c2 %lu\n", sizeof c, sizeof c2);
22
23     return 0;
24 }
```

# Opération sur les pointeurs

- ▶ Considérons un tableau **tab** de **n** éléments. Alors
  - ▶ **tab** correspond à l'**adresse** de **tab[0]**;
  - ▶ **tab + 1** correspond à l'**adresse** de **tab[1]**;
  - ▶ ...
  - ▶ **tab + n - 1** correspond à l'**adresse** de **tab[n - 1]**;
- ▶ On peut calculer la **différence** entre deux pointeurs de même type;
- ▶ De la même façon, l'**incrément** et la **décrément** de pointeurs sont possibles;
- ▶ Finalement, deux pointeurs peuvent être **comparés**.



## Exemple - Académique

```
1 // pointeur4.c
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     int a[3] = {1, -1, 2}, *pi, *pi2;
6     pi = a;
7     pi2 = &a[2];
8     printf("%ld ", pi2 - pi);
9     printf("%d ", *(--pi2));
10    printf("%d\n", *(pi + 1));
11    if (pi + 1 == pi2)
12        printf("pi et pi2 pointent vers la même case mé-
13                moire.\n");
14    return 0;
15 }
```

**Affiche :**

2 -1 -1

pi et pi2 pointent vers la même case mémoire.

```
1  int *pi, tab[10];
```

- ▶ La déclaration d'un tableau **réserve** l'espace mémoire nécessaire pour stocker le tableau;
- ▶ La déclaration de \*pi ne réserve **aucun espace** mémoire (sauf l'espace pour stocker une adresse);
- ▶ Les expressions

```
1  *pi = 6;  
2  *(pi + 1) = 5;
```

sont **valides**, mais ne réservent pas l'espace mémoire correspondant. Autrement dit, le compilateur pourrait éventuellement **utiliser** cet espace. Donc ?

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

# Chaînes de caractères

- ▶ Une **chaîne de caractères** est représentée par un **tableau de caractères** terminant par le caractère `\0`;

t	o	m	a	t	e	\0
---	---	---	---	---	---	----

- ▶ Des fonctions élémentaires sur les **caractères** se trouvent dans la bibliothèque `ctype.h`;
- ▶ D'autre part, la bibliothèque standard `string.h` fournit plusieurs fonctions permettant de **manipuler** les **chaînes de caractères**.

# Arguments de la fonction main

- ▶ `int main(int argc, char *argv[]);`
- ▶ Le paramètre `argv` est un tableau de **pointeur vers des caractères**;
- ▶ `argv[argc] == NULL` est vrai;
- ▶ Quelle est la sortie affichée par le programme suivant avec la commande `gcc ex8.c && ./a.out bonjour toi ?`

```
1 // ex8.c
2 #include <stdio.h>
3 int main(int argc, char **argv) {
4     printf("argc est : %d\n", argc);
5     for (int i = 0; i < argc; ++i) {
6         printf("%s\n", argv[i]);
7     }
8     return 0;
9 }
```

Fonction	Description
int isalpha(c)	Retourne une valeur non nulle si c est alphabétique, 0 sinon
int isupper(c)	Retourne une valeur non nulle si c est majuscule, 0 sinon
int islower(c)	Retourne une valeur non nulle si c est minuscule, 0 sinon
int isdigit(c)	Retourne une valeur non nulle si c est un chiffre, 0 sinon
int isalnum(c)	Retourne isalpha(c)    isdigit(c)
int isspace(c)	Retourne une valeur non nulle si c est un espace, un saut de ligne, un caractère de tabulation, etc.
char toupper(c)	Retourne la lettre majuscule correspondant à c
char tolower(c)	Retourne la lettre minuscule correspondant à c

**Attention!** Les fonctions `toupper`, `tolower`, etc. sont définies sur les **caractères** et non sur les **chaînes**.

- ▶ La fonction `unsigned int strlen(char *s)` retourne la **longueur** d'une chaîne de caractères;
- ▶ La fonction `int strcmp(char *s, char *t)` retourne
  - ▶ une valeur **négative** si  $s < t$  selon l'ordre lexicographique;
  - ▶ une valeur **positive** si  $s > t$ ;
  - ▶ la valeur **0** si  $s == t$ .
- ▶ Quelle est la différence entre  $s == t$  et `strcmp(s, t)` ?

# Exemple

