Introduction à la programmation en C

Cours 2
09/01/2013

Compilation de plusieurs fichiers,
Gestion de la mémoire.

Samy BLUSSEAU, Miguel COLOM

I. Compilation de plusieurs fichiers

Souvenons-nous des fichiers leap.c et diviseurs.c, écrits pendant cours précédent (ou...ouvrons-les à nouveau).

Ces fichiers ont la structure suivante :

- Des fonctions sont écrites
- La fonction *main* fait appel à ces fonctions.

Chacun de ces fichiers a permis de construire un programme (un fichier exécutable) :

gcc leap.c -o leap

Que faire si on veut utiliser dans un nouveau programme une des fonctions déjà implémentées (par exemple is_leap_year) ?

On copie/colle ?

Et si cette fonction est recopiée dans un grand nombre de fichiers, et qu'on veut la modifier (pour l'améliorer, corriger un bug)...on perd un temps fou et ses nerfs !

En général, les fonctions d'un programme sont écrites dans un ou plusieurs fichiers séparés du fichier contenant le *main* .

Ainsi on peut regrouper des fonctions dans des fichiers selon les critères de notre choix, et les utiliser dans plusieurs programmes sans les copier.

Les programmes sont alors plus lisibles, plus faciles à modifier, et surtout à corriger en cas de bug !

Alors...séparons !

Dans un nouveau répertoire, copier le fichier leap.c, et créer un nouveau fichier « calendrierlib.c ».

Couper la fonction is_leap_year du nouveau leap.c, et la coller dans calendrierlib.c.

Notre code source s'étend maintenant sur deux fichiers.

Pour construire un unique programme, il va falloir :

1) Déclarer les fonctions

2) Analyser chaque fichier « .c » pour créer des fichiers objets « .o »

3) Générer un exécutable à partir des fichiers objets.

1) Déclarer les fonctions

Une fonction n'est pas connue en dehors de son fichier. Ici, la fonction main() de leap.c ne connait pas la fonction is_leap_year.

Il faut donc « déclarer » is_leap_year au début du fichier leap.c, juste en dessous des #include :

int is_leap_year(int year) ;

Sans accolade, mais avec un « ; » !

1) Déclarer les fonctions

En fait, il faut aussi déclarer une fonction au début du fichier dans laquelle elle se trouve !

Une fonction foo1, placée avant foo2 dans un fichier, ne voit pas foo2.
Pourtant, on peut vouloir appeler foo2 dans foo1 !

Donc de même, au début calendrierlib.c :

int is_leap_year(int year) ;

Toujours sans accolade, et toujours avec un « ; » !

leap.c

```
int is_leap_year(int year) ;


int main(int argc, char** argv) {

…
int ans = is_leap_year( year) ;

...
}
```

calendrierlib.c

```
int is_leap_year(int year) ;


int is_leap_year(int year) {

                ...

}
```

