

Algorithmique et programmation – Cours 6.

Structurer les données.

Pierre Boudes

17 octobre 2012



Mémoire et tableaux en C

Déclaration d'un tableau en C

Pour aller plus loin en C

Les tableaux : une structure de données

Les chaînes de caractères

Les enregistrements (struct)

Déclaration d'un type utilisateur struct

Utilisation d'un type utilisateur struct

Conclusion

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :
 - La déclaration `int toto`; aura pour effet de réserver l'espace mémoire nécessaire au stockage d'un entier.
 - Dans la suite du programme, l'adresse de cet espace mémoire sera utilisée partout où il est fait référence à cette variable (identificateur `toto`).
 - C'est le codage machine des entiers en binaire qui sera employé pour manipuler cette donnée.

Remarque.

La taille d'un `int` est en principe exactement celle d'un mot mémoire, c'est à dire 4 ou 8 octets.

Déclarations multiples et répétitives ?

```

int main () {
    int i;
    /* repeter des declarations ? */
    for (i = 0; i < 1024; i = i + 1) {
        int X_i; /* marchera pas */
    }
    int somme = 0;
    /* saisie */
    for (i = 0; i < 1024; i = i + 1) {
        printf("X_i_?");
        scanf("%d", &X_i);
    }
    /* calcul */
    for (i = 0; i < 1024; i = i + 1) {
        somme = somme + X_i;
    }
}

```

Solution : un tableau

```

int main () {
    int i;
    int X[1024]; /* 1024 variables ! */
    int somme = 0;
    /* saisie */
    for (i = 0; i < 1024; i = i + 1) {
        printf("X[%d]□?", i);
        scanf("%d", &X[i]);
    }
    /* calcul */
    for (i = 0; i < 1024; i = i + 1) {
        somme = somme + X[i];
    }
    return EXIT_SUCCESS;
}

```

Déclaration d'un tableau statique unidimensionnel

- Du point de vue logiciel la mémoire se présente comme une succession d'octets, numérotés par les entiers à partir de 0.

Adresses :	0	1	2	...
octets (valeurs) :	01000110	11010111	00000001	...

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto [3];
```

Adresses :	...	344			348				352			...
Valeur :	...	1...0			0...1			1...1			...	
Identificateur :	...	toto[0]			toto[1]			toto[2]			...	

- La taille doit être connue à de la compilation
- Les cases sont accessibles, comme s'il s'agissait de variables, à l'aide des identificateurs `toto[0]`, `toto[1]`, `toto[2]`
- La numérotation commence à zéro. Si n est la taille la dernière case est donc numérotée $n - 1$.

Attention !

Il ne faut jamais accéder à une case au delà de la numérotation : `toto[3]`, `toto[-1]`, etc. Le compilateur ne vous préviendra pas de votre erreur, mais le programme va boguer.

L'erreur d'exécution `segmentation fault` signifie que le programme a effectué un accès à une case mémoire qui ne lui était pas réservée (mais il faut beaucoup s'écarter des bons indices du tableau).

Premier exemple

```
int main()  
{  
    /*Declaration et initialisation de variables*/  
    int tableau[3] = {3,5,8};  
  
    tableau[0] = 3; ← inutile  
    tableau[1] = 5; ← inutile  
    tableau[2] = tableau[0] + tableau[1]; ← inutile  
  
    return EXIT_SUCCESS;  
}
```


Second exemple

```

int main()
{
    /* Declaration et initialisation de variables */
    int tab[3] = {3,5,8};
    int i; /* var. de boucle */

    for (i = 0; i < 3; i = i + 1) /* pour chaque case */
    {
        printf("tab[%d] = %d\n", i, tab[i]);
    }
    return EXIT_SUCCESS;
}
    
