

Langage C

ENSTA - TC 1ère année

Goran Frehse


U2IS

2018-2019

`goran.frehse@ensta-paristech.fr`

basé sur le cours de François Pessaux

Constantes, macros et préprocesseur (2)



- Possibilité de macros « paramétrées » :
`#define TWICE(x)((x)* (x))`
- Remplacement **textuel** en substituant x par le texte argument lors de l'utilisation.
- `Bla TWICE(ah bon ?) →`
`Bla ((ah bon ?) * (ah bon ?))`
-  Syntaxe : pas d'espace entre le nom de la macro et la parenthèse ouvrante (définition **et** utilisation).
- Possibilité d'arguments multiples : `#define MUL(a,b)((a)* (b))`
- `Bli MUL(((15)) , autre truc) →`
`Bli (((((15))) * (autre truc))`

- Vous en avez assez de taper `unsigned long int` ?
- Vous devez changer partout `int` en `"unsigned int"` ?
- \Rightarrow Définissez un **alias**.
- `typedef unsigned long int uli_t ;`
 - ▶ `uli_t` : nom, abréviation de `unsigned long int`.
 - ▶ Utilisable ensuite comme nom de type : `uli_t i = 42 ;`
- Ne définit **pas** un nouveau type !
- De manière générale : `typedef <type existant> <nom>`
- Fonctionnera avec les types que nous verrons plus tard.

Contrainte de type (transtypage, cast)


- Parfois besoin de changer le type selon lequel une donnée est « vue ».
 - float → int, unsigned int → long int, ...
- ⇒ Utilisation d'un **cast** : (nom-type) expression.

```
float v = 3.14159 ;  
int i ;  
unsigned long j = 3000000 ;  
  
i = (int) v ;  
j = 15 * ((unsigned long) (i + 1)) + j ;
```

-  Possibilité de **perte** / **corruption** d'information ! (c.f. sides 5 et 7).
-  « *Conversion* » entier ↔ flottant : **change** la représentation interne.
- Autres « *conversions* » : ne changent pas la représentation, juste la façon de **considérer** cette représentation.

Les tableaux

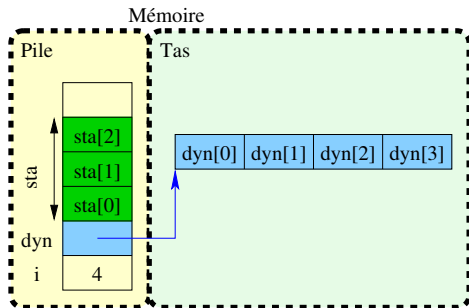
« Rappels, remarques »

- Tableau =
 - ensemble de « cases » mémoire **consécutives**,
 - du **même type**,
 - dénoté par **un seul** nom de variable.
 - Accès à 1 case particulière (« élément ») par **indexation** : `t[3]`.
 -  Contrairement à Python, pas d'indexation avancée
 - Le type d'un élément détermine la taille d'une « case ».
 - \Rightarrow Si on connaît où se trouve la 1^{ère} « case », (« **base** ») on trouve facilement l'endroit où se trouve la $i^{\text{ème}}$ « case » en mémoire.
 - Numérotation des cases (« **indices** ») commence à **0** !
 - Élément _{i} à l'octet : « base » + $i \times$ taille d'un élément.
- \Rightarrow Accès aux éléments **rapide** – en temps constant !

« Tableaux et mémoire »

- Deux sortes de tableaux :

- **Statiques** : taille **connue** à la **compilation**.
→ gérés à la compilation comme des variables « *standard* ».
- **Dynamiques** : taille déterminée à l'**exécution**.
⇒ Nécessite une fonction d'allocation de mémoire.
⇒ Cours ultérieur : uniquement des tableaux statiques pour commencer.



```
int i ;  
/* Blabla ... i ← 4 */  
/* Dynamique. */  
int dyn[] = «allouer i cases»  
/* Statique. */  
int sta[3] ;
```

« Déclaration de tableaux statiques en C »

- *type nom* [*taille*] ;
- La *taille* doit être connue à la compilation : constante entière littérale (plus de liberté en C99).
- `int t[5] ;` *t* est un tableau de 5 entiers.
- On préfère nommer la taille à mettre un littéral : plus facile pour changer la taille et les traitements qui en dépendent.
 - ▶ Utilisation d'un `#define`.
 - On change juste la valeur d'initialisation du `define`.

```
#define DIMENSION (3)
int point[DIMENSION] ;
for (i = 0; i < DIMENSION; i++) calcul (point[i]) ;
```

« Déclaration de tableaux statiques en C »

```
int f (...)
{
    int dimension ;
    ...
    dimension =
        truc + machin * fact (bidule + chose) ;
    int point[dimension] ;
    for (...) point[i] = blabla ;
    ...
}
```

NON !