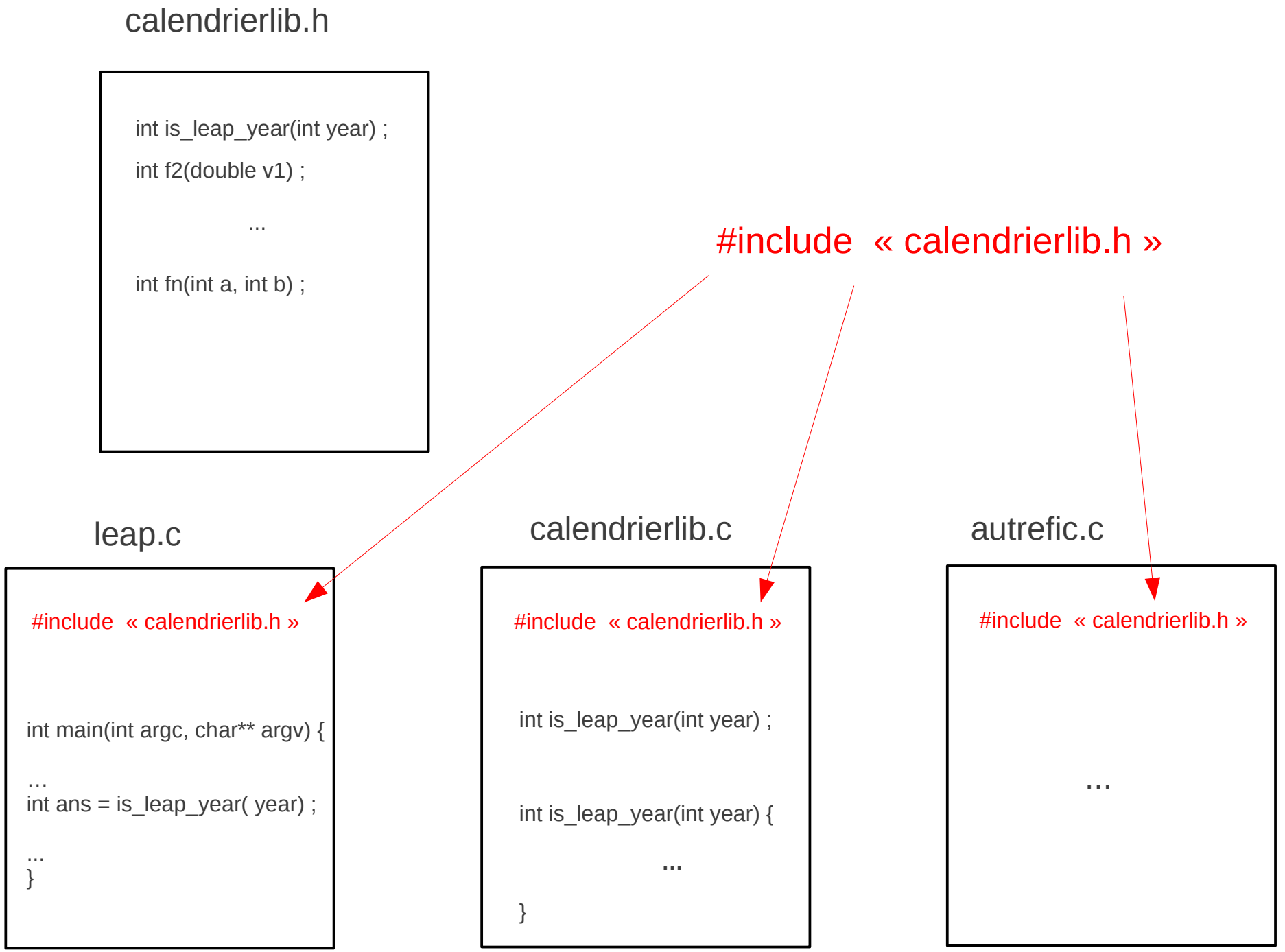
Et si :

- calendrierlib.c contient **n** fonctions
- ces **n** fonctions sont appelées dans le *main* de leap.c
- ces **n** fonctions sont appelées dans un autre fichier « autrefic.c »

...

On déclare 3 fois les **n** fonctions ?!?

Non ! On utilise des fichiers « en-tête », ou *headers* en anglais


calendrierlib.h
int is_leap_year(int year) ;
int f2(double v1) ;
...
int fn(int a, int b) ;
#include « calendrierlib.h »
leap.c
#include « calendrierlib.h »
int main(int argc, char** argv) {
…
int ans = is_leap_year(year) ;
...
}
calendrierlib.c
#include « calendrierlib.h »
int is_leap_year(int year) ;
int is_leap_year(int year) {
...
}
autrefic.c
#include « calendrierlib.h »
…

1) Déclarer les fonctions...dans une en-tête !

- Créer un fichier calendrierlib.h, pour y inclure la déclaration de is_leap_year.
- Remplacer les déclarations précédentes dans leap.c et calendrierlib.c, par #include « calendrierlib.h »

On peut passer à la compilation !

