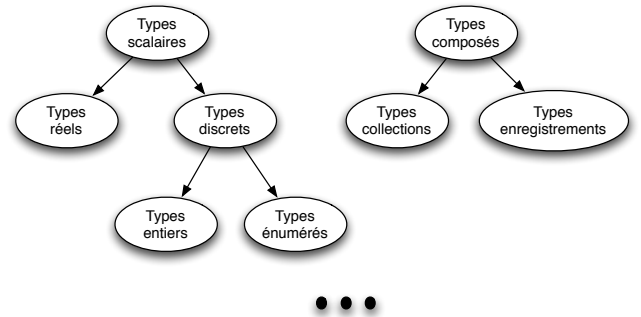


Luc Fabresse
luc.fabresse@imt-nord-europe.fr



Plan

- 1 Énumérations
- 2 Tableaux
- 3 Pointeurs
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

Types énumérés

Syntaxe `enum <nom type> { <liste valeurs> }`

Effet Définit un nouveau type de données nommé <nom type> dont les valeurs possibles sont données en extension

Exemple

```
enum jour_semaine {
    lundi,
    mardi,
    mercredi,
    jeudi,
    vendredi,
    samedi,
    dimanche
};

enum jour_semaine un_jour = lundi; // déclaration de la variable un_jour
```

Types énumérés (2)

Variante

```
enum {
    lundi,
    mardi,
    mercredi,
    jeudi,
    vendredi,
    samedi,
    dimanche
} un_jour; // déclaration de la variable un_jour

un_jour = lundi;
```

Types énumérés (3)

Règles :

- Chaque valeur d'un type énuméré est associé à une valeur entière
- La valeur entière peut être décidée par le programmeur
- Dans le cas contraire :
 - La 1^{re} valeur est associée à la valeur 0
 - Une valeur est associée à la valeur de son prédécesseur incrémenté de 1

Exemple

```
enum jour_semaine {
    lundi = 1,
    mardi,
    mercredi,
    jeudi,
    vendredi,
    samedi,
    dimanche
};

enum jour_semaine un_jour;

printf("num du jour dans la semaine : %u\n", un_jour);
```

Plan

- 1 Énumérations
- 2 Tableaux
- 3 Pointeurs
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

Tableaux unidimensionnels

Déclaration

Syntaxe `<type> <nom variable> [taille]`

Effet Déclare une variable tableau ayant taille éléments de même type

- indice du 1^{er} element est 0
- indice du dernier element est taille-1

Exemple

```
// tableau de 20 entiers
int notes[20];

// tableau de 12 caractères
char nom_famille[12];
```

Syntaxe : <type> <nom variable> [taille_dimension1] [taille_dimension2]

Autant de [] que de dimensions

Exemple

```
/* tableau de réels à 2 dimensions :
 * 5 lignes et 4 colonnes */
float matrice[5][4];
```

Opération d'indexation

Syntaxe <nom variable>[<indice>]

- <indice> est une expression à valeur entière

Effet permet d'accéder à l'élément du tableau rangé à la position <indice>

Exemple

```
// tableau de 20 entiers
int notes[20];

notes[0] = 0;
notes[1] = notes[2] = notes[3] = notes[4] = 0;

matrice[0][0] = 10.1;
matrice[0][1] = matrice[0][2] = matrice[0][3] = 19.4;
```

Opération d'initialisation

Syntaxe <type> <nom variable>[<taille>] = { <liste valeurs> }

Effet permet d'accéder à l'élément du tableau rangé à la position <indice>

Exemple

```
int tabint[3] = {1, -5, 6};
float tabfl[] = {0.45, 6.78, 789.5};
float matrIdent[2][2] = {{1.0, 0.0}, {0.0, 1.0}};
```

Exemple

```
char message[5] = "Hello";
int l = 0;
char c = message[l];

while( c != '\0' ) {
    c = message[l];
    l++;
}

printf("longueur >%s< = %d\n", message, l);
```

'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	------

Exemple

```
longueur chaine >Hello< = 6
```

<string.h>

- strcmp
- strcpy
- ...

