ECE PARIS • LYON
ÉCOLE D'INGÉNIEURS

Algorithmique et programmation structurée en C

Cours n°5

Langage C
Structures et tableaux

Antoine Hintzy

Plan du cours

Les tableaux

Un tableau permet de **regrouper plusieurs éléments de même type** de façon ordonnée, dans ce que l'on appelle les **cases** de ce tableau. Ils permettent de faire des **collections** (de points par exemple).

⊙ **Déclaration d'un tableau :**

```
typeDesElements

nomDuTableau[NOMBRE_DE_CASES];
```

⊙ **Exemple :**

```
int tableauEntiers[10]; // tableau de 10 cases/entiers (10x4=40 octets)
```

Les tableaux – Statique

Le terme **tableau statique** est généralisé pour désigner ce genre de tableaux à taille (nombre de cases) fixe.

Le terme exact est **tableau automatique**.

Ces termes concernent l'**allocation** mémoire. Nous reviendrons dessus plus tard.

Nous utiliserons les deux termes (**statique**/**automatique**), qui ne sont pas à confondre avec le terme **dynamique** que nous verrons plus tard. Le terme **automatique** précise que le tableau sera automatiquement supprimé de la mémoire à la fin du bloc dans lequel il a été déclaré.

Les tableaux – Taille physique

On appelle **taille physique** le nombre de cases réservées en mémoire pour un tableau. Cette taille est choisie au moment de la déclaration du tableau, entre **[**crochets**]**.

```
#define TAILLE_PHYSIQUE 10
int main(void) {
  int entiers[TAILLE_PHYSIQUE]; // ce tableau permettra de stocker au maximum 10 entiers
...
```

Dans cet exemple, nous déclarons un tableau nommé "**`entiers`**" qui possède **`10`** cases pouvant contenir un entier (**`int`**).

Ce tableau occupe donc $10 \times \text{taille d'un entier} = 10 \times 4 \text{ octets} = 40 \text{ octets}$ en mémoire centrale.

NB : Une fois un tableau statique/automatique déclaré, il est impossible de lui rajouter ou lui supprimer des cases.

En revanche, rien ne nous oblige à toutes les utiliser.

😌 **Bonnes pratiques :** Déclarez la taille physique de chaque tableau dans une macro (**`#define`**).

Ajoutez un **`s`** à la fin du nom des tableaux afin d'indiquer qu'il s'agit bien d'une collection (nous devons le savoir pour agir en conséquence !).

Les tableaux – Indice d’une case

Chaque case de tableau a un indice dans ce tableau.

On accède aux éléments d'un tableau par leur **indice**, que l'on place entre **[**crochets**]**.

⚠️ Les indices commencent à **0**.

Ils vont de **0** *à la taille physique -* **1**.

⚠️ Les crochets ont une utilisation différente au moment de la déclaration et après.

```c
#include <stdio.h>
#define TAILLE_PHYSIQUE 10

int main(void) {
    int monTableau[TAILLE_PHYSIQUE]; // déclaration, donc ici, les crochets définissent la taille physique !
    // ...initialisation du tableau (voir slides suivantes)...
    printf("%d", monTableau[0]); // accès à la première case du tableau
    printf("%d", monTableau[1]); // accès à la deuxième case du tableau
    printf("%d", monTableau[TAILLE_PHYSIQUE-1]); // accès à la dernière case du tableau

    // Possibilité d'accéder à un indice contenu dans une variable :
    int indice = 3;
    printf("%d", monTableau[indice]); // 4ème case car "indice" vaut 3 (commence à 0)
```

Les tableaux – Exemple de parcours

```c
#include <stdio.h>
#define NB_ELEM_MAX 10

int main(void) {
    int i = 0;
    int monTableau[NB_ELEM_MAX]; // tableau d'au plus 10 entiers
    for (i = 0; i < NB_ELEM_MAX; i++) {
        // monTableau[i] correspond à la valeur de la case parcourue
        monTableau[i] = i * i; // on donne à chaque case la valeur de son indice au carré
        printf("%d ", monTableau[i]); // puis on affiche la valeur de la case parcourue
    }
    return 0;
}
```

Les tableaux – Exercice

Dans le `main`, déclarez un tableau permettant de stocker au plus 20 entiers, puis, affichez son contenu.