2) Analyse de chaque fichier « .c » pour créer des fichiers objets « .o »

gcc -c leap.c

gcc -c calendrierlib.c

3) Générer un exécutable à partir des fichiers objets.

gcc leap.o calendrierlib.o -o leap

Automatiser la compilation de plusieurs fichiers : le Makefile

Un fichier Makefile contient les instructions indiquant :

1) Comment générer le code objet intermédiaire de chacun des fichiers

2) Comment combiner les fichiers objets pour créer un unique fichier exécutable

Le Makefile présente plusieurs avantages :

- Il permet de **n'écrire qu'une fois les diverses instructions de compilation**, il suffit ensuite d 'exécuter une commande (e général « make » pour compiler tous les fichiers et obtenir l'exécutable)

- Il est capable de ne compiler que les fichiers qui en ont besoin (ceux qui ont été modifiés depuis la dernière compilation), ce qui accélère la compilation .

Un Makefile s'articule en trois sections :

- 1) Une section de définition de variables (ou macros) qui seront utilisées comme « abréviations » dans les autres sections

- 2) Une section où l'on définit le nom des fichiers « objectifs » finaux, c'est à dire ceux qu'on veut récupérer à la fin de l'exécution du Makefile (typiquement, le nom de l'exécutable) .

- 3) Une section où l'on précise comment construire les autres fichiers objectifs (typiquement, les fichiers objets).

Exemple :
Télécharger les fichiers hello.c, volume.c, volume.h, ainsi que le fichier Makefile, et les ouvrir.

Un exemple complet de Makefile :

```
CSTRICT   = -Wall -Wextra -ansi
EXEC      = program

default: $(EXEC)
all: $(EXEC)

# ------- C files -------
hello.o: hello.c
	$(CC) $(CSTRICT) -c hello.c

volume.o: volume.c volume.h
	$(CC) $(CSTRICT) -c volume.c

# ------- Main -------
$(EXEC):  hello.o volume.o
	$(CC)  hello.o volume.o -o $(EXEC)

clean:
	rm -f *.o

distclean: clean
	rm -f $(EXEC)
```

II. Gestion de la mémoire

La mémoire RAM (**R**andom **A**ccess **M**emory)

Programmer en C implique la gestion active de la mémoire,
et donc la connaissance de son fonctionnement.

Contrairement à d'autres langages (Matlab, Java, Python...) qui contrôlent automatiquement les références en mémoire (« garbage collector »), en C c'est le programmeur qui s'en charge.

Cela demande un petit effort, récompensé par une vraie compréhension de ce que fait la machine, et de meilleures performances.

La mémoire est un espace de stockage d'information.

En informatique, l'unité d'information est le bit (« binary digit » : 0 ou 1).
La mémoire est découpée en cases de 8 bits, appelées octets (Bytes en anglais...attention à la confusion entre bit et Byte !)
1 Byte = 1 Octet = 8 bits.

Pour se repérer dans cet espace, on associe à chaque case un numéro, qui est son adresse : on parle d'adresse mémoire.

Ainsi, la mémoire est un grand tableau dont chaque case contient 8 bits d'information.
Les instructions, données, variables d'un programme sont lues et écrites dans ces cases.

Exemple : En C, un nombre décimal est codé sur 32 ou 64 bits (standard IEEE 754-2008). Il est donc écrit sur 4 ou 8 cases en mémoire.

Disque dur et mémoire RAM

Le disque dur est un autre moyen de stockage de l'information.

Le microprocesseur peut accéder au disque (ex : lecture/écriture d'un fichier), mais **cet accès est beaucoup plus lent** que les échanges avec la RAM.
En effet avec la RAM, qui est une puce électronique, on atteint des débits de 18 000 Mo/s (DDR3-2200), contre au plus 600 Mo/s avec un disque dur SATA 3.
De plus un disque dur, qui est en général un dispositif mécanique, doit tourner jusqu'à ce que la tête de lecture soit placée sur la donnée voulue, alors que le temps employé par la RAM pour accéder à une donnée, est constant (~9 ns).

