UNIVERSITÉ MOHAMED PREMIER - OUJDA

Faculté Des Sciences

Département De Mathématiques Et Informatique

Cours du module

Programmation en langage C

SMI S4

Prof. JAARA

Année Universitaire 2015/2016

Merci de nous rendre visite sur
http://fso.umpoujda.com/

Chapitre 1 : La gestion dynamique de la mémoire

En langage C un programme comporte trois types de données :

- Statiques;
- Automatiques ;
- Dynamiques.

✓ Les **données statiques** occupent un emplacement parfaitement défini lors de la compilation.

✓ Les **données automatiques**, en revanche, n'ont pas une taille définie a priori. En effet, elles ne sont créées et détruites qu'au fur et à mesure de l'exécution du programme. Elles sont souvent gérées sous forme de ce que l'on nomme une **pile** (*stack* en anglais).

✓ Les **données dynamiques** n'ont pas non plus de taille définie a priori. Leur création ou leur libération dépend, cette fois, de demandes explicites faites lors de l'exécution du programme. Leur gestion, qui ne saurait se faire à la manière d'une pile, est indépendante de celle des données automatiques. Plus précisément, elle se fait généralement dans ce que l'on nomme un **tas** (*heap* en anglais) dans lequel on cherche à allouer ou à libérer de l'espace en fonction des besoins.

1 Notion de pointeur – Les opérateurs * et &

1.1 Introduction

Nous avons déjà été amené à utiliser l'opérateur **`&`** pour désigner l'adresse d'une ***lvalue***. D'une manière générale, le langage C permet de manipuler des adresses par l'intermédiaire de variables nommées pointeurs.
En guise d'introduction à cette nouvelle notion, considérons les instructions :

```
int * ad ;
int n ;
n = 20 ;
ad = &n ;
*ad = 30 ;
```

La première réserve une variable nommée **`ad`** comme étant un pointeur sur des entiers. Nous verrons que **`*`** est un opérateur qui désigne le contenu de l'adresse qui le suit. Ainsi, à titre mnémonique, on peut dire que cette déclaration signifie que **`*ad`**, c'est-à-dire l'objet d'adresse **`ad`**, est de type **`int`** ; ce qui signifie bien que **`ad`** est l'adresse d'un entier. L'instruction : **`ad = &n ;`** affecte à la variable **`ad`** la valeur de l'expression **`&n`**. L'opérateur **`&`** (que nous avons déjà utilisé avec **`scanf`**) est un opérateur unaire qui fournit comme résultat l'adresse de son opérande.
Ainsi, cette instruction place dans la variable **`ad`** l'adresse de la variable **`n`**. Après son exécution, on peut schématiser ainsi la situation :

FSO.UMPOUJDA.COM

L'instruction suivante : **`*ad = 30 ;`** signifie : affecter à la ***lvalue*** **`*ad`** la valeur **`30`**. Or **`*ad`** représente l'entier ayant pour adresse **`ad`** (notez bien que nous disons l'entier et pas simplement la valeur car, ne l'oubliez pas, **`ad`** est un pointeur sur des entiers). Après exécution de cette instruction, la situation est la suivante :

Bien entendu, ici, nous aurions obtenu le même résultat avec : **`n = 30 ;`**

1.2 Quelques exemples

Voici quelques exemples d'utilisation de ces deux opérateurs. Supposez que nous ayons effectué ces déclarations :

```
int *ad1, *ad2, *ad ;
int n = 10, p = 20 ;
```

Les variables **`ad1`**, **`ad2`** et **`ad`** sont donc des pointeurs sur des entiers. Remarquez bien la forme de la déclaration, en particulier, si l'on avait écrit :
`int *ad1, ad2, ad ;` la variable **`ad1`** aurait bien été un pointeur sur un entier (puisque **`*ad1`** est entier) mais **`ad2`** et **`ad`** auraient été, quant à eux, des entiers. Considérons maintenant ces instructions :

```
ad1 = &n ;
ad2 = &p ;
*ad1 = *ad2 + 2 ;
```

Les deux premières placent dans **`ad1`** et **`ad2`** les adresses de **`n`** et **`p`**. La troisième affecte à **`*ad1`** la valeur de l'expression : **`*ad2 + 2`**
Autrement dit, elle place à l'adresse désignée par **`ad1`** la valeur (entière) d'adresse **`ad2`**, augmentée de **`2`**. Cette instruction joue donc ici le même rôle que : **`n = p+2;`**
De manière comparable, l'expression : **`*ad1 +=3`** jouerait le même rôle que : **`n=n+3`** et l'expression : **`(*ad1)++`** jouerait le même rôle que **`n++`** (nous verrons plus loin que, sans les parenthèses, cette expression aurait une signification différente).