```

Trace du second exemple

```

1  int main()
2  {
3      /* Declaration et initialisation de variables */
4      int tab[3] = {3,5,8};
5      int i; /* var. de boucle */
6
7      for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8      {
9          printf("tab[%d]=%d\n", i, tab[i]);
10     }
11     return EXIT_SUCCESS;
12 }

```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5
10				2	
9					tab[2] = 8
10				3	
11	Renvoie EXIT_SUCCESS				

Pour aller plus loin ~~✎~~

- La taille d'un tableau statique gagne à être fixée par une constante symbolique (`#define N 3`).
- L'identificateur du tableau (*ie* `tab` dans la déclaration `int tab[3];`) est lui même une variable. Sa valeur est l'adresse de la première case du tableau `tab[0]`.
 - Les variables dont la valeur est une adresse s'appellent des **pointeurs** ;
 - La notation *esperluette*, `&x`, donne accès à l'adresse d'une variable ;
 - La notation *étoile*, `*tab`, ne s'applique qu'à une adresse, elle donne alors accès à la valeur contenue à cette adresse.
 - Les expressions `tab[i]` et `*(tab + i)` sont identiques en C.
- Pour les grands tableaux, il est nécessaire d'allouer dynamiquement l'espace mémoire.
- Si on ne veut pas s'ennuyer avec les pointeurs (et l'allocation) on peut emballer nos tableaux statiques dans des *struct*.



La structure de donnée tableau

Les tableaux sont surtout une manière très naturelle d'organiser des données de même type. *Beaucoup d'algorithmes utilisent des tableaux.*

- On peut accéder très rapidement à n'importe quel élément si on connaît son indice, on peut aussi très simplement parcourir un tableau ;
- supprimer ou insérer un élément sont des opérations plus lentes, comme y chercher un élément si le tableau n'est pas préalablement classé dans un certain ordre. **Connaissez vous la recherche dichotomique ?**
- Classer un tableau peut prendre plus de temps, parce que classer des éléments prend du temps qu'ils soient sous forme de tableau ou non. **Connaissez vous un algorithme de tri (rangement sous forme classée) ?** Exploite t'il la structure de tableau ou une autre structure ?

Les chaînes de caractères

- Une chaîne de caractère est un tableau de caractère, terminé par le caractère spécial de code ascii zéro (on parle de sentinelle), `\0`.

- Plutôt que d'écrire :

```
char nom[7] = {'m', 'o', 'n', '\u00a0', 'n', 'o', 'm'};
```

- on peut utiliser directement :

```
char nom[8] = "mon\u00a0nom";
```

- Le huitième caractère, `\0` sert à délimiter le texte.

- On peut même écrire :

```
char nom[] = "mon\u00a0nom"; /* 8 cases */
```

- Avec `printf` et `scanf` on utilise `%s`.
- Pour les entrées on utilise surtout `fgets` :

```
fgets(nom, 64, stdin); /* lit 63 car. maxi */
```

Autre problème de données multiples

Entrer la première fraction : -1 / 42

...

Quel type de sortie donner à la fonction de saisie ?

```
int [2] saisir_fraction(); /* error: (syntaxe) */
int * saisir_fraction(); /* probleme avec la memoire */
```

Il nous faut *emballer* ces deux entiers.

Les enregistrements ou « struct »

- Une structure *empaquette* plusieurs valeurs nommées et s'utilise comme un **type**.
- Chacune des valeurs est accessible à l'aide d'un nom, choisi au moment de la déclaration de la structure. On parle des **champs** de la structure.
- Un type structure doit être déclaré avant d'être utilisé. Ne pas confondre : **déclaration de variable** (dans une fonction), **déclaration de type structure** et **déclaration de fonction**.
- Les valeurs des champs sont accessibles individuellement à l'aide de la **notation pointée**.
- On peut aussi manipuler globalement la valeur d'une donnée de type structure. Par exemple : faire une affectation entre variables d'un même type structure (copie les champs, y compris les tableaux statiques).

Déclaration d'un type utilisateur struct