De même, la communication du CPU avec les périphériques (carte graphique, clavier, …) est plus lente qu'avec la RAM.

C'est pour cela qu'**on privilégie l'utilisation de** cette dernière, et que le code machine d'un programme est copié en mémoire (**RAM**) en vue de son exécution par le processeur.

Gérer la mémoire en C : pointeurs, variables

Variables :
Ce sont, en quelques sorte, des représentations de la mémoire dans le langage haut niveau.
Lorsque on veut décrire, en langage C, une procédure appliquée à des données, on stocke ces données dans des variables.

Ex :

```
int a;        // Déclaration de la variable
a = 3;        // affectation
```

En écrivant « a = 3; » on stocke l'entier 3 dans la variable « a » .
Comme cette valeur va être enregistrée en mémoire, on précise auparavant le type de la variable, ici un int (pour « integer », entier).
Cette précision permet :

1) De **réserver la place mémoire** nécessaire (en C un entier est codé sur 32 ou 64 octets, selon le processeur)
→ On répond à la question : « **Combien** d'octets ? »

2) De **lire et écrire** la valeur **dans le bon format**
→ « **Comment** les octets sont utilisés pour représenter la valeur à stocker ? »

Variables :

D'autres types en C :

- float
- double
- char

pour les principaux, mais il y en a d'autres.

Pointeurs :

Du **côté** du **langage** haut niveau, une variable est identifiée par **un nom**, que nous choisissons.

Côté machine, c'est l' **adresse mémoire** qui identifie une donnée.
En C, **un pointeur est une variable qui contient une adresse mémoire**.

Ainsi, l'adresse d'une variable de type int peut être stockée dans un « pointeur d'entier », de type int*, et plus généralement on a un type de pointeur pour chaque type : float* , double*, char* …

On peut faire pointer un pointeur vers une adresse « impossible », « vide », grâce à la constante « NULL » :

```
float* p2 = NULL ;
```

Opérateur de référence

On peut récupérer l'adresse mémoire d'une variable grâce à l'opérateur « & » :

```
int a = 3 ;
int* p = &a ;
```

- a est un entier et vaut 3.
- p est un pointeur d'entier : il peut contenir l'adresse d'une variable de type int.
- Il reçoit l'adresse de l'entier a.

L'opérateur **& est l'opérateur de référence**.

Il **s'applique aux variables** et **renvoie un pointeur**.

On dit qu'**on a référencé la variable a**.

Opérateur de déréférence

On peut aussi récupérer la valeur stockée à l'adresse pointée par un pointeur,
grâce à l'opérateur « * » :

```
int a = 3 ;
int* p = &a ;
int b = *p ;
```

La 3e ligne est ici équivalente à « int b = a ; » car p pointe sur a, et **b reçoit le contenu de la case d'adresse p**.

L'opérateur * **est l'opérateur de déréférence** (ou d'indirection).

Il **s'applique aux pointeurs** et **renvoie la valeur de la variable pointée**.

On dit qu'**on a déréférencé le pointeur p**.

En revanche **on ne peut pas faire** :

```
int a = 3 ;
int b = 5 ;
int* p = &b ;

/* Jusque ici tout va bien... */

&a = p ; // NON !!!
```

On ne change pas l'adresse d'une variable « à la main » !

Tout ce qu'on peut faire, c'est demander (**allouer**) **de la mémoire**, et **accéder** aux cases mémoire que la machine nous a attribuées, grâce aux adresses.

Variable (int)	Adresse	Contenu
bananes	341923	5
poires	341927	12
pommes	341931	8
cerises	341935	79

- **bananes** vaut **5**
- **&banane** vaut **341923**

Si on fait :

```
int* ptr = &bananes ;
```

alors **ptr** vaut **341923**.

Si on ajoute :

```
*ptr = 35 ;
```

alors *(341923) vaut 35 , donc bananes vaut désormais 35 !