1.3 Incrémentation de pointeurs

Jusqu'ici, nous nous sommes contenté de manipuler, non pas les variables pointeurs elles mêmes, mais les valeurs pointées. Or si une variable pointeur **`ad`** a été déclarée ainsi : **`int *ad ;`** une expression telle que : **`ad+1`** a un sens pour C.

FSO.UMPOUJDA.COM

En effet, **ad** est censée contenir l'adresse d'un entier et, pour C, l'expression ci-dessus représente **l'adresse de l'entier suivant**.
Notez bien qu'il ne faut pas confondre un pointeur avec un nombre entier. En effet, l'expression ci-dessus ne représente pas l'adresse de **ad** augmentée de un (octet). Plus précisément, la différence entre **ad+1** et **ad** est ici de **sizeof(int)** octets (n'oubliez pas que l'opérateur **sizeof** fournit la taille, en octets, d'un type donné). Si **ad** avait été déclarée par : **double *ad ;** cette différence serait de **sizeof(double)** octets.
De manière comparable, l'expression : **ad++** incrémente l'adresse contenue dans **ad** de manière qu'elle désigne **l'objet** suivant.
Notez bien que des expressions telles que **ad+1** ou **ad++** sont, en général, valides, quelle que soit l'information se trouvant réellement à l'emplacement correspondant. D'autre part, il est possible d'incrémenter ou de décrémenter un pointeur de n'importe quelle quantité entière.
Par exemple, avec la déclaration précédente de **ad**, nous pourrions écrire ces instructions :

```
ad += 10 ;
ad -= 25 ;
```

2 Comment simuler une transmission par adresse avec un pointeur

Nous avons vu que le mode de transmission par valeur semblait interdire à une fonction de modifier la valeur de ses arguments effectifs et nous avions mentionné que les pointeurs fourniraient une solution à ce problème.
Nous sommes maintenant en mesure d'écrire une fonction effectuant la permutation des valeurs de deux variables. Voici un programme qui réalise cette opération avec des valeurs entières :

```
#include <stdio.h>
main()
{
void echange (int *ad1, int *ad2) ;
int a=10, b=20 ;
printf ("avant appel %d %d\n", a, b) ;
echange (&a, &b) ;
printf ("après appel %d %d", a, b) ;
}
void echange (int *ad1, int *ad2)
{ int x ;
x = *ad1 ;
*ad1 = *ad2 ;
*ad2 = x ;
}
```

Utilisation de pointeurs en argument d'une fonction

```
avant appel 10 20
après appel 20 10
```


FSO UMPOUJDA.COM

Les arguments effectifs de l'appel de **echange** sont, cette fois, les adresses des variables **n** et **p** (et non plus leurs valeurs). Notez bien que la transmission se fait toujours par valeur, à savoir que l'on transmet à la fonction **echange** les valeurs des expressions **&n** et **&p**.
Voyez comme, dans **echange**, nous avons indiqué, comme arguments muets, deux variables pointeurs destinées à recevoir ces adresses. D'autre part, remarquez bien qu'il n'aurait pas fallu se contenter d'échanger simplement les valeurs de ces arguments en écrivant :

```
int * x ;
x = ad1 ;
ad1 = ad2 ;
ad2 = x ;
```

Cela n'aurait conduit qu'à échanger (localement) les valeurs de ces deux adresses alors qu'il a fallu échanger les valeurs situées à ces adresses.

3 Un nom de tableau est un pointeur constant

En langage C, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau. Nous allons en examiner les conséquences en commençant par le cas des tableaux à un indice ; nous verrons en effet que, pour les tableaux à plusieurs indices, il faudra tenir compte du type exact du pointeur en question.

3.1 Cas des tableaux à un indice

Supposons, par exemple, que l'on effectue la déclaration suivante : **int t[10] ;**
La notation **t** est alors totalement équivalente à **&t[0].**
L'identificateur **t** est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau, c'est-à-dire, ici, **int***. Ainsi, voici quelques exemples de notations équivalentes :

t+1	&t[1]
t+i	&t[i]
t[i]	*(t+i)

Pour illustrer ces nouvelles possibilités de notation, voici plusieurs façons de placer la valeur **1** dans chacun des **10** éléments de notre tableau **t** :

```
int i ;
for (i=0 ; i<10 ; i++)    *(t+i) = 1 ;
int i ;
int* p :
for (p=t, i=0 ; i<10 ; i++, p++) *p = 1 ;
```

Dans la seconde façon, nous avons dû recopier la valeur représentée par `t` dans un pointeur nommé **p**. En effet, il ne faut pas perdre de vue que le symbole **t** représente une adresse constante (**t** est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que **t++** aurait été invalide, au même titre que,

FSO.UMPOUJDA.COM

par exemple, **`3++`. Un nom de tableau est un pointeur constant ; ce n'est pas une *lvalue***.