```
1 //ex1.c
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     char s[] = "bonjour";
7     char t[] = "patate";
8
9     printf("Longueur de \"%s\" et \"%s\" : %lu, %lu\n",
10           s, t, strlen(s), strlen(t));
11     printf("strcmp(\"%s\", \"%s\") : %d\n", s, t,
12           strcmp(s, t));
13     return 0;
14 }
```

## Sortie :

Longueur de "bonjour" et "patate" : 7, 6  
strcmp("bonjour", "patate") : -14



## ► Les fonctions

```
1      char *strcat(char *s, const char *t);  
2      char *strncat(char *s, const char *t, int n);
```

permettent de **concaténer** deux chaînes de caractères;

- Plus précisément, la chaîne **t** est ajoutée à la fin de la chaîne **s** ainsi qu'un caractère **\0**;
- La chaîne **s** doit avoir une **capacité suffisante** pour contenir le résultat de la **concaténation**;
- Le paramètre **n** donne une limite **maximale** du nombre de caractères à concaténer.

Quel résultat donne le code suivant ?

```
1 // ex2.c
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     char s[10] = "Salut ";
7     char t[] = "toi!";
8     strcat(s, t);
9     printf("%s\n", s);
10    return 0;
11 }
```

## ► Les fonctions

```
1 char *strcpy(char *s, const char *t);  
2 char *strncpy(char *s, const char *t, int n);
```

permettent de **copier** une chaîne de caractère dans une autre;

- Dans ce cas, la chaîne **t** est copiée dans la chaîne **s** et un caractère `\0` est ajouté à la fin;
- Comme pour `strcat`, la chaîne **s** doit avoir une **capacité suffisante** pour contenir la copie;
- Le paramètre **n** donne une limite **maximale** du nombre de caractères à copier;
- Quelle est la différence entre `s = t` et `strcpy(s, t)` ?

# Segmentation d'une chaîne

## ► La fonction

```
1 char *strchr(char *s, int c);
```

retourne un **pointeur** vers la première occurrence du **caractère** **c** dans **s**.

## ► La fonction

```
1 char *strtok(char *s, const char *delim);
```

permet de **décomposer** une chaîne de caractères en **plus petites chaînes** délimitées par des caractères donnés;

- Le paramètre **s** correspond à la chaîne qu'on souhaite **segmenter**, alors que le paramètre **delim** donne la liste des caractères considérés comme **délimiteurs**;
- Très **utile** lorsqu'on souhaite extraire des données d'un **fichier texte**.

# Décomposition avec champs vides

- ▶ La fonction **strtok** ne gère pas les cas où certains champs sont **vides**;

- ▶ Par exemple, si les données sont

```
1 "124:41:3::23:10"
```

il ne sera pas détecté qu'il y a une donnée **manquante** entre 3 et 23;

- ▶ La fonction

```
1 char *strsep(char **s, const char *delims);
```

résoud ce problème.

- ▶ **Attention !** Les fonctions **strtok** et **strsep** modifient la chaîne **s**.

# Exemple (1/2)

