

# Chapitre 4 : Les bases du C, partie 3

Guy Francoeur

basé sur les travaux d'  
Alexandre Blondin Massé, professeur

Département d'informatique  
Université du Québec à Montréal

6 janvier 2019  
Construction et maintenance de logiciels  
INF3135

# Table des matières

1. Tableaux multidimensionnels
2. Structures et unions
3. Types énumératifs
4. Types de données

# Table des matières

1. Tableaux multidimensionnels
2. Structures et unions
3. Types énumératifs
4. Types de données

# Tableaux multidimensionnels

- **Déclaration :**

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

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

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

- **Initialisation :**

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

- **Accès** à un élément :

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

# Affectations

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

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

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

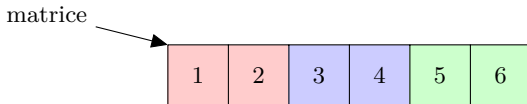
```
int a[3][2] = { {1}, {3,4}, {5} };  
int b[3][2] = { 1,3,4,5 };
```

- 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.



```
//ex5.c
#include <stdio.h>

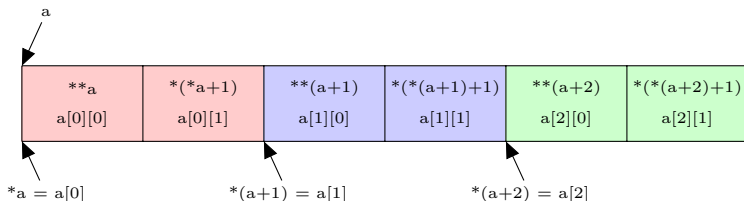
int main() {
    int matrice[3][2] = { {1,2}, {3,4}, {5,6} };
    int i, j;

    for (i = 0; i < 3; ++i)
        for (j = 0; j < 2; ++j)
            printf("%p -> %d ", &matrice[i][j], matrice[i][j]);
    return 0;
}
```

## 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

```
//ex6.c
#include <stdio.h>

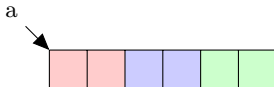
int main() {
    int m[2][3] = { {1,2,3}, {4,5,6} };
    int *p[2] = {m[0], m[1]};
    int **q;
    q = (int**)m;
    int i, j;

    printf("%p %p %p\n", m, p, q);
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 3; ++j)
            printf("%p %p %p\n", &m[i][j], &p[i][j], &q[i][j]);
    return 0;
}
```

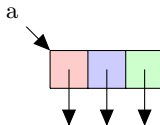
```
0x7ff5fbff700 0x7ff5fbff720 0x7ff5fbff700
0x7ff5fbff700 0x7ff5fbff700 0x200000001
0x7ff5fbff704 0x7ff5fbff704 0x200000005
0x7ff5fbff708 0x7ff5fbff708 0x200000009
0x7ff5fbff70c 0x7ff5fbff70c 0x400000003
0x7ff5fbff710 0x7ff5fbff710 0x400000007
0x7ff5fbff714 0x7ff5fbff714 0x40000000b
```

# Représentation abstraite

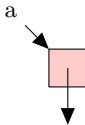
► `int a[3][2];`



► `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[]`

```
//ex7.c
#include <stdio.h>

int main() {
    char *mois[] = {"lundi", "mardi", "mercredi", "jeudi",
                   "vendredi", "samedi", "dimanche"};

    char **p;

    p = mois;
    printf("%c %c %s %s\n", **p, *mois[0], *(p+1), mois
           [1]);
    return 0;
}
```

**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**.

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

# Table des matières

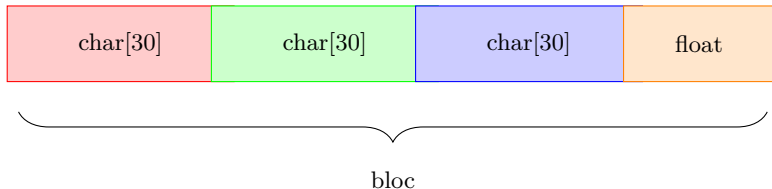
1. Tableaux multidimensionnels
2. Structures et unions
3. Types énumératifs
4. Types de données

- ▶ 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**;

```
struct Point2d {  
    float x;  
    float y;  
};
```

# Exemples

```
struct Livre {  
    char  titre [30];  
    char  auteur [30];  
    char  editeur [30];  
    float prix;  
};
```



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

```
struct Point2d p;
```

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

- ▶ **Initialisation** :

```
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.