https:

[//stackoverflow.com/questions/4415524/common-array-length-macro-for-c](https://stackoverflow.com/questions/4415524/common-array-length-macro-for-c)

Macro pour retrouver les tailles des dimensions d'un tableau

```
#define ARRAY_LENGTH(x) ((sizeof(x)/sizeof(0[x])) / ((size_t)(!(sizeof(x) % sizeof(0[x])))))

int mat[4][5][6];
printf("taille = %li\n", ARRAY_LENGTH(mat));
printf("taille = %li\n", ARRAY_LENGTH(mat[1]));
printf("taille = %li\n", ARRAY_LENGTH(mat[1][1]));
```

```
taille = 4
taille = 5
taille = 6
```

4 définitions équivalentes de la même fonction :

```
void f(int b[10][15][20]);
void f(int b[100][15][20]);
void f(int b[][15][20]);
void f(int (*b)[15][20]);
```

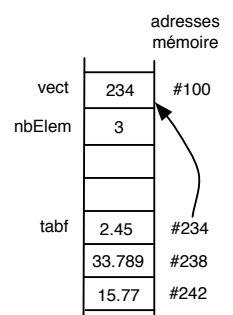
Remarque :

- la taille de la dimension la plus interne du tableau peut être omise dans la signature
- lorsqu'un tableau est passé en paramètre d'une fonction, le compilateur passe toujours l'adresse du premier élément (int* dans l'exemple précédent)
- Le résultat d'une fonction ne peut pas être un tableau

Exemple

```
float sommeVect(float vect[], int nbElem) {
    int i;
    float sigma;
    for (i=0, sigma = 0; i < nbElem; i++)
        sigma += vect[i];
    return sigma;
}

int main(void) {
    float tabf[3] = {2.45, 33.789, 15.77};
    float somme;
    somme = sommeVect(tabf, 3);
    return 0;
}
```



- 1 Enumérations
- 2 Tableaux
- 3 **Pointeurs**
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

Définitions

- Un pointeur est une donnée (constante ou variable) contenant l'adresse mémoire d'une donnée
- Le type d'un pointeur est lié au type de donnée pointée (type de base, type tableau, ...)
- La valeur NULL représente une adresse mémoire invalide

Utilisations

- Allocation dynamique de mémoire
- Simulation du passage par variable
- Structure de données dynamiques (listes chaînées, arbres, ...)

Pointeurs : déclaration

Syntaxe `<type> * <nom variable>`

- Comme une variable avec * précédant le nom

Effet Déclare une variable contenant une **adresse** donc n'alloue l'espace que pour une adresse.

Exemple

```
int *pi; /* pointeur vers un int */
char *pc; /* pointeur vers un char */
```

Pointeurs et opérateurs

L'opérateur unaire &

Syntaxe `&<nom variable>`

Effet retourne l'adresse de la variable `<nom variable>`

Exemple `&x`

L'opérateur unaire *

Syntaxe `*<nom variable>`

Effet retourne la valeur stockée à l'adresse désignée par la variable `<nom variable>`

Exemple `*x`

Pointeurs : exemples

Pointeurs (cf. MOOC C)

Exemples d'initialisation de pointeurs

```
char i = 'z';
char t[3] = {'a', 'b', 'c'};

char *p1, *p2;

p1 = &i; // p1 <- l'adresse de i
p2 = &t[2]; // p <- l'adresse de t[2]
```

Accès à l'entité référencée

```
*p1 = 'x'; // i <- 'x' car p==&i
*p2 = 'd'; // t[2] == 'd'
```

		adresses mémoire
i	'z'	#100
		#101
t[0]	'a'	#102
t[1]	'b'	#103
t[2]	'c'	#104
p1	100	#238
p2	104	#242