```
1 #include <stdio.h>
2 #include <string.h>
3 #define DELIMS ":"
4
5 int main() {
6     char s[80];
7     char *pc, *ps;
8
9     strcpy(s, "124:41:3::23:10");
10    printf("Avec strtok:\n");
11    pc = strtok(s, DELIMS);
12    while (pc != NULL) {
13        printf("/%s/\n", pc);
14        pc = strtok(NULL, DELIMS);
15    }
16
17    strcpy(s, "124:41:3::23:10");
18    printf("Avec strsep:\n");
19    ps = s;
20    while ((pc = strsep(&ps, DELIMS)) != NULL) {
21        printf("/%s/\n", pc);
22    }
23    return 0;
24 }
```

## Exemple (2/2)

### Résultat :

Avec strtok:

/124/

/41/

/3/

/23/

/10/

Avec strsep:

/124/

/41/

/3/

//

/23/

/10/

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données



# Tableaux multidimensionnels

## ► Déclaration :

```
1 // Matrice de 3 lignes et 2 colonnes
2 int matrice[3][2];
```

- Si la variable est **locale** (**automatique**), alors le tableau contient des valeurs **quelconques**;

- Le nombre de **dimensions** est **illimité**;

## ► Initialisation :

```
1 int matrice[3][2] = { {1,2}, {3,4}, {5,6} };
```

- **Accès** à un élément :

```
1 matrice[1][1] = 8;
```

# Affectations

- Les deux affectations suivantes sont **équivalentes** :

```
1 int a[3][2] = { {1,2}, {3,4}, {5,6} };  
2 int a[3][2] = { 1,2,3,4,5,6 };
```

- En revanche, les affectations suivantes ne sont pas **équivalentes** :

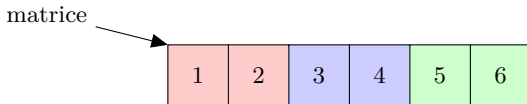
```
1 int a[3][2] = { {1}, {3,4}, {5} };  
2 int b[3][2] = { 1,3,4,5 };
```

- En effet, on a

$a[0][0] = 1, b[0][0] = 1$   
 $a[0][1] = 0, b[0][1] = 3$   
 $a[1][0] = 3, b[1][0] = 4$   
 $a[1][1] = 4, b[1][1] = 5$   
 $a[2][0] = 5, b[2][0] = 0$   
 $a[2][1] = 0, b[2][1] = 0$

# Mémoire réservée

- ▶ Les éléments sont d'abord rangés selon la **première dimension**, ensuite, selon la **deuxième**, etc.

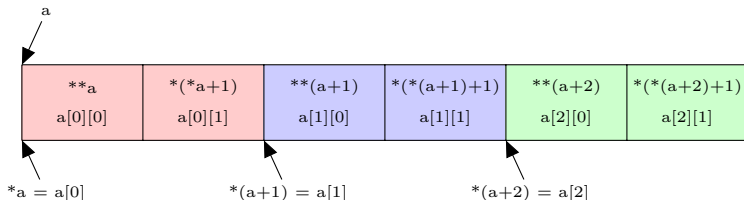


```
1 //ex5.c
2 #include <stdio.h>
3
4 int main() {
5     int matrice[3][2] = { {1,2}, {3,4}, {5,6} };
6     int i, j;
7
8     for (i = 0; i < 3; ++i)
9         for (j = 0; j < 2; ++j)
10             printf("%p -> %d ", &matrice[i][j], matrice[i][j]);
11     return 0;
12 }
```

## Sortie :

```
0x7fff5fbff720 -> 1 0x7fff5fbff724 -> 2 0x7fff5fbff728 -> 3 0x7fff5fbff72c -> 4
0x7fff5fbff730 -> 5 0x7fff5fbff734 -> 6
```

# Tableaux et pointeurs



- ▶ Remarquez que `a`, `*a` et `a[0]` ont la même **valeur**;
- ▶ En revanche, `a` est de type `int **` alors que `*a` et `a[0]` sont de type `int *`.

# Trois types de déclarations

- ▶ `int a[3][2];`
  - ▶ Réserve **six** emplacements contigus de taille `int`;
  - ▶ L'expression `(int *)a == a[0]` est **vraie**.
- ▶ `int *a[3];`
  - ▶ Réserve **trois** emplacements contigus de taille `int*`;
  - ▶ Permet d'avoir des lignes de **taille variable**;
  - ▶ L'expression `(int *)a == a[0]` est **fausse**.
- ▶ `int **a;`
  - ▶ Réserve **un** emplacement de taille `int**`;
- ▶ Dans les trois cas, on peut utiliser l'adressage `a[i][j]`.

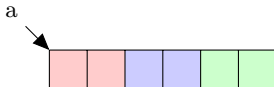
# Exemple