```
//compound.c
#include <stdio.h>

struct Rectangle {
    float x;
    float y;
    float width;
    float height;
};

int main() {
    struct Rectangle r = {1.0, 2.0, 5.0, 6.0};
    // r = {3.0, 8.0, 9.0, 7.0}; Syntaxe non valide
    r = (struct Rectangle) {3.0, 8.0, 9.0, 7.0};
    float a = 0.0, b = 0.0, c = 1.0, d = 2.0;
    r = (struct Rectangle) {.x      = a,
                           .y      = d,
                           .width  = b,
                           .height = c};

    return 0;
}
```

# Manipulation des structures

```
struct Point2d p1 = {-1.2, 2.1};  
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 .** :

```
void affichePoint(struct Point2d p) {  
    printf("(%f, %f)", p.x, p.y);  
}
```

```
int main() {  
    struct Point2d p = {2.0, -1.2};  
    affichePoint(p);  
}
```

**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 :

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

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

# Types composés

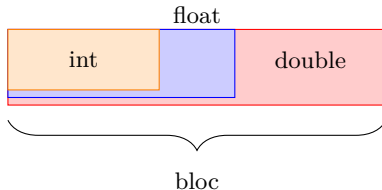
- ▶ Il est possible de créer des **structures** ayant des membres qui sont **eux-mêmes des structures**;
- ▶ On peut aussi composer des **structures** avec des **pointeurs** et des **tableaux**;

```
struct Segment {  
    struct Point2d p;  
    struct Point2d q;  
};  
  
struct Carre {  
    struct Point2d points[4];  
};
```

# 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**.

```
union Nombre {  
    int    i;  
    float  f;  
    double d;  
};
```



# Exemple

```
//union.c
#include <stdio.h>
int main() {
    union Nombre {
        int    i;
        float  f;
        double d;
    };
    union Nombre n;
    n.i = 3;
    printf("%d %f %lf\n", n.i, n.f, n.d);
    n.f = 2.0;
    printf("%d %f %lf\n", n.i, n.f, n.d);
    n.d = 3.0;
    printf("%d %f %lf\n", n.i, n.f, n.d);
}
```

**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.

```
//union_init.c
#include <stdio.h>

int main() {
    union Nombre {
        int    i;
        float  f;
        double d;
    };
    union Nombre n1 = {3};
    printf("%d %f %lf\n", n1.i, n1.f, n1.d);
    union Nombre n2 = {2.1};
    printf("%d %f %lf\n", n2.i, n2.f, n2.d);
}
```

## 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 :

```
//anonyme.c
#include <stdio.h>
#include <stdbool.h>

struct Choix {
    bool estNombre;
    union {
        float nombre;
        char *chaine;
    };
};

void afficherChoix(struct Choix *choix) {
    if (choix->estNombre) {
        printf("%lf\n", choix->nombre);
    } else {
        printf("%s\n", choix->chaine);
    }
};

int main() {
    struct Choix choix = {false, .chaine = "oui"};
    afficherChoix(&choix);
    choix = (struct Choix){true, 3.14};
    afficherChoix(&choix);
    return 0;
}
```



# Table des matières

1. Tableaux multidimensionnels
2. Structures et unions
3. Types énumératifs
4. Types de données

# Types énumératifs

- Déclaration

```
enum Jour {LUN, MAR, MER, JEU,  
           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 :

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

# Limite des types énumératifs

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

```
//enum1.c
#include <stdio.h>

int main() {
    typedef enum sexe {M = 1, F = 2} Sexe;
    Sexe s = 8;
    int t = M;
    printf("%d %d\n", s, t);
    return 0;
}
```

**Affiche** : 8 1

# Table des matières

1. Tableaux multidimensionnels
2. Structures et unions
3. Types énumératifs
4. Types de données

# L'instruction typedef

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

```
typedef char NAS[9];  
typedef char *String;  
typedef struct {  
    float x;  
    float y;  
} Point2d;
```

```
NAS nas;  
String s;  
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`.

# Utilisation de typedef

- ▶ De nombreux programmeurs **expérimentés** considèrent que l'instruction typedef est utilisée de façon **abusive**;
- ▶ Voir une **discussion intéressante sur Stack Overflow**, en particulier **cette réponse**.
- ▶ En tant que **programmeurs**, cependant, si vous avez à lire du code écrit en C, il est probable que vous rencontriez les **deux pratiques**;
- ▶ Il est donc important d'être familier avec les **typedefs**.

# 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** accessibles dans n'importe quel fichier, il faut alors les déclarer dans une **interface** (fichier .h) qu'on inclut à l'aide de l'instruction **#include** dans le préambule.

# 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

```
//enum2.c
#include <stdio.h>

int main() {
    typedef struct {
        int quantite;
        float poids;
    } Fruit;
    int a[5];

    printf("%lu %lu %lu %lu %lu\n", sizeof(int),
        sizeof(float), sizeof(Fruit), sizeof a,
        sizeof "bonjour");
    return 0;
}
```

**Affiche** : 4 4 8 20 8