```

struct bulletin_s {
    double temperature; /* temperature de l'air */
    int force;          /* force du vent (Beaufort) */
}; /* <-- attention ';' (cas particulier) */
  
```

- Cette déclaration est placée en dehors de toute fonction, entre les définitions de constantes symboliques (`#define ...`) et les déclarations de fonctions utilisateur.
- L'effet de cette déclaration est de signaler au compilateur qu'un nouveau type est disponible et comment il peut être utilisé (liste des champs). Aucun espace mémoire n'est réservé à ce moment (ce n'est pas une déclaration de variable).
- **Attention au point-vigule final.**

Utilisation d'un type utilisateur struct : variables

```
int main()  
{  
    struct bulletin_s x = {0.5, 4};  
    struct bulletin_s y;  
  
    y = x; /* copie globale */  
    x.temperature = 13.4; /* modif. d'un champ */  
    ...  
}
```

- Initialisation : syntaxe proche de celle des tableaux.
- la déclaration d'une variable de type structure reprend le mot clé struct et le nom donné à la structure.
- On accède aux éléments d'une structure à l'aide de la notation pointée : `nom_variable.nom_champ`

Utilisation d'un type utilisateur struct : fonctions

```
/* declaration de fonctions utilisateur */  
struct bm_s moyenne_bm(struct bm_s x, struct bm_s y);
```

On emploie `struct nom_struct`, comme pour une déclaration de variable.

```
/* definitions des fonctions utilisateur */  
struct bm_s moyenne_bm(struct bm_s x, struct bm_s y)  
{  
    struct bm_s nouveau_bm;  
  
    nouveau_bm.temperature = (x.temperature  
                             + y.temperature) / 2.0;  
    nouveau_bm.force = (x.force + y.force) / 2;  
  
    return nouveau_bm;  
}
```



Avec typedef

On utilise souvent **typedef** pour se passer du mot clé struct et donner un véritable nom de type à la structure.

```
typedef struct bm_s {  
    double temperature;  
    int force;  
} bm_t; /* <-- bm_t === struct bm_s */  
  
bm_t moyenne_bm(bm_t x, bm_t y);
```

Intérêt des structures (enregistrements)

Intérêt des structures :

- *lisibilité* : regrouper un ensemble de données dans un même type, nommé de façon explicite, facilite la relecture du code ;
- *augmente les possibilités* : les structures permettent d'écrire des fonctions qui retournent plusieurs valeurs, en l'absence de pointeurs.
- *modularité* : on peut rajouter des champs très facilement, avec très peu de modifications.
- *Incontournables* : les langages orientés objets généralisent la notion de structure. Un objet est une structure dont les champs peuvent être aussi bien des données que des fonctions (hors programme).



Erreurs communes

- Message d'erreur *étrange* :

```
7 struct bm_s {
8     double temperature; /* temperature de l'air */
9     int force;          /* force du vent (Beaufort)
10 }
11
12 /* Declaration des fonctions utilisateur */
13 void afficher_bm(struct bm_s);
```

14: error: two or more data types in declaration specifiers

Oubli du point-virgule!

- Champ inexistant :

```
22     x.toto = 3; /* erreur: pas de champs toto */
```

prog.c: In function 'main':

prog.c:22: error: 'struct bm_s' has no member named 'toto'



Conclusion

- Un tableau est un **lot de déclarations identiques**, numérotées.
- Une structure **regroupe des variables de différents types**, dûment nommées.
- Pour les tableaux (ou les chaînes) comme pour les structures, on ne peut pas utiliser les opérations usuelles : comparaisons, test d'égalité, opérations arithmétiques. Il faut introduire des fonctions spécifiques.
- On évitera de passer des tableaux à des fonctions (pointeur, mémoire), on les emballera avant dans des structures :

```
#define TAILLE_MAX 1024
typedef struct tableau_s {
    int taille; /* maximum TAILLE_MAX */
    int t[TAILLE_MAX];
} tableau_t;
```