Initialisation de tableaux

- Lorsqu'on **déclare** un tableau, il n'est pas initialisé.
- Il contient « *ce qu'il y avait dans la mémoire* ».
- Comme pour les autres variables, il faut l'initialiser :
⇒ donner une valeur à chaque case.
- Initialisation case par case :

```
float t[3] ;  
t[0] = 1.0 ;  
t[1] = 1.5 ;  
t[2] = 2.0 ;
```

- Tableaux statiques : on peut initialiser d'un coup **à la définition** :

```
float t[3] = { 1.0, 1.5, 2.0 } ;
```

- `t = { 2.0, 4.7 } ;` **ne marche pas !**

- Pour effectuer une opération sur un tableau, utiliser une boucle pour le parcourir.
- Boucle sur les indices de 0 à « *taille du tableau - 1* ».

```
int t[SIZE] ;
int i = 0 ;
while (i < SIZE) {
    do_something (t[i]) ;
    i++ ;
}
```

- Exemple : initialisation

```
int t[SIZE] ;
int i ;
for (i = 0; i < SIZE; i++) t[i] = i * i ; /* t[i] = i2. */
```

- Boucle for très pratique!

Petit retour sur les chaînes de caractères string

- En C : chaîne = un tableau de caractères...
- ...terminé par le caractère `'\0'`.

str.c

```
#include <stdio.h>
int main ()
{
    int i ;
    char foo[] = "stupefix" ;
    for (i = 0; foo[i] != '\0'; i++)
        printf ("%c ", foo[i]) ;
    printf ("\n") ;

    return (0) ;
}
```

```
mymac:/tmp$ gcc str.c -o str
mymac:/tmp$ ./str
s t u p e f i x
```

Piratage par dépassement de limite

```
#include <stdio.h>
int main ()
{
    char* noms[2];
    char* motsdepasse[2];
    noms[0]="Alice";
    noms[1]="Bob";
    motsdepasse[0]="secret";
    motsdepasse[1]="confidentiel";

    char* gentil = noms[1];
    printf("noms[1]: %s\n", gentil);
    char* mechant = noms[1]+4;
    printf("noms[1]+4: %s\n", mechant);

    return (0) ;
}
```

Sortie :

```
noms[1]: Bob
noms[1]+4: secret
```

Piratage par dépassement de limite

```
#include <stdio.h>
int main ()
{
    char* noms[2];
    char* motsdepasse[2];
    noms[0]="Alice";
    noms[1]="Bob";
    motsdepasse[0]="secret";
    motsdepasse[1]="confidentiel";

    char* gentil = noms[1];
    printf("noms[1]: %s\n", gentil);
    char* mechant = noms[1]+4;
    printf("noms[1]+4: %s\n", mechant);

    return (0) ;
}
```

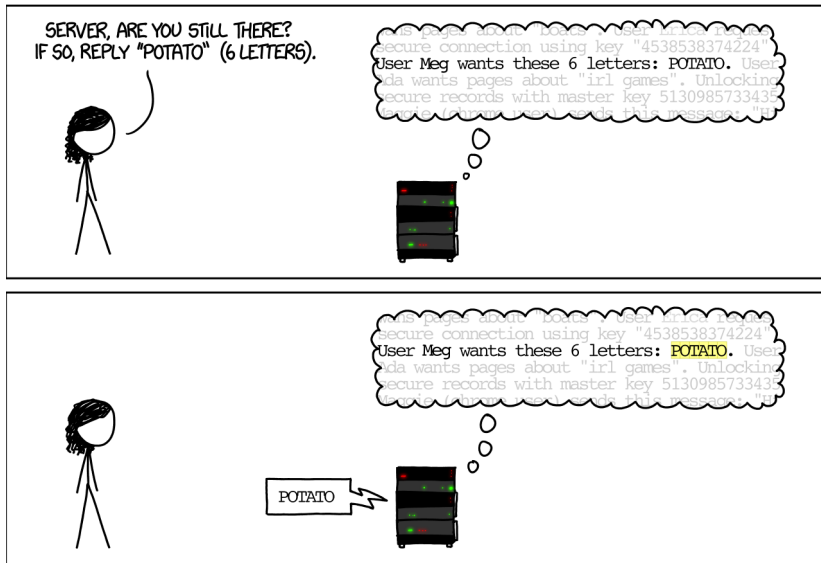
Sortie :

noms[1]: Bob
noms[1]+4: secret

Mémoire :