Exercice :

```
int bananes = 5 ;
int cerises = 79 ;
int* p = &bananes ;
int** q = &p ;
```

Variable	Adresse	Contenu
bananes	&bananes	5
cerises	&cerises	79
p	&p	&bananes
q	&q	&p

Sans rien programmer, déterminer ce que valent les 4 variables après exécution des deux lignes suivantes :

```
*q = &cerises ;
*p = 3 ;
```

Exercice :

- Créer un fichier pointeurs.c

- Directement dans le *main*
  - déclarer et initialiser un entier a avec une valeur au choix.
  - déclarer un pointeur d'entier et l'initialiser avec l'adresse de a
  - afficher la valeur de a et son adresse
  (le format pour afficher un pointeur avec printf est «%p »)

- Écrire (toujours dans le *main*) le code correspondant à l'exercice précédent, afficher les 4 variables d'intérêt et comparer avec ce qui avait été prédit.

Les tableaux.

Point de vue « **haut niveau** » :
Un tableau est une **famille finie de variables de même type**.
Chaque variable est identifiée par son indice dans le tableau.
En C, les indices commencent toujours à 0.

Exemple : créons un tableau de cinq « double » :

```
double t[5] ;
```

On vient de déclarer un tableau t de longueur 5, dont chaque variable est un double.
On peut initialiser chaque variable comme suit (par exemple) :

```
t[0] = 123 ;
t[1] = 3.1415 ;
t[2] = 0.07 ;
t[3] = 91201.3 ;
t[4] = 27.1087 ;
```

...ou avec une boucle « for »...mais SURTOUT, on ne doit pas chercher à accéder à t[5] qui n'est PAS dans le tableau!

Dans l'exemple précédent

double t[5] ;

on connaît le type de t[i] pour 0 <= i <= 4 : c'est un double.

Mais **quel est le type de t** ?

C'est un **pointeur de double**, c'est à dire un double* .

On commence à voir le lien entre tableaux et mémoire...mais passons à un exemple !

Exemple :

- Créer un fichier tableaux.c. Dans le main(), déclarer puis initialiser un tableau de taille, type, et valeurs de votre choix.
- Afficher à l'écran la valeur et l'adresse de chaque indice.
- Que remarque-t-on dans la suite des adresses mémoire?

Exemple (suite) :

En pointant sur l'adresse d'une variable, grâce à un pointeur p, il est possible d'obtenir l'**adresse de la position suivante** en mémoire, en faisant « **p+1** ».
Un pointeur étant associé à un type, la machine sait combien d'octets il faut ajouter pour aller à la position suivante.

Reprendre tableaux.c et afficher, en plus de l'adresse de chaque variable du tableau, l'adresse suivante correspondante.

Expliquer alors ce qui a été observé précédemment.

Point de vue machine :

Un tableau est un ensemble de cases mémoires **contiguës** !

Et la variable **t**, qui est un pointeur, **n'est autre que l'adresse de la première de ces cases**.

Récapitulons :

Quand on écrit **double t[5] ;** , on alloue une quantité suffisante de mémoire contiguë pour recevoir cinq variables de type double. Le début de cette plage mémoire se situe à l'adresse contenue dans t (qui est aussi l'adresse de t[0]) !

Et si on a besoin d'un tableau de taille n, où n est un entier donné par l'utilisateur du programme, donc non connu à l'avance... ?

```
int n ;

// on récupère l'entier fourni par l'utilisateur...
printf(« Entrer la taille du tableau (entier >0).\n ») ;
scanf(« %d », &n) ;

// … et après ?

double t[n] ; // NON !
```

Allocation dynamique de la mémoire

Dans ce cas (très courant !) on doit allouer **dynamiquement** la mémoire, avec la syntaxe suivante :

double* t = (double*) malloc(n*sizeof(double)) ;

- La fonction malloc (pour « memory allocation ») alloue des octets de mémoire.
- L'opérateur « sizeof() » renvoie le nombre d'octets nécessaires pour stocker une variable du type passé en argument (ici : double).
- Avec (double*) on précise que les octets alloués vont servir à coder des doubles (on précise le format).