- Lancez l'exécutable plusieurs fois, sans modifier le programme.
- Que constatez-vous au niveau des valeurs du tableau ?
- À votre avis, pourquoi cela se produit-il ?
- Proposez une solution et testez-la.

Tableaux – Exemple

Lorsque nous déclarons une variable, nous récupérons un espace en mémoire centrale ayant potentiellement déjà servi, et comportant donc déjà des **0** et des **1**. C'est pourquoi **il faut toujours initialiser ses variables, tableaux compris** !

```c
#include <stdio.h>
#define NB_ELEM_MAX 10

int main(void) {
int i = 0;
  int monTableau[NB_ELEM_MAX];

  // On initialise toutes les cases du tableau à 0 afin de ne pas avoir de valeur indésirable :
  for (i = 0; i < NB_ELEM_MAX; i++) {
      monTableau[i] = 0;
  }

  return 0;
}
```

Tableaux – Exemple

```c
#include <stdio.h>
#define TAILLE 5

int main(void) {
int i = 0;
  int tab[TAILLE] = {5, 3, 12, 65, 35};
  // ou :
  int tab[] = {5, 3, 12, 65, 35}; // taille physique déduite
                                  // (déconseillé car elle n'est stockée nulle part)
                                  // Quand même spécifier la taille entre []

  for (i = 0; i < TAILLE; i++) {
    printf("%d ", tab[i]); // affiche successivement : 5 3 12 65 35
  }

  return 0;
}
```

Tableaux – Exemple

```c
#include <stdio.h>
#define TAILLE 5

int main(void) {
int i = 0;
  int tab[TAILLE] = {0}; // initialise toutes les cases à 0

  for (i = 0; i < TAILLE; i++) {
    printf("%d ", tab[i]); // affiche successivement : 0 0 0 0 0
  }

  return 0;
}
```

Tableaux – Taille logique

Ce n'est pas parce qu'un tableau possède x cases que nous devons toutes les utiliser.

Il faut prévoir en amont suffisamment de cases pour toutes les utilisations possibles de notre application. Mais toutes ne seront pas forcément utilisées. Ainsi, on peut par exemple ne stocker que 3 réels dans un tableau de taille physique prévue pour stocker jusqu'à 10 réels.

Le nombre de cases réellement utilisées est appelée **taille logique** du tableau. On la stocke dans une variable puisque sa taille est amenée à changer.

Remarques : On parcourt en général la taille logique du tableau. La taille logique doit forcément être inférieure ou égale à la taille physique. Il se peut que l'on connaisse à l'avance le nombre de cases exactes dont on a besoin, dans ce cas les tailles physique et logique se confondent.

Tableaux – Exemple

```c
#include <stdio.h>
#define NB_ELEM_MAX 100

int main(void) {
int i = 0;
    float monTableau[NB_ELEM_MAX]; // taille physique du tableau : 100 nombres décimaux
    int nbValeurs = 0; // taille logique initiale du tableau, nombre de nombres décimaux réellement stockés dans le tableau

    printf("Combien de valeurs souhaitez-vous ajouter ?\n"); // on demande la taille logique
    scanf("%d", &nbValeurs);

    if (nbValeurs > NB_ELEM_MAX) {
        printf("Désolé, mais nous ne pouvons stocker que %d valeurs.", NB_ELEM_MAX);
        return 1; // met fin au programme (au main)
    }

    for (i = 0; i < nbValeurs; i++) { // on remplit les cases de la taille logique
       printf("Veuillez saisir la valeur n°%d\n", i+1);
       scanf("%f", &monTableau[i]);
    }

    for (i = 0; i < nbValeurs-1; i++) { // tous les éléments, sauf le dernier (-1), sont suivis d'une virgule
        printf("%f, ", monTableau[i]);
    }
    printf("%f.", monTableau[nbValeurs-1]); // le dernier, lui, est suivi d'un point
    return 0;
}
```

NB : Pas besoin d'initialiser le tableau s'il a une taille logique. Ce n'est pas grave si les cases en dehors de la taille logique ont une valeur non maitrisée.