3.2 Cas des tableaux à plusieurs indices

Comme pour les tableaux à un indice, l'identificateur d'un tableau, employé seul, représente toujours son adresse de début. Toutefois, si l'on s'intéresse à son type exact, il ne s'agit plus d'un pointeur sur des éléments du tableau. En pratique, ce point n'a d'importance que lorsque l'on effectue des calculs arithmétiques avec ce pointeur (ce qui est assez rare) ou lorsque l'on doit transmettre ce pointeur en argument d'une fonction ; dans ce dernier cas, cependant, nous verrons que le problème est automatiquement résolu par la mise en place de conversions, de sorte qu'on peut ne pas s'en préoccuper.

À simple titre indicatif, nous vous présentons ici les règles employées par C, en nous limitant au cas de tableaux à deux indices.
Lorsque le compilateur rencontre une déclaration telle que : **`int t[3][4];`** il considère en fait que **`t`** désigne un tableau de **3** éléments, chacun de ces éléments étant lui-même un tableau de **4** entiers. Autrement dit, si **`t`** représente bien l'adresse de début de notre tableau **`t`**, il n'est plus de type **`int*`** (comme c'était le cas pour un tableau à un indice) mais d'un type « pointeur sur des blocs de **4** entiers », type qui devrait se noter théoriquement (vous n'aurez probablement jamais à utiliser cette notation : **`int [4]*`**

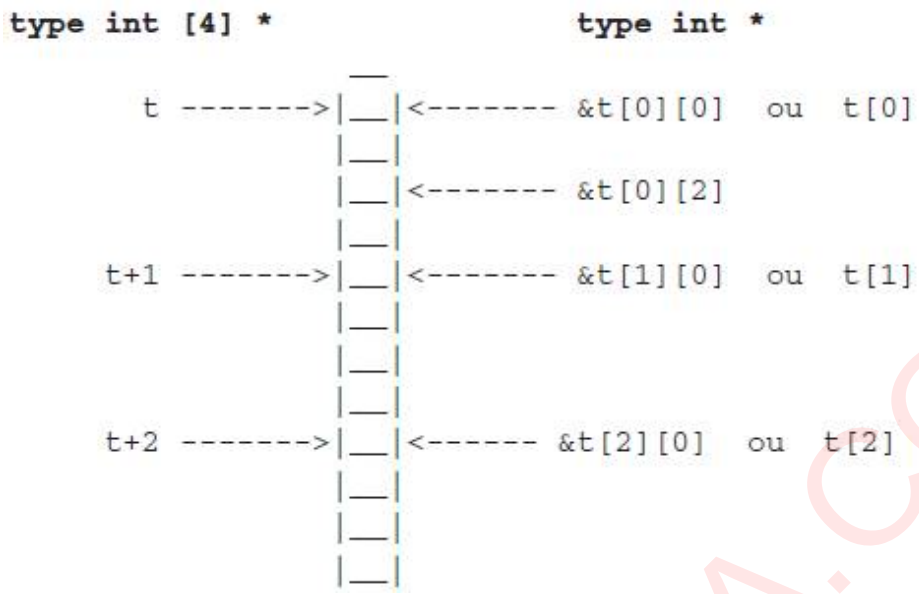
Dans ces conditions, une expression telle que **`t+1`** correspond à l'adresse de **`t`**, augmentée de **4** entiers (et non plus d'un seul !). Ainsi, les notations **`t`** et **`&t[0][0]`** correspondent toujours à la même adresse, mais l'incrémentation de **`1`** n'a pas la même signification pour les deux.

D'autre part, les notations telles que **`t[0], t[1]`** ou **`t[i]`** ont un sens. Par exemple, **`t[0]`** représente l'adresse de début du premier bloc (de **4** entiers) de **`t`**, **`t[1],`** celle du second bloc...