```
1 //ex6.c
2 #include <stdio.h>
3
4 int main() {
5     int m[2][3] = { {1,2,3}, {4,5,6} };
6     int *p[2] = {m[0], m[1]};
7     int **q;
8     q = (int**)m;
9     int i, j;
10
11     printf("%p %p %p\n", m, p, q);
12     for (i = 0; i < 2; ++i)
13         for (j = 0; j < 3; ++j)
14             printf("%p %p %p\n", &m[i][j], &p[i][j], &q[i][j]);
15     return 0;
16 }
```

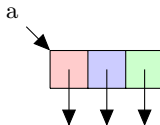
```
0x7ff5fbff700 0x7ff5fbff720 0x7ff5fbff700
0x7ff5fbff700 0x7ff5fbff700 0x200000001
0x7ff5fbff704 0x7ff5fbff704 0x200000005
0x7ff5fbff708 0x7ff5fbff708 0x200000009
0x7ff5fbff70c 0x7ff5fbff70c 0x400000003
0x7ff5fbff710 0x7ff5fbff710 0x400000007
0x7ff5fbff714 0x7ff5fbff714 0x40000000b
```

# Représentation abstraite

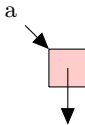
► `int a[3][2];`



► `int *a[3];`



► `int **a;`



# Tableaux de chaînes de caractères

Lorsqu'on souhaite définir un tableau dont les éléments sont des **chaînes de caractères**, on utilise plutôt le type `char *a[]`

```
1 //ex7.c
2 #include <stdio.h>
3
4 int main() {
5     char *mois [] = {"lundi", "mardi", "mercredi", "jeudi",
6                     "vendredi", "samedi", "dimanche"};
7     char **p;
8
9     p = mois;
10    printf("%c %c %s %s\n", **p, *mois[0], *(p+1), mois
11           [1]);
11    return 0;
12 }
```

**Sortie** : l l mardi mardi



# Tableaux multidimensionnels en arguments

- ▶ Il est alors nécessaire de spécifier la taille de **chaque dimension**, sauf la **première**;
- ▶ **Raison** : autrement, le compilateur ne sait pas comment gérer l'**indexation** s'il ne connaît pas la taille de chaque ligne;
- ▶ Il est possible de déclarer l'en-tête de la fonction avec **des pointeurs**, mais à ce moment-là, il faut utiliser différentes **astuces d'indexation**.

```
1 //ex9.c
2 #include <stdio.h>
3 int retourneEntree(int *m, int i, int j, int tailleLigne) {
4     return *(m + tailleLigne * i + j);
5 }
6 int main() {
7     int m[2][3] = { {1,2,3}, {4,5,6} };
8     printf("%d", retourneEntree((int*)m, 1, 1, 3)); //
9         Affiche 5
10    return 0
11 }
```

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres  
la fonction `main()`  
les fonctions
6. Structures et unions
7. Types énumératifs

- ▶ Elles sont l'unité de **base** de programmation;
- ▶ Chaque fonction doit effectuer **une** tâche bien précise;
- ▶ Elles permettent d'appliquer la stratégie **diviser-pour-régner**;
- ▶ Elles sont à la base de la **réutilisation**;
- ▶ Elles favorisent la **maintenance** du code;
- ▶ Lorsqu'elles sont **appelées**, l'exécution du bloc appelant est suspendue jusqu'à ce que l'instruction **return** ou la **fin** de la fonction soit atteinte.

# La fonction main

- ▶ La fonction **principale** de tout programme C. C'est cette fonction que le **compilateur** recherche pour exécuter le programme;
- ▶ La fonction main d'un programme n'acceptant aucun **argument** est