<https://codecast.france-ioi.org/v2/player?id=1479999497689>

Opérations sur les pointeurs

Addition/soustraction d'un entier n à un pointeur

ajout/retrait de n fois la taille de l'entité désignée à l'adresse du pointeur

Accès à l'entité référencée

```
char lettres[]={'H','E','L','L','O'};
char *p;
p = &lettres[0];
p+= 2; /* p prend pour valeur &lettres[2] */
*p = 'l'; /* équivalent à lettres[2] = 'l' */

// Incréméntation d'un pointeur
p++; /* p prend pour valeur &lettres[3] */
*p = 'l'; /* équivalent à lettres[3] = 'l' */
```

Pointeurs passés en paramètres

Rappel :

En C, il n'existe que le mode de passage par valeur

Le passage par variable :

- permet d'agir sur les paramètre effectifs
- peut être obtenu en passant des pointeurs (adresses) en paramètre

Exemple

```
void incrementer(int *x) {
    (*x)++;
}

int main(void) {
    int a = 10;
    incrementer(&a); // on passe l'adresse de a
    return 0;
}
```

Les valeurs passées en paramètre via des pointeurs peuvent être modifiées et sont donc à la fois **paramètre** et **résultat**.

Une variable tableau c'est :

un pointeur constant sur le premier élément du tableau

Exemple

```
int t[10];
int * p;
p = t; /* équivalent à p = &t[0] */
*t; /* équivalent à *p et à t[0] */
*(t+1); /* équivalent à t[1] */
```

Sont interdits (car pointeur **constant**) :

- l'affectation : `t = &i`
- l'incrément : `t++`
- la décrément : `t--`

Exemple

```
// Déclaration
char tab[3] = {'a','b','c'};
char *p[3] = {tab, tab+1, tab};

// Accès aux objets référencés par les éléments
*p[1] = 'x'; // équivalent à tab[1]='x'

// Modification des éléments
p[1]++; // p[1] == &tab[2]
```

tab[0]	'a'	#100
tab[1]	'b'	#101
tab[2]	'c'	#102
		#103
p[0]	100	
p[1]	101	#238
p[2]	100	#242

Pointeurs sur fonction

Exemple

```
#include <stdio.h>

float plus(float a, float b) { return a+b; }
float moins(float a, float b) { return a-b; }
float mult(float a, float b) { return a*b; }
float diviser(float a, float b) { return a/b; } // bl=0

int main(void) {
    float (*operation)(float, float); // un pointeur sur fonction
    float a, b; char op;

    printf("enter un nombre : "); scanf("%f", &a);
    operation = NULL;
    while(operation == NULL) {
        printf("Enter operation sign (+, -, *, /) : "); scanf("%c", &op);
        switch(op) {
            case '+': operation = plus; break;
            case '-': operation = moins; break;
            case '*': operation = mult; break;
            case '/': operation = diviser; break;
            default: ;
        }
        printf("enter un nombre : "); scanf("%f", &b);
        printf("resultat : %f\n", operation(a, b));
    }
    return 0;
}
```

Comment lire des déclarations avec pointeurs ? (1)

cf. <http://blog.parr.us/2014/12/29/how-to-read-c-declarations/>

Quelle différence entre :

```
int *a[3];

// et

int (*a)[3];
```

Comment lire des déclarations avec pointeurs ? (2)

cf. <http://blog.parr.us/2014/12/29/how-to-read-c-declarations/>

Règle :

- 1 Start at the variable name (or innermost construct if no identifier is present)
- 2 Look right without jumping over a right parenthesis; say what you see
- 3 Look left without jumping over a parenthesis; say what you see
- 4 Jump out a level of parentheses if any and restart at #2

`int *a[3]`

a, (right) array of 3, (left) pointer to, (right) nothing, (left) int
a is an array of 3 pointers to int

`int (*a)[3]`

a, (right) nothing, (left) pointer to, (right) array of 3, (left) int
a is a pointer to an array of 3 ints

Structures : déclaration

Définition

Ensemble de plusieurs variables, parfois de type différents, regroupées sous un seul nom pour les traiter en bloc

Exemple

```
// Déclaration d'un type structure
struct livre { // nom de la structure
    char Titre[40]; // champs
    int Annee;
    int Cote;
};

// Déclaration d'une variable de type structure
struct livre langageC;
```