Rappel : Sous-programmes - Passage par valeur

Jusqu'à maintenant, lorsque nous passions une variable (`int`, `double`…) en paramètre d'un sous-programme, celui-ci était toujours passé **par valeur**.

Cela veut dire que le sous-programme appelé reçoit une **valeur** pour chaque paramètre au moment de l'appel, qu'elle copie dans une variable locale.

Cette variable n'existe que dans ce sous-programme. Elle peut avoir le même nom que dans le programme appelant.

En passage par valeur, le sous-programme appelé ne peut pas modifier la valeur de la variable d'origine dans le programme appelant. Il travaille sur une copie.

Rappel : Sous-programmes - Passage par valeur - Exemple

```c
void incrementerEtAfficher(int n) {
    n = n + 1;
    printf("%d\n", n); // 3 affiche 4
}

int main(void) {
    int a = 3;
    printf("%d\n", a); // 1 affiche 3
    incrementerEtAfficher(a); // 2

    printf("%d\n", a); // 4 affiche 3 (seule la copie a été modifiée)
    return 0;
}
```

La valeur de la variable **a** du main est copiée dans la variable **n** de la procédure **incrementerEtAfficher()** au moment de son appel (2). Ces deux variables existent dans des **blocs** (d'accolades) différents, c'est pourquoi elles pourraient porter le même nom (**n** aurait pu s'appeler **a** également, car chacune n'existe que dans le bloc d'accolades dans lequel elle a été déclarée).

Rappel : Sous-programmes - Passage par valeur - Exemple

```c
int incrementerEtAfficher(int n) {
    n = n + 1;
    3 printf("%d\n", n); // affiche 4
    return n;
}

int main(void) {
    int a = 3;
    1 printf("%d\n", a); // affiche 3
    4 a = 2 incrementerEtAfficher(a);

    5 printf("%d\n", a); // affiche 4 (seule la copie a été modifiée dans la fonction,
                        // mais la valeur modifiée nous est retournée et nous l'avons affectée à a)

    6 printf("%d\n", incrementerEtAfficher(a)); // affiche 5, la variable a vaut toujours 4

    7 printf("%d\n", a); // affiche 4
    return 0;
}
```

Pour récupérer une donnée passée par valeur qui aurait été modifiée dans un sous-programme, il faut que ce dernier nous la retourne. On peut alors l'affecter à une variable qui prendra alors la valeur modifiée.

Passage par adresse de tableaux

Les tableaux, eux, sont toujours passés par adresse *(mémoire)*.

Nous verrons plus en détails ce que cela veut dire dans les prochains cours.

Pour faire simple, cela veut dire que lorsqu'on passe un tableau en paramètre d'un sous-programme, ce n'est pas sa/ses valeur.s qui est/ sont copiée.s, mais son adresse dans la mémoire centrale. Le sous-programme appelé ne va donc pas travailler sur une copie du tableau mais sur le même tableau en accédant directement au même espace mémoire.

Donc, si on modifie les valeurs d'un tableau dans un sous-programme appelé, les modifications seront également visibles dans le tableau du programme appelant (il s'agit du même tableau).