```
1 int main();
```

- ▶ Par convention, la valeur de **retour** de la fonction main est 0 si tout s'est bien déroulé et un **entier** correspondant à un **code d'erreur** différent de 0 autrement.

# Les arguments de la fonction main

- ▶ Lorsque la fonction main accepte des paramètres, elle est de la forme :

```
1      int main(int argc, char *argv[]);
```

- ▶ `argc` correspond au **nombre d'arguments** (incluant le nom du programme);
- ▶ `argv` est un tableau de **chaînes de caractères**, vues comme des **pointeurs**.
- ▶ `argv[0]` est une chaîne de caractères représentant le **nom du programme**;
- ▶ `argv[1]` est le **premier argument**, etc.

# Récupération des arguments de la fonction main

- ▶ Il faut se souvenir que **argv** est de type char;
- ▶ Donc, un argument (nombre) est reçu dans le main() comme une chaîne de caractère;

```
1 // stdlib.h
2 double strtod(const char *chaîne, char **fin);
3 unsigned long strtoul(const char *chaîne, char **fin, int
    base);
4 long strtol(const char *chaîne, char **fin, int base);
```

- ▶ **chaîne** : chaîne qu'on veut **traiter**;
- ▶ **fin** : ce qui **reste de la chaîne** après traitement;
- ▶ **base** : quelle est la base du nombre dans la chaîne;
- ▶ Les fonctions **atof**, **atoi**, **atol**, etc. sont **déconseillées**, car elles ne permettent pas de **valider** si la conversion s'est bien déroulée.

# Arguments et paramètres

```
1 int max(int x, int y) {  
2     if (x >= y) return x;  
3     else return y;  
4 }  
5  
6 printf(max(3, 4));
```

- ▶ Un **paramètre** d'une fonction est une **variable formelle** utilisée dans cette fonction (ex : x et y);
- ▶ Les fonctions ont **aucun**, **un** ou **plusieurs** paramètres d'**entrée**;
- ▶ Elles renvoient **au plus** un résultat en **sortie**.

# Cas 1

Quelles sont les valeurs affichées par ce programme ?

```
1 //error_swap.c
2 #include <stdio.h>
3 void echanger(int a, int b) {
4     int z = a;
5     a = b;
6     b = z;
7 }
8
9 int main() {
10     int a = 5, b = 6;
11     echanger(a, b);
12     printf("%d %d\n", a, b);
13     return 0;
14 }
```



## Cas 2

```
1 // swap.c
2 #include <stdio.h>
3 void echanger(int *a, int *b) {
4     int z = *a;
5     *a = *b;
6     *b = z;
7 }
8
9 int main() {
10     int a = 5, b = 6;
11     echanger(&a, &b);
12     printf("%d %d\n", a, b);
13     return 0;
14 }
```

# Passage par valeur ou adresse

- ▶ Les types de bases sont passés par **valeur**;
- ▶ Une **copie** de la valeur est transmise à la fonction;
- ▶ La **modification** de cette valeur à l'intérieur de la fonction **n'affecte pas** celle du bloc appelant.
- ▶ —
- ▶ La valeur n'est pas copiée;
- ▶ La variable d'origine reçoit les changements locaux.

# Passage d'un tableau

- ▶ Les tableaux comme **paramètres** d'une fonction :

```
1 float produit_scalaire(float a[], float b[], int d);
```

- ▶ Un tableau est représenté par un **pointeur constant**;
- ▶ Il est donc passé par **adresse** lors de l'appel d'une fonction;
- ▶ Si la fonction n'est pas censée **modifier** le tableau qu'elle reçoit en paramètre, il est convenable d'utiliser le mot réservé **const**.