Plan

- 1 Énumérations
- 2 Tableaux
- 3 Pointeurs
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

Utilisation de l'opérateur « . » (notation pointée)

```
struct livre l;
l.Annee = 1983;
strcpy(l.Titre, "Le langage C");
```

Utilisation de l'opérateur « -> » avec un pointeur sur structure

```
struct livre l;
struct livre *pl = &l;
(*pl).Annee = 1990;
pl->Annee = 1991; /* équivalent à (*pl).Annee = 1991 */
```

Initialisation : possibilité d'affecter tous les champs en une seule fois

```
struct livre l1 = {"Le langage C", 1983, 1023 };
```

Affectation entre structures : recopie des champs

```
struct livre l2;
l2 = l1; /* après affectation l2.Année = 1983, ... */
```

Remarques :

- Les champs de type pointeur sont partagés en cas d'affectation
- La taille mémoire d'une structure = la somme des tailles de tous ses champs

Affectation entre structures : recopie des champs

```
enum Mois {jan = 1, fev, mars, avr, mai, juin, juil, aout, sept, oct, nov, dec};

struct Date {
    int jour;
    enum Mois mois;
    int annee;
};

int main(void) {
    struct Date naissance;
    naissance.jour = 4;
    naissance.mois = juil;
    naissance.annee = 1990;
}
```

Tableaux de structures

Déclaration

```
struct complexe {
    float reel;
    float imaginaire;
};

/* tableau de 100 éléments de type struct complexe */
struct complexe tabComplexe[100];
```

Accès aux champs des éléments

```
tabComplexe[1].reel = 5;
tabComplexe[1].imaginaire = 10;
```

Pointeurs sur structure passés en paramètres

Utiliser un pointeur sur la structure afin :

d'autoriser la modification des champs de la structure passée en paramètre

Accès aux champs des éléments

```
void mult(struct complexe * pc, float m) {
    pc->reel *= m;
    pc->imaginaire *= m;
}

int main(void) {
    struct complexe c = {10,3};
    mult(&c, 10);
    /* c.reel = 100, c.imaginaire = 30 */
    return 0;
}
```

L'instruction typedef

Syntaxe typedef <type existant> <nouveau nom>

Effet déclaration du type <nouveau nom>

Exemple

```
enum MesMois {jan = 1, fev, mars, avr, mai, juin,
    juil, aout, sept, oct, nov, dec};

typedef enum MesMois Mois; // déclaration du type MesMois

struct DateSimple {
    int jour;
    Mois mois; // Simplification !
    int annee;
};

typedef struct DateSimple Date;

int main(void) {
    Date naissance; // Simplification !
    Date armistice = {11,nov,1918};

    naissance.jour = 1;
    naissance.mois = jan;
    naissance.annee = 1970;

    return 0;
}
```

Structures passées en paramètres

Passage par valeur = recopie des valeurs des champs

```
#include <stdio.h>

struct complexe {
    float reel;
    float imaginaire;
};

struct complexe plus(struct complexe a, struct complexe b){
    struct complexe resultat;
    resultat.reel = a.reel + b.reel;
    resultat.imaginaire = a.imaginaire + b.imaginaire;
    return resultat;
}

void affiche(struct complexe a) {
    printf("(%.2f + i * %.2f) ",a.reel,a.imaginaire);
}

int main(void) {
    struct complexe c1 = {0.0,-1.0},c2 = {2.0,0.0},c3;
    c3 = plus(c1, c2);
    affiche(c1);printf(" + ");affiche(c2);printf(" = ");affiche(c3);printf("\n");
    return 0;
}
```

```
1 (0.00 + i * -1.00) + (2.00 + i * 0.00) = (2.00 + i * -1.00)
```

Plan

- 1 Énumérations
- 2 Tableaux
- 3 Pointeurs
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

Plan

- 1 Énumérations
- 2 Tableaux
- 3 Pointeurs
- 4 Enregistrements (structures)
- 5 Définition de nouveaux types
- 6 Le code source ça coule de source
- 7 Exercices