Sous-programmes - Passage par adresse de tableau

```c
#include <stdio.h>
#define MAX 10

void modifierTableau(int monTableau[MAX]) {
    monTableau[0] = 33;
    printf("%d\n", monTableau[0]); // (2) affiche 33
}
int main(void) {
    int tab[MAX] = {0};
    printf("%d\n", tab[0]); // (1) affiche 0
    modifierTableau(tab);
    printf("%d\n", tab[0]); // (3) affiche 33 !
    return 0;
}
```

Sous-programmes - Tableau en paramètre

- Pour passer un tableau en paramètre d'un sous-programme, on utilise

  la syntaxe suivante :

```
void unSousProgramme(int tableauDEntiers[TAILLE_PHYSIQUE]){ /*...*/}
```

- Pour un tableau à une seule dimension (comme ici), on n'est pas obligés de préciser la taille physique entre les crochets :

```
void unSousProgramme(int tableauDEntiers[]) { /* ... */ }
```

Sous-programmes - Tableau en paramètres - Exemple

```c
#include <stdio.h>
#define NB_ELEM_MAXI 100

int remplissageTableau(int tableauDEntiers[]) {
    int n = 0;
    printf("Combien d'entiers voulez-vous ajouter ?\n");
    scanf("%d", &n);
    if (n > NB_ELEM_MAXI) {
        printf("Erreur : Le tableau est limité à %d éléments.", NB_ELEM_MAXI);
        return -1;
    }
    printf("Entrez les éléments un à un :\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &tableauDEntiers[i]); // On modifie directement le tableau original
    }
    return n; // On peut utiliser le return pour la taille logique
}

void afficherTableau(int tableau[], int taille) { /* ... */ }

int main(void) {
    int tailleLogiqueTableau = 0;
    int monTableau[NB_ELEM_MAXI];
    tailleLogiqueTableau = remplissageTableau(monTableau);
    if (tailleLogiqueTableau > 0) {
        afficherTableau(monTableau, tailleLogiqueTableau);
    }
    return 0;
}
```

Sous-programmes - Tableau en paramètres - Exemple

```c
#include <stdio.h>
#define NB_ELEM_MAXI 100 int remplissageTableau(int tableauDEntiers[]) { /* ... */}

void afficherTableau(int tableau[], int taille) { // Un tableau passé en paramètre est quasiment toujours accompagné de sa/ses taille.s
    printf("Le tableau contient %d éléments : ", taille);
    for (int a = 0; a < taille; a++) {
        printf("%d ", tableau[a]);
    }
}

int main(void) {
    int tailleLogiqueTableau = 0;
    int monTableau[NB_ELEM_MAXI];
    tailleLogiqueTableau = remplissageTableau(monTableau);
    if (tailleLogiqueTableau > 0) {
        afficherTableau(monTableau, tailleLogiqueTableau);
    }
    return 0;
}
```

Tableaux à plusieurs dimensions

Également appelés **matrices**, les tableaux à deux dimensions sont des **tableaux de tableaux**.

```
#define NB_LIGNES_MAX 10
#define NB_COLONNES_MAX 20

int monTableau[NB_LIGNES_MAX][NB_COLONNES_MAX];

// Initialisation à 0 :
int monTableauInitialise[NB_LIGNES_MAX][NB_COLONNES_MAX]={{0}};
```

Chaque case de la première dimension (lignes) est un tableau d'entiers (colonnes).

Tableaux à plusieurs dimensions

On parcourt un tableau à deux dimensions avec deux variables d'itération, généralement appelées **`i`** et **`j`**.

```
#define NB_LIGNES_MAX 10
#define NB_COLONNES_MAX 20 int
int main () {
  int i = 0, j = 0;
  int monTableau[NB_LIGNES_MAX][NB_COLONNES_MAX];
  // initialisation à 0 de toutes les cases :
  for (i = 0; i < NB_LIGNES_MAX; i++) {
     for (j = 0; j < NB_COLONNES_MAX; j++) {
         monTableau[i][j] = 0;
     }
   }
   return 0;
}
```

Initialisation directe : `int monTableau[NB_LIGNES_MAX][NB_COLONNES_MAX] = {{0}};`