```
1 float produit_scalaire(const float a[]  
2                          ,const float b[]  
3                          ,int d);
```

# Exemple

```
1 //passage_tableau.c
2 #include <stdio.h>
3
4 float produit_scalaire(const float a[], const float b[],
5                        unsigned taille) {
6
7     float p = 0.0;
8     for (int i = 0; i < taille; ++i) {
9         p += a[i] * b[i];
10    }
11    return p;
12 }
13
14 int main() {
15     float u[] = {1.0, -2.0, 0.0};
16     float v[] = {-1.0, 1.0, 3.0};
17     printf("%f\n", produit_scalaire(u, v, 3));
18     return 0;
19 }
```

Affiche : -3.000000

# Fonction retournant un tableau

- ▶ Une fonction ne peut pas **retourner** un **pointeur** créé dans la fonction, sauf s'il y a eu **allocation dynamique**;
- ▶ En particulier, on ne peut pas retourner un **tableau** comme résultat. Il faut plutôt que le tableau soit un des **arguments** de la fonction.

# Exemple

```
1 #include <stdio.h>
2
3 int* initialise_tableau(unsigned taille) {
4     int tableau[taille];
5     int i;
6     for (i = 0; i < taille; ++i)
7         tableau[i] = 0;
8     return tableau;
9 }
10
11 int main() {
12     int *tableau;
13     tableau = initialise_tableau(4);
14     printf("%d\n", tableau[0]);
15     return 0;
16 }
```

## Affiche :

exo18.c: In function 'initialise\_tableau':

exo18.c:8: warning: function returns address of local variable 0

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

# Les structures

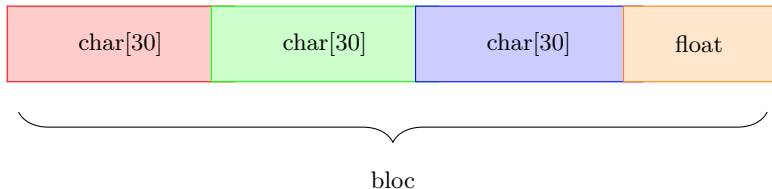
- ▶ Aussi appelées **enregistrements**;
- ▶ Permet de regrouper sous un même **bloc** des données de types **différents**;
- ▶ Définissent un **nouveau type** de données (données **composées**);
- ▶ Déclarées à l'aide du mot réservé **struct**;

```
1  struct Point2d {  
2      float x;  
3      float y;  
4  };
```



# Exemples

```
1 struct Livre {  
2     char  titre [30];  
3     char  auteur [30];  
4     char  editeur [30];  
5     float prix;  
6 };
```



- ▶ **Déclaration** d'une variable de type `struct Point2d` :

```
1 struct Point2d p;
```

- ▶ Attention de ne pas oublier le mot `struct` dans la déclaration.

- ▶ **Initialisation** :

```
1 struct Point2d p = {2.0, -1.2};
```

- ▶ On peut combiner déclaration, initialisation et définition.

# Affectation (*compound literal*)

- ▶ On peut **initialiser** une structure en spécifiant les **champs**;
- ▶ On peut aussi faire une **affectation** en bloc.

```
1 //compound.c
2 #include <stdio.h>
3
4 struct Rectangle {
5     float x;
6     float y;
7     float width;
8     float height;
9 };
10
11 int main() {
12     struct Rectangle r = {1.0, 2.0, 5.0, 6.0};
13     // r = {3.0, 8.0, 9.0, 7.0}; Syntaxe non valide
14     r = (struct Rectangle) {3.0, 8.0, 9.0, 7.0};
15     float a = 0.0, b = 0.0, c = 1.0, d = 2.0;
16     r = (struct Rectangle) {.x      = a,
17                             .y      = d,
18                             .width  = b,
19                             .height = c};
20     return 0;
21 }
```

# Manipulation des structures