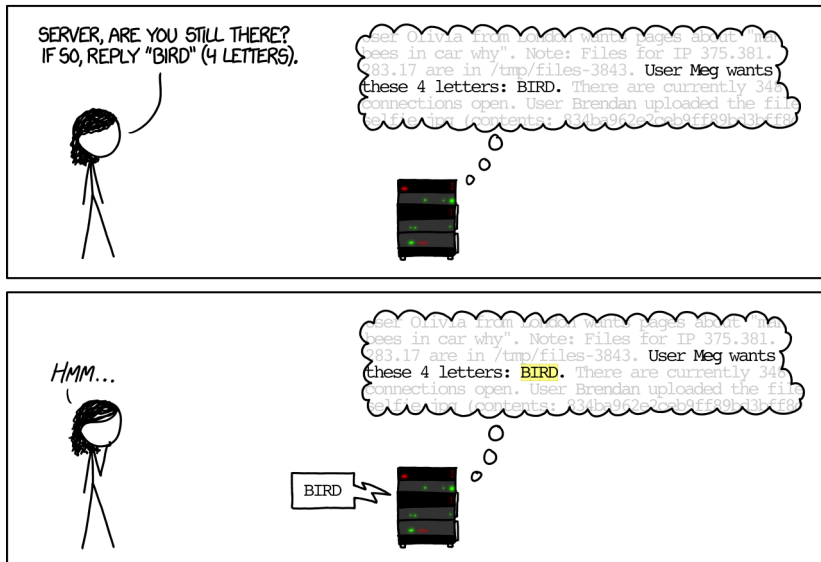
24	?	?	\0	l	e	i	t	n
16	e	d	i	f	n	o	c	\0
8	t	e	r	c	e	s	\0	b
0	o	B	\0	e	c	i	l	A

Heartbleed (500k serveurs affectés en 2014)



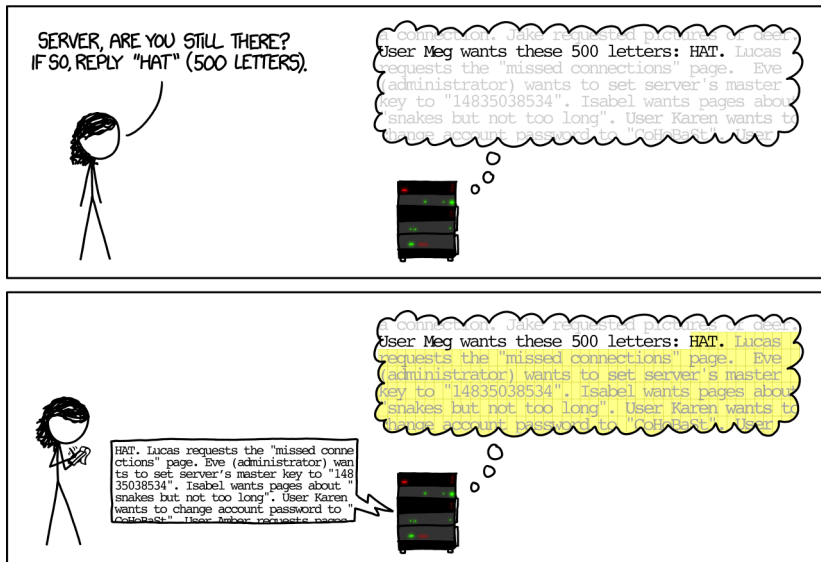
xkcd.com/1354/

Heartbleed (500k serveurs affectés en 2014)



xxcd.com/1354/

Heartbleed (500k serveurs affectés en 2014)



xkcd.com/1354/

Tableaux à plusieurs dimensions

- Une image est une partie de plan : coordonnées x et y .
- Espace à 2 dimensions \Rightarrow structure 2 D \Rightarrow tableau à 2 dimensions \simeq matrice.
- *type nom [taille₁] [taille₂] ;*
- Accès par indexation sur les 2 dimensions : `t[x][y] = ... ;`
- Généralisable à n dimensions : `int t[3][2][6][7][4] ;`
- Tailles de tableaux statiques : connues à la compilation.
- Tableaux dynamiques : subtilités d'initialisation, on verra plus tard.