Cette fois, il s'agit bien de pointeurs de type **`int *.`** Autrement, dit les notations suivantes sont totalement équivalentes (elles correspondent à la même adresse et elles sont de même type) :

```
 t[0]      &t[0][0]
t[1]       &t[1][0]
```

Voici un schéma récapitulant ce que nous venons de dire.



Remarque :

`t[1]` est une constante ; ce n'est pas une `lvalue`. L'expression : `t[1]++` est invalide. Par contre, `t[1][2]` est bien une `lvalue`.

4 Les opérateurs réalisables sur des pointeurs

Nous avons déjà vu ce qu'étaient la somme ou la différence d'un pointeur et d'une valeur entière. Nous allons examiner ici les autres opérations réalisables avec des pointeurs.

4.1 La comparaison de pointeurs

Il ne faut pas oublier qu'en C un pointeur est défini à la fois par une adresse en mémoire et par un type. On ne pourra donc **comparer que des pointeurs de même type**. Par exemple, voici, en parallèle, deux suites d'instructions réalisant la même action : mise à **1** des **10** éléments du tableau `t` :

```
int t[10] ;
int * p ;
for (p=t ; p<t+10 ; p++)
*p = 1 ;
```

```
int t[10] ;
int i ;
for (i=0 ; i<10 ; i++)
t[i] = 1 ;
```

4.2 La soustraction de pointeurs

Là encore, quand deux pointeurs sont **de même type**, leur différence fournit le nombre d'éléments du type en question situés entre les deux adresses correspondantes. L'emploi de cette possibilité est assez rare.

FSO.UMPOUJDA.COM

4.3 Les affectations de pointeurs et le pointeur nul

Nous avons naturellement déjà rencontré des cas d'affectation de la valeur d'un pointeur à un pointeur de même type. A priori, c'est le seul cas autorisé par le langage C (du moins, tant que l'on ne procède pas à des conversions explicites). Une exception a toutefois lieu en ce qui concerne la valeur entière **0**, ainsi que pour le type générique **`void *`** dont nous parlerons un peu plus loin. Cette tolérance est motivée par le besoin de pouvoir représenter un pointeur nul, c'est-à-dire ne pointant sur rien. Bien entendu, cela n'a d'intérêt que parce qu'il est possible de comparer n'importe quel pointeur (de n'importe quel type) avec ce « pointeur nul ».
D'une manière générale, plutôt que la valeur **0**, il est conseillé d'employer la constante **`NULL`** prédéfinie dans **`stdio.h`**, et également dans **`stddef.h`** (bien entendu, elle sera remplacée par la constante entière **`0`** lors du traitement par le préprocesseur, mais les programmes source en seront néanmoins plus lisibles). Avec ces déclarations :

```
int * n ;
double * x ;
```

ces instructions seront correctes :

```
n = 0 ;          /* ou mieux */        n = NULL ;
x = 0 ;          /* ou mieux */        x = NULL ;
if (n == 0) ... /* ou mieux */        if (n == NULL) ...
```

4.4 Les conversions de pointeurs

Il n'existe aucune conversion implicite d'un type pointeur dans un autre. En revanche, il est toujours possible de faire appel à l'opérateur de **`cast`**. D'une manière générale, nous vous conseillons de l'éviter, compte tenu des risques qu'elle comporte. En effet, on pourrait penser qu'une telle conversion revient finalement à ne s'intéresser qu'à l'adresse correspondant à un pointeur, sans s'intéresser au type de l'objet pointé.
Malheureusement, il faut tenir compte de ce que certaines machines imposent aux adresses des objets ce que l'on appelle des « contraintes d'alignement ». Par exemple, un objet de 2 octets sera toujours placé à une adresse paire, tandis qu'un caractère (objet d'un seul octet) pourra être placé (heureusement) à n'importe quelle adresse. Dans ce cas, la conversion d'un **`char *`** en un **`int *`** peut conduire soit à l'adresse effective du caractère lorsque celle-ci est paire, soit à une adresse voisine lorsque celle-ci est impaire.