```
1 struct Point2d p1 = {-1.2, 2.1};  
2 struct Point2d p2;
```

- ▶ L'affectation `p2 = p1` copie les champs des structures;
- ▶ Les structures sont passées par **valeurs** aux fonctions;
- ▶ Pour accéder aux différents **membres** d'une structure, il faut utiliser l'opérateur **point .** :

```
1 void affichePoint(struct Point2d p) {  
2     printf("(%f, %f)", p.x, p.y);  
3 }  
4  
5 int main() {  
6     struct Point2d p = {2.0, -1.2};  
7     affichePoint(p);  
8 }
```

Sortie : (2.000000, -1.200000)

# Pointeur sur une structure

- ▶ Lorsqu'on a un pointeur sur une structure, on doit utiliser l'opérateur `->`;
- ▶ La plupart du temps, il est préférable de passer les structures par **adresse** aux fonctions;
- ▶ C'est plus **efficace**, en particulier lorsque les structures sont de taille **importante**;
- ▶ Par exemple, **comparaison** de deux points :

```
1 int ptemp(const struct Point2d *p,  
2           const struct Point2d *q) {  
3     if (p->x != q->x) return p->x - q->x;  
4     else return p->y - q->y;  
5 }
```

- ▶ L'expression `p->x` est équivalente à `(*p).x`.

# Types composés

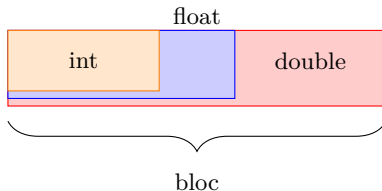
- ▶ Il est possible de créer des **structures** ayant des membres qui sont **eux-mêmes des structures**;
- ▶ On peut aussi composer des **structures** avec des **pointeurs** et des **tableaux**;

```
1 struct Segment {  
2     struct Point2d p;  
3     struct Point2d q;  
4 };  
5  
6 struct Carre {  
7     struct Point2d points[4];  
8 };
```

# Unions

- ▶ Permettent de créer des variables dont le contenu **diffère** selon le contexte;
- ▶ La variable sera créée avec une taille **suffisamment grande** pour contenir le type le **plus volumineux**;
- ▶ La **syntaxe** est la même que pour les **structures**.

```
1 union Nombre {  
2     int    i;  
3     float  f;  
4     double d;  
5 };
```



# Exemple

```
1 //union.c
2 #include <stdio.h>
3 int main() {
4     union Nombre {
5         int    i;
6         float  f;
7         double d;
8     };
9     union Nombre n;
10    n.i = 3;
11    printf( "%d %f %lf\n", n.i, n.f, n.d);
12    n.f = 2.0;
13    printf( "%d %f %lf\n", n.i, n.f, n.d);
14    n.d = 3.0;
15    printf( "%d %f %lf\n", n.i, n.f, n.d);
16 }
```

**Affiche :**

```
3 0.000000 0.000000
1073741824 2.000000 0.000000
0 0.000000 3.000000
```



# Initialisation des unions

- ▶ Comme les **structures**, les **unions** peuvent être initialisées en **bloc**;
- ▶ Par contre, seul le premier membre peut être initialisé.

```
1 //union_init.c
2 #include <stdio.h>
3
4 int main() {
5     union Nombre {
6         int i;
7         float f;
8         double d;
9     };
10    union Nombre n1 = {3};
11    printf("%d %f %lf\n", n1.i, n1.f, n1.d);
12    union Nombre n2 = {2.1};
13    printf("%d %f %lf\n", n2.i, n2.f, n2.d);
14 }
```

## Résultat :

```
3 0.000000 0.000000
2 0.000000 0.000000
```

# Structures et unions anonymes

- On peut déclarer des **structures** et des **unions** dans d'autres **structures** sans leur donner de nom :