Tableaux à plusieurs dimensions et sous-programme

Pour passer un tableau à deux dimension (ou plus) en paramètre d'un sous-programme, nous devons préciser la taille physique de chaque dimension, sauf la première qui reste facultative.

```
void unSousProgramme(int tableau3D[][TP_DIM_2][TP_DIM_3]){/*...*/ }
```

Il est toutefois conseillé de spécifier la taille physique de chaque dimension :

```
void unSousProgramme(int tableau3D[TP_DIM_1][TP_DIM_2][TP_DIM_3]){/*...*/ }
```

Structure – Stockage de plusieurs informations

- Comment stockeriez-vous les informations sur un point dans un repère à 3 dimensions ? (x, y, z)
- Et si nous avons plusieurs points ?
- Comment passer les données d'un de ces points à un sous-programme ?

Structure – Stockage de plusieurs informations

Comment stockeriez-vous les informations sur un point dans un repère à 3 dimensions ? (x, y, z)

```
int x = 3,
    y = 5,
    z = 6;
```

Structure – Stockage de plusieurs informations

Et si nous avons plusieurs points ?

```
int xPoint1 = 3,  yPoint1 = 5,  zPoint1 = 6;
int xPoint2 = 6,  yPoint2 = 2,  zPoint2 = 8;
int xPoint3 = 12, yPoint3 = -3, zPoint3 = 5;
```

…

Imaginez :

- Si chaque point est composé de plus de 3 attributs, 10 par exemple.
- Si nous avons 500 points.
- Comment pourrions-nous parcourir tous les points de notre programme pour les afficher par exemple ?

Comment passer les données d'un point à un sous-programme ?

```c
int estDansLaZone(int x, int y, int z, // point à tester
                  int xCoin1, int yCoin1, int zCoin1,  // coin de la zone
                  int xCoin2, int yCoin2, int zCoin2); // coin opposé de la zone

int main(void) {
    int x = 4, y = 8, z = 2;
    if (estDansLaZone(x, y, z, 0, 0, 0, 10, 5, 5)) { /* ... */ }
    return 0;
}
```

On est obligé de passer les 3 attributs de chaque point au sous-programme, un par un, alors qu'ils représentent un même point, une même entité.

Imaginez :

⊙ Si chaque point est composé de plus de 3 attributs, 10 par exemple.

Structures ou tableaux ?

- **Les structures** : pour regrouper les données qui concernent un même point (leur `x`, leur `y` et leur `z`) dans une même variable.

- **Les tableaux** : pour stocker de façon ordonnée tous les points dans une même variable, afin de pouvoir les parcourir par exemple.

Comme tous les points ont 3 coordonnées, ils sont assez lourds à transporter de fonction en fonction.

Il faut à chaque fois ajouter 3 paramètres x, y, z.

Les structures - Définition d'une structure

De plus, on s'y perd vite avec des noms tels que **`xPoint1`**, **`xPoint2`**...

Les structures permettent de regrouper des variables, de même type ou non, qui concernent une même entité.

Nous allons donc créer un nouveau type **`struct Point`**, regroupant 3 entiers **`x`**, **`y`**, **`z`**.

Nous pourrons alors créer plusieurs variables de type **`struct Point`** (ex : **`p1`**, **`p2`**...), qui contiendront chacune leurs propres coordonnées (ex : **`p1.x`**, **`p1.y`**, **`p2.x`**...).

Les structures - Définition d'une structure

Tous les points auront la même structure, c'est-à-dire le même contenu.

Nous devons **définir** cette structure **`struct Point`** *(le nom est à notre choix)*, afin de spécifier de quoi sera composé chaque point.

```
// De préférence dans les fichiers .h pour une utilisation dans d'autres fichiers
struct Point
{
    int x, y, z;            // coordonnées
    int rouge, vert, bleu; // couleur au format RGB
    char lettre;            // lettre (étiquette)
};
```

Une structure peut contenir plusieurs variables, de types semblables ou non.