Tableaux statiques à plusieurs dimensions

st_array.c

```
#include <stdio.h>

#define X (4)
#define Y (3)

int main ()
{
    int x, y ;
    int c[X][Y] ; /* Statique/ */

    for (x = 0; x < X; x++) {
        for (y = 0; y < Y; y++)
            c[x][y] = x + y ;
    }

    for (x = 0; x < X; x++) {
        for (y = 0; y < Y; y++)
            printf ("%d ", c[x][y]) ;
        printf ("\n") ;
    }
    return (0) ;
}
```

```
mymac:/tmp$ gcc st_array.c -o st_array
```

```
mymac:/tmp$ ./st_array
```

```
c:
```

```
0 1 2
```

```
1 2 3
```

```
2 3 4
```

```
3 4 5
```

Un tableau bien utile : argv du main

- Rappel du prototype : `int main (int argc, char *argv[])`
- argv : Tableau contenant les chaînes de caractères passées sur la ligne de commande lors du lancement du programme.
- Permet de passer des entrées au programme au moment du lancement.
- Contient toujours au moins 1 chaîne : `argv[0]` = nom du programme.

args.c

```
#include <stdio.h>

int main (int argc, char *argv[])
{
    int i ;
    for (i = 0; i < argc; i++)
        printf ("i=%d  %s\n", i, argv[i]) ;
    return (0) ;
}
```

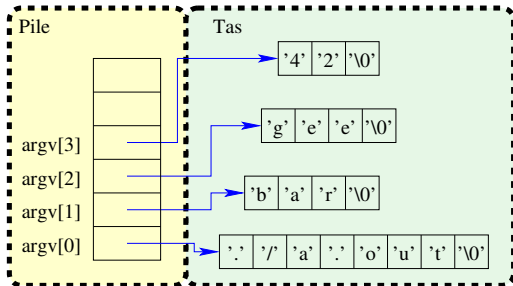
```
mymac:/tmp$ dmd args.d
mymac:/tmp$ ./args bar gee 42
i=0  ./args
i=1  bar
i=2  gee
i=3  42
```

Quelques remarques sur argv (1)

- Déterminer le nombre d'éléments pour savoir jusqu'où aller dans argv.
- → Utilisation de la valeur de argc.

```
mymac:/tmp$ ./args bar gee 42
```

Mémoire



Quelques remarques sur argv (2)

- argv « *contient* » **uniquement** des chaînes de caractères.
- On peut parfois recevoir des nombres en arguments.
- \Rightarrow Nécessité de conversion chaîne \rightarrow nombre.
- Nécessite une transformation algorithmique, et non un changement « simple » de regarder la zone mémoire !
- Fonctions à disposition (requièrent `#include <stdlib.h>`) :
 - ato**i** : string \rightarrow **int**
 - ato**l** : string \rightarrow **long** int
 - ato**ll** : string \rightarrow **long long** int
 - ato**f** : string \rightarrow **float**

Quelques remarques sur argv (3)

string-to-nums.c

```
#include <stdio.h>
#include <stdlib.h>  /* Pour accéder à atoXXX */
int main (int argc, char *argv[])
{
    int i = atoi (argv[1]) ;
    long l = atol (argv[2]) ;
    long long ll = atoll (argv[3]) ;
    float f = atof (argv[4]) ;
    printf ("i: %d, l: %ld, ll: %lld, f: %f\n", i, l, ll, f) ;
    return (0) ;
}
```

```
mymac:/tmp$ gcc -Wall string-to-nums.c
mymac:/tmp$ ./a.out 1 2 4559924234322424 4.5
i: 1, l: 2, ll: 4559924234322424, f: 4.500000
```

Structures de données élémentaires

- Vus jusqu'à présent :
 - Scalaires (entiers, flottants, caractères),
 - Chaînes de caractères,
 - Tableaux.
- Comment modéliser une localisation GPS :
48°42'39.1"N 2°13'09.4"E ?
 - Orientation de latitude (Nord ou Sud),
 - degrés et minutes de latitude (entiers),
 - secondes et fractions de latitude (flottant),
 - orientation de longitude (Est ou Ouest),
 - etc comme pour la latitude...
- ⇒ Besoin de 2 nouveaux types de données.