```
1 //anonyme.c
2 #include <stdio.h>
3 #include <stdbool.h>
4
5 struct Choix {
6     bool estNombre;
7     union {
8         float nombre;
9         char *chaine;
10    };
11 };
12
13 void afficherChoix(struct Choix *choix) {
14     if (choix->estNombre) {
15         printf("%lf\n", choix->nombre);
16     } else {
17         printf("%s\n", choix->chaine);
18     }
19 };
20
21 int main() {
22     struct Choix choix = {false, .chaine = "oui"};
23     afficherChoix(&choix);
24     choix = (struct Choix){true, 3.14};
25     afficherChoix(&choix);
26     return 0;
27 }
```

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

# Types énumératifs

## ► Déclaration

```
1      enum Jour {LUN, MAR, MER, JEU,  
2              VEN, SAM, DIM};
```

- Une des façons de définir des **constantes**;
- La première valeur prend la valeur **0**, la seconde prend la valeur **1**, etc.
- Seules des valeurs **entières** sont permises :

```
1      // Ne fonctionne pas !!!  
2      enum ConstanteMath {PI = 3.141592654,  
3                          E = 2.7182818};
```

# Limite des types énumératifs

L'instruction `enum` ne permet pas de détecter les **incohérences**;

```
1 //enum1.c
2 #include <stdio.h>
3
4 int main() {
5     typedef enum sexe {M = 1, F = 2} Sexe;
6     Sexe s = 8;
7     int t = M;
8     printf("%d %d\n", s, t);
9     return 0;
10 }
```

**Affiche** : 8 1

# Table des matières

1. Tableaux
2. Pointeurs
3. Chaînes de caractères
4. Tableaux multidimensionnels
5. Fonctions, paramètres
6. Structures et unions
7. Types énumératifs
8. Types de données

# L'instruction typedef

- ▶ Permet de définir de **nouveaux types**;

```
1 typedef char NAS[9];  
2 typedef char *String;  
3 typedef struct {  
4     float x;  
5     float y;  
6 } Point2d;  
7  
8 NAS nas;  
9 String s;  
10 Point2d p;
```

- ▶ Améliore la **lisibilité** du code dans **certains cas**;
- ▶ Les types sont seulement des **synonymes** : par exemple, toute fonction ayant un paramètre de type `char *` acceptera en argument le type `String`.

# Portée de struct, union et typedef

- ▶ Mêmes propriétés que les **variables** et les **fonctions**;
- ▶ Si déclaré **localement**, alors limité au bloc dans lequel ils sont **déclarés**;
- ▶ Si déclaré **globalement**, alors accessible jusqu'à la fin du fichier;
- ▶ Par contre, impossible de les déclarer **externes**;
- ▶ Pour rendre des **structures**, des **unions** et des **types** disponibles dans n'importe quel fichier, il faut les déclarer dans un fichier .h. Vous devez ensuite inclure le .h à l'aide de l'instruction **#include**.



# L'opérateur sizeof

- ▶ Retourne le nombre d'**octets** utilisés par
  - ▶ **un type de données** : `sizeof(int);`
  - ▶ **une valeur constante** : `sizeof("bonjour");`
  - ▶ **le nom d'une variable** : `sizeof(matrice);`
- ▶ L'expression est évaluée à la **compilation**;
- ▶ Permet de produire du code **plus portable**;
- ▶ **Très utile** pour l'allocation dynamique.

# Exemple

```
1 //enum2.c
2 #include <stdio.h>
3
4 int main() {
5     typedef struct {
6         int quantite;
7         float poids;
8     } Fruit;
9     int a[5];
10
11     printf("%lu %lu %lu %lu %lu\n", sizeof(int),
12         sizeof(float), sizeof(Fruit), sizeof a,
13         sizeof "bonjour");
14     return 0;
15 }
```

Affiche : 4 4 8 20 8