4.5 Les pointeurs génériques

En C, un pointeur correspond à la fois à une adresse en mémoire et à un type. Précisément, ce typage des pointeurs peut s'avérer gênant dans certaines circonstances telles que celles où une fonction doit manipuler les adresses d'objets de type non connu (ou, plutôt, susceptible de varier d'un appel à un autre).
Dans certains cas, on pourra satisfaire un tel besoin en utilisant des pointeurs de type **`char*`**, lesquels, au bout du compte, nous permettront d'accéder à n'importe quel octet de la mémoire. Toutefois, cette façon de procéder implique obligatoirement l'emploi de conversions explicites.
En fait, la norme ANSI a introduit le type pointeur suivant (il n'existait pas dans la définition initiale du langage C, effectuée par Kernighan et Ritchie) : **`void*`** Celui-ci

FSO.UMPOUJDA.COM

désigne un **pointeur sur un objet de type quelconque** (on parle souvent de «Pointeur générique »). Il s'agit (exceptionnellement) d'un pointeur sans type.
Une variable de type **void*** ne peut pas intervenir dans des opérations arithmétiques; notamment, si **p** et **q** sont de type **void***, on ne peut pas parler de **p+i** (**i** étant entier) ou de **p-q** ; on ne peut pas davantage utiliser l'expression **p++ ;** ceci est justifié par le fait qu'on ne connaît pas la taille des objets pointés. Pour des raisons similaires, il n'est pas possible d'appliquer l'opérateur d'indirection ***** à un pointeur de type **void***.
Les pointeurs génériques sont théoriquement compatibles avec tous les autres ; autrement dit, les affectations **type* -> void*** sont légales (ce qui ne pose aucun problème) mais les affectations **void* -> type*** (elles seront d'ailleurs illégales en C++) le sont également, ce qui présente les risques évoqués précédemment à propos des contraintes d'alignement.
On notera bien que, lorsqu'il est nécessaire à une fonction de travailler sur les différents octets d'un emplacement de type quelconque, le type **void*** ne convient pas pour décrire les différents octets de cet emplacement et il faudra quand même recourir, à un moment ou à un autre, au type **char*** (mais les conversions **void* --> char*** ne poseront jamais de problème de contraintes d'alignement). Ainsi, pour écrire une fonction qui « **met à zéro** » un emplacement de la mémoire dont on lui fournit l'adresse et la taille (en octets), on pourrait songer à procéder ainsi :

```
void raz (void *adr, int n)
{
int i ;
for (i=0 ; i<n ; i++, adr++) *adr = 0 ; // illégal
}
```

Manifestement, ceci est illégal et il faudra utiliser une variable de type **char*** pour décrire notre zone :

```
void raz (void *adr, int n)
{
int i ;
char *ad = adr ;
for (i=0 ; i<n ; i++, ad++) *ad = 0 ;
}
```

Voici un exemple d'utilisation de notre fonction **raz** :

```
void raz (void *, int) ; /* prototype réduit */
int t[10] ;              /* tableau à mettre à zéro */
double z ;               /* double à mettre à zéro */
....
raz (t, 10*sizeof(int)) ;
raz (z, sizeof (z)) ;
```

FSO.UMPOUJDA.COM

5 Les outils de base de la gestion dynamique : malloc et free

Commençons par étudier les deux fonctions les plus classiques de gestion dynamique de la mémoire, à savoir ***malloc*** et ***free***.

5.1 La fonction malloc

a) Premier exemple : Considérez ces instructions :

```
#include <stdlib.h>
.....
char * adr ;
.....
adr = malloc(50) ;
.....
for (i=0 ; i<50 ; i++) *(adr+i) = 'x' ;
```

L'appel : **`malloc (50)`** alloue un emplacement de **50** octets dans le tas et en fournit l'adresse en retour. Ici, cette dernière est placée dans le pointeur **`adr`**.
L'instruction **`for`** qui vient à la suite n'est donnée qu'à titre d'exemple d'utilisation de la zone ainsi créée.

b) Second exemple

```
long * adr ;
.....
adr = malloc(100 * sizeof(long)) ;
.....
for (i=0 ; i<100 ; i++) *(adr+i) = 1 ;
```

Cette fois, nous nous sommes alloué une zone de **`100 * sizeof(long)`** octets (notez l'emploi de **`sizeof`** qui assure la portabilité). Mais nous l'avons considérée comme une suite de **100** entiers de type **`long`**. Pour ce faire, vous voyez que nous avons pris soin de placer le résultat de **`malloc`** dans un pointeur sur des éléments de type **`long`**. N'oubliez pas que les règles de l'arithmétique des pointeurs font que : **`adr + i`** correspond à l'adresse contenue dans **`adr`**, augmentée de **`sizeof(long)`** fois la valeur de `i` (puisque **`adr`** pointe sur des objets de longueur **`sizeof(long)`**).

FSO.UMPOUJDA.COM

5.2 La fonction free

L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin. Le rôle de la fonction **free** est de libérer un emplacement préalablement alloué.