Lorsqu'on utilise malloc, la mémoire reste allouée jusqu'à ce que le programmeur la libère explicitement en faisant :

free(t) ;

Fuites mémoire

Si un programme alloue de la mémoire avec malloc, sans la libérer avec free,
il se produit une fuite mémoire (*memory leak*) :

la quantité de mémoire disponible diminue, et peut devenir insuffisante...

D'où la règle d'or :

AUTANT DE « FREE » QUE DE « MALLOC » DANS UN PROGRAMME !

Exercice :

Reprendre tableaux.c :

- Demander à l'utilisateur d'entrer un entier n > 2
- Créer un tableau contenant les n premiers termes de la suite de Fibonacci.
- Afficher le contenu du tableau
- Libérer la mémoire allouée !

Les fonctions (et la mémoire !)

Une fonction en C est identifiée par

- Un nom
- Des arguments (ou paramètres) : les variables reçues par la fonction
  Aucun, un ou plusieurs arguments
- Des instructions comprises entre deux accolades
- Une seule variable de retour (éventuellement vide)

Exemples :

```
int main(int argc, char** argv) {

                    …

    return 0 ;
}
```

```
float somme(float a, float b) {
    return a+b ;
}
```

```
void affiche_tableau_entiers(int* t, int taille) {

  int i;
  for( i = 0 ; i < taille ; i++ )
    printf( "t[%d] = %d\n", i, t[i] );

}
```

Les variables déclarées dans une fonction sont dites **variables locales**.
(par opposition aux **variables globales**, déclarées en dehors de toute fonction, et à éviter !)

Elles n'existent que pendant l'exécution de la fonction, et ne sont donc plus en mémoire au sortir de la fonction :

- Soit parce que la variable est effacée automatiquement
  (dans le cas d'une allocation statique, sans malloc).

- Soit parce que le programmeur a pris soin de restituer dans la fonction, par des « free » la mémoire allouée par des malloc dans cette même fonction.

De façon générale, la portée d'une variable locale est limitée par les deux accolades les plus proches entre lesquelles la variable est déclarée.

Les paramètres d'une fonction sont également des variables locales !

Ainsi, il est impossible pour une fonction de modifier l'un de ses paramètres, puisque un telle modification disparaît au sortir de la fonction.

Exemple :
Soit la fonction :

```
void triple(int a){
        a = 3*a ;
}
```

Alors, si on fait :

```
int b = 8 ;
triple(b) ;
```

b vaut toujours 8 à la sortie de « triple » !
On dit qu'on a passé b **par valeur** à la fonction.

Si l'on veut modifier une variable en appelant une fonction,
il faut passer l'adresse (ou référence) de la variable.

On dit que l'on passe le paramètre par référence

Exemple :

```
void triple(int* a){
      *a = 3* (*a) ;
}
```

Alors, si on passe b à triple par référence:

```
int b = 8 ;
triple( &b ) ;
```

Maintenant, b vaut 24 !

Remarque : Un tableau étant déjà une adresse, il est toujours passé par référence, et donc modifiable par une fonction.

Exercice de synthèse :

1) Dans un fichier tri.c, écrire au moins deux fonctions qui implémentent chacune un algorithme de tri de votre choix.

Chacune d'elle :

- reçoit en paramètre un tableau d'entiers (un int*) à trier et la taille du tableau.

- ne renvoie rien (void)

- le tableau passé en paramètre doit être trié après exécution de la fonction.

2) Créer un header tri.h dans lequel seront déclarées les fonctions implémentées dans tri.c

3) Créer un fichier test_tri.c contenant le main du programme, et incluant tri.h.
Dans le main, demander à l'utilisateur de choisir une longueur n de tableau, puis d'entrer successivement les n valeurs à trier.
Appeler les fonctions de tri implémentées pour trier le tableau, et afficher le résultat.

4) Créer un Makefile permettant de compiler ce programme.