😌 **Bonne pratique :** adoptez une syntaxe particulière pour les noms de structures. En commençant pas une majuscule par exemple.

Les structures – Exemple

Maintenant que nous avons défini ce qu'est un point, nous pouvons en créer :

```c
struct Point
{
    int  x, y, z;
    int rouge, vert, bleu;
    char lettre;
};

int main(void) {
    // Déclaration d'un nouveau point de type "struct Point" :
    struct Point monPremierPoint; // si ça avait été un entier, nous aurions écrit : int monPremierPoint;

    monPremierPoint.x = 12; // on accède aux attributs/variables d'une entité de structure avec un "."
    monPremierPoint.y = 3;
    monPremierPoint.z = 8;
    monPremierPoint.rouge = 0;
    monPremierPoint.vert  = 0;
    monPremierPoint.bleu  = 0;
    monPremierPoint.lettre  = 'C';

    return 0;
}
```

😌 **Bonne pratique :** pour distinguer le nom d'un **type** structure et le nom d'une **variable** (entité/instance), les noms de variables ne commencent jamais par une majuscule.

Les structures - Exemple

Ou plus simplement :

```c
struct Point
{
    int  x, y, z;
    int rouge, vert, bleu;
    char lettre;
};

int main(void) {
    // Déclaration et initialisation d'un nouveau point de type "struct Point" :
    struct Point monPremierPoint = { 12, 3, 8, 0, 0, 0, 'C'}; // même ordre que dans la définition
                                                              // x y z rouge vert bleu lettre

    printf("x = %d", monPremierPoint.x); // ⚠ le '=' n'est pas une affectation/instruction, car entre "guillemets"

    return 0;
}
```

Les structures - Exemple

```c
#include <stdio.h>

struct Point {
    float x, y, z; // autre exemple allégé, avec des nombres décimaux
};

int main(void) {
    struct Point monPoint = { 1.0f, 2.0f, 3.0f };

    printf("x: %.2f y: %.2f z: %.2f\n", monPoint.x, monPoint.y, monPoint.z);
    // %.2f (%f) indique que l'on affiche que 2 chiffres après la virgule

    printf("Veuillez entrer les coordonnées x, y, et z d'un point :\n");
    scanf("%f%f%f", &monPoint.x, &monPoint.y, &monPoint.z);

    printf("x: %.2f y: %.2f z: %.2f\n", monPoint.x, monPoint.y, monPoint.z);
    return 0;
}
```

Les structures - Exemple avec sous-programmes

```c
#include <stdio.h>

struct Point {
float x, y, z;
};

struct Point demanderPoint() {
    struct Point p;
    printf("Veuillez entrer les coordonnées x, y, et z d'un point :\n");
    scanf("%f%f%f", &p.x, &p.y, &p.z);
    return p;
}

void afficherPoint(struct Point p) {
    printf("x=%.2f y=%.2f z=%.2f\n", p.x, p.y, p.z);
}

int main(void) {
    struct Point monPoint;
    monPoint = demanderPoint();
    afficherPoint(monPoint);
    return 0;
}
```

Création d'un alias de nom de type - `typedef`

En C, nous pouvons créer/définir un nouveau type, **alias** d'un type existant :

```
typedef typeExistant nouveauType;
```

Exemple :

```
typedef char lettre;

lettre maVariable = 'a'; // est
maintenant équivalent à : char
maVariable = 'a';
```

💡 `typedef`! est surtout utilisé lors de la définition de **structures** afin de ne plus avoir à écrire `struct` à chaque fois.