Types énumérés (1)

- Une orientation de latitude c'est « *Nord* » ou « *Sud* »
... et c'est tout !
- Besoin de valeurs « *atomiques* » disjointes (somme disjointe).
- Encoder par des entiers ?
 - ▶ `int Nord = 30, Sud = 17 ;`
ou bien `#define NORD (30) ... #define SUD (17).`
- La valeur entière importe peu \Rightarrow pourquoi devoir la spécifier ?
- Spécifier les valeurs : source d'erreur si on ré-attribue la même valeur :
 - ▶ `int Nord = 1, Sud = 01`

Types énumérés (2)

- Déclaration : `enum nom-type { val1 , val2 ... };`
- Déclaration d'une variable : `enum nom-type nom-variable ;`
- Utilisation des valeurs : leur `nom`. I.e : `val1, val2 ...`

```
enum lat_orient_t { La_North, La_South } ;
enum long_orient_t { Lo_West, Lo_East } ;

enum long_orient_t invert (enum long_orient_t o)
{
    enum lat_orient_t res ;
    switch (o) {
        case Lo_West: res = Lo_East ; break ;
        case Lo_East: res = Lo_West ; break ;
        default: printf("Erreur") ; break ;        // Ne devrait jamais
                                                    se produire.
    }
    return (res) ;
}
```

Les structures (1)

- Besoin d'agréger des données de **types différents**.
 - Tableaux? → **NON** car agrègent uniquement des données de même type.
- ⇒ Structure : groupement de données par **champs nommés**.
- Un peu comme les classes en Python, mais sans méthodes/fonctions.

```
enum lat_orient_t { La_North, La_South } ;

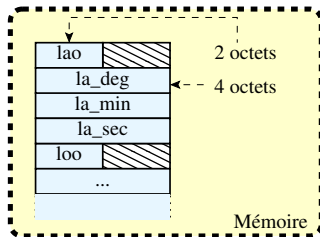
struct gps_loc_t {
    enum lat_orient_t lao ;
    unsigned int la_deg, la_min ;
    float la_sec ;
    enum long_orient_t loo ;
    unsigned int lo_deg, lo_min ;
    float lo_sec ;
};
```

Les structures (2)

- Déclaration de variable : **struct** nom-type nom-variable ;
 - ▶ `struct gps_loc_t somewhere ;`
- Initialisation **à la déclaration** : énumération **dans l'ordre** entre accolades et séparées par des virgules :
 - ▶ `struct gps_loc_t maison =
 { La_North, 48, 42, 39.1, Lo_East, 2, 13, 9.4 } ;`
- Accès à une donnée par nom de champ :
 - ▶ Notation **pointée** si : `struct gps_loc_t maison :`
`printf ("%d, %d\n", maison.la_deg, maison.la_min);`
 - ▶ Notation **fléchée** si : `struct gps_loc_t *maison :`
`printf ("%d, %d\n", maison->la_deg, maison->la_min);`
- Nommage des champs :
 - ▶ ⇒ Facilité d'accès aux différentes parties de la structure.
 - ▶ ⇒ Indépendance par rapport à l'**organisation en mémoire**.

Les structures (3)

- Champs stockés de manière (presque) contiguë en mémoire.
- Attention : il peut y avoir des « *trous* » entre les champs
 - ▶ Contraintes d'architecture matérielle, de performance, etc.
- ⇒ Ne pas s'appuyer sur l'agencement effectif mémoire.



- Les structures sont gérées **comme les autres types** :
 - ▶ Peuvent être des **variables locales** (détruites en fin de portée).
 - ▶ **Recopiées** lorsque passées en **arguments** de fonction.
 - ▶ ⇒ Pour de grosses structures, coût de recopie et coût de pile !
 - ▶ Les passer comme argument **par adresse** (cours ultérieur).

- Besoin de rendre le logiciel **robuste**
 - ▶ Aux changements de structure / implémentation interne.
 - ▶ Aux modifications sauvages mettant en péril des invariants.
- Différentes formes de structures de données vues. . .
 - ▶ Utiliser la plus adaptée au modèle à implémenter !
 - ▶ Ex : struct à 4 champs nord, sud, est, ouest versus tableau de taille fixe = 4.
- **Modularité** : répartir dans des fichiers différents les traitements sur des concepts différents.
 - ▶ Projet réaliste = plusieurs $10(0)^{\text{aines}}$ de milliers de lignes
 - ▶ \Rightarrow Tout n'est pas dans le même fichier.
 - ▶ \Rightarrow Répartition en fonction des sous-problèmes.
- **Abstraction** : ne montrer « à l'extérieur » que le minimum.
 - ▶ \Rightarrow Ne mettre dans les .h que ce qui est nécessaire.