Voici un exemple de programme, exécuté ici dans un environnement DOS. Il vous montre comment **malloc** peut profiter d'un espace préalablement libéré sur le tas. Notez que la dernière allocation a pu se faire dans l'espace libéré par le précédent appel de **free**.

```
#include <stdio.h>
#include <stdlib.h>
main()
{
char * adr1, * adr2, * adr3 ;
adr1 = malloc (100) ;
printf ("allocation de 100 octets en %p\n", adr1) ;
adr2 = malloc (50) ;
printf ("allocation de 50 octets en %p\n", adr2) ;
free (adr1) ;
printf ("libération de 100 octets en %p\n", adr1) ;
adr3 = malloc (40) ;
printf ("allocation de 40 octets en %p\n", adr3) ;
}
```

***Exemple d'utilisation de la fonction* free**

```
allocation de 100 octets en 06AC
allocation de 50 octets en 0714
libération de 100 octets en 06AC
allocation de 40 octets en 06E8
```

Notez que la dernière allocation a pu se faire dans l'espace libéré par le précédent appel de **free**.

6 D'autres outils de gestion dynamique : calloc et realloc

Bien qu'elles soient moins fondamentales que les précédentes, les deux fonctions **calloc** et **realloc** peuvent s'avérer pratiques dans certaines circonstances.

FSO.UMPOUJDA.COM

6.1 La fonction calloc

La fonction : **void * calloc (size_t nb_blocs, size_t taille)** ***(stdlib.h)*** alloue l'emplacement nécessaire à **`nb_blocs`** consécutifs, ayant chacun une taille de **`taille`** octets.
Contrairement à ce qui se passait avec **`malloc`**, où le contenu de l'espace mémoire alloué était imprévisible, **`calloc`** remet à zéro chacun des octets de la zone ainsi allouée. La taille de chaque bloc, ainsi que leur nombre sont tous deux de type **`size_t`**. On voit ainsi qu'il est possible d'allouer en une seule fois une place mémoire (de plusieurs blocs) beaucoup plus importante que celle allouée par `malloc` (la taille limite théorique étant maintenant **`size_t*size_t`** au lieu de **`size_t`**).

6.2 La fonction realloc

La fonction : **void * realloc (void * pointeur, size_t taille)** ***(stdlib.h)*** permet de modifier la taille d'une zone préalablement allouée (par **`malloc`**, **`calloc`** ou **`realloc`**).
Le pointeur doit être l'adresse de début de la zone dont on veut modifier la taille. Quant à **`taille`**, de type **`size_t`**, elle représente la nouvelle taille souhaitée. Cette fonction restitue l'adresse de la nouvelle zone ou un pointeur nul dans le cas où l'allocation a échoué.
Lorsque la nouvelle taille demandée est supérieure à l'ancienne, le contenu de l'ancienne zone est conservé (quitte à le recopier si la nouvelle adresse est différente de l'ancienne). Dans le cas où la nouvelle taille est inférieure à l'ancienne, le début de l'ancienne zone (c'est-à-dire `taille` octets) verra son contenu inchangé.

7 Exemple d'application de la gestion dynamique : création d'une liste chaînée

Comme nous l'avons déjà évoqué en introduction de ce chapitre, il n'est pas possible de déclarer un tableau dont le nombre d'éléments n'est pas connu lors de la compilation. En revanche, les possibilités de gestion dynamique du langage C nous permettent d'envisager d'allouer des emplacements aux différents éléments du tableau au fur et à mesure des besoins.
Si l'on n'a pas besoin d'accéder directement à chacun des éléments, on peut se contenter de constituer ce que l'on nomme une « liste chaînée », dans laquelle :

- Un pointeur désigne le premier élément ;
- Chaque élément comporte un pointeur sur l'élément suivant.

Dans ce cas, les emplacements des différents éléments peuvent être alloués de façon dynamique, au fur et à mesure des besoins. Il n'est plus nécessaire de connaître d'avance leur nombre ou une valeur maximale (ce qui serait le cas si l'on créait un tableau de tels éléments).
Appliquons cela à des éléments de type **`point`**, structure comportant les champs suivants :

```
struct point { int num ;
               float x ;
               float y ;
             } ;
```

Chaque élément doit donc contenir un pointeur sur un élément de même type. Il y a là une récursivité des déclarations qui est autorisée en C. Ainsi, nous pourrons adapter notre précédente structure de la manière suivante :

```
struct element { int num ;
                 float x ;
                 float y ;
                 struct element * suivant ;
               } ;
```

Vous voyez que nous avons