Les structures - Alias de nom avec typedef - 1/3

On utilise quasiment toujours **`typedef`** lors de la définition d'une structure, afin de ne pas avoir à réécrire à chaque fois le type complet **`struct Point`** :

Nous allons par exemple créer le type "**`Point3D`**", alias du type "**`struct Point`**" :

```c
struct Point {
    int x, y, z;
};

typedef struct Point Point3D;

int main() {
    Point3D monPoint; // est maintenant équivalent à : struct Point monPoint;
    return 0;
}
```

NB : Il est tout à fait possible de reprendre le nom **`Point`** à la place de **`Point3D`** :

```c
typedef struct Point Point;
```

Les structures - Alias de nom avec typedef - 2/3

Il est possible de combiner la définition de la structure avec l'instruction **`typedef`** :

```
typedef struct Point {
    int x, y, z;
} Point3D;

int main(void) {
    Point3D monPoint; // est maintenant équivalent à : struct Point monPoint;
    return 0;
}
```

Les structures - Alias de nom avec typedef - 3/3

Il n'est alors même plus utile de donner un nom à la structure, seul l'alias donne un nom de type :

```
typedef struct {// la structure de base n'a pas de nom, uniquement un alias
    int x, y, z;
} Point3D;

int main(void) {
     Point3D monPoint;
     return 0;
}
```

Structures regroupées et tableaux

Les structures peuvent regrouper des données de tous types, mélangés. Y-compris des structures ou même des tableaux (que nous allons voir juste après).

```c
#include <stdio.h>

typedef struct {
    unsigned char rouge, vert, bleu;
    // Nous aurions pu choisir le type int (4 octets) car nous voulons des entiers pour les taux de rouge, vert et bleu.
    // Mais unsigned char permet lui aussi de stocker un nombre entier (table ASCII), sur 1 seul octet au lieu de 4.
    // Les taux RGB vont de 0 à 255, or un octet permet de stocker 256 valeurs (2^8 = 256) !
    // Nous faisons donc cela pour économiser la mémoire (gain de 9 octets ici).
} Couleur;

typedef struct {
    int coord[3] ; // tableau de 3 coordonnées
    Couleur couleur; // la structure Couleur doit être définie avant pour être connue
} Point;

int main(void) {
    Point monPoint; // déclarer un point du type de la structure Point
    monPoint.coord[0] = 3; // affecter la valeur 3 à la première coordonnée du tableau coord de la structure monPoint
    monPoint.couleur.rouge = 195; // affecter la valeur 195 à rouge de la structure couleur qui se trouve dans la structure monPoint
    // ...
    return 0;
}
```

Tableau de structures

Un tableau peut regrouper des structures. En reprenant l'exemple précédent, on peut regrouper des points dans un tableau d'une taille physique définie.

```
#define TAILLE_PHYSIQUE 10

int main(void) {
    Point monPoint[TAILLE_PHYSIQUE]; // déclarer un tableau de points, chaque case étant du type de la structure Point
    for (int i = 0; i < TAILLE_PHYSIQUE; i++) {
       // monPoint[i] correspond à la structure de la case i parcourue dans le tableau monPoint
       monPoint[i].coord[0] = 3; // affecter la valeur 3 à la première coordonnée coord de la case i du tableau monPoint
       monPoint[i].couleur.rouge = 195; // affecter la valeur 195 à rouge de la structure couleur qui se trouve dans la case i du tableau monPoint
       printf("%d %d\n", monPoint[i].coord[0], monPoint[i].couleur.rouge); // puis on affiche les valeurs de la case parcourue
    }

    // ...
    return 0;
}
```

Résumé : structures et tableaux

- Une **structure** est un **type** de **structuration des données** permettant de **regrouper plusieurs données**/*attributs/variables/valeurs*, de type sembable ou non, **représentant une même entité**.

  `xPoint1`, `yPoint1` seront réunis dans une structure `Point p1`. Chaque `Point` aura son propre `x` et son propre `y`.

- Un **tableau** permet de **regrouper plusieurs données de même type** de façon **ordonnée** (accès par **indice**). Ils permettent de créer une **collection** d'éléments facilement parcourables depuis une unique variable.

Une seule variable (le tableau) pour stocker plusieurs éléments. Possibilité d'accéder au Nième élément, et donc de parcourir les éléments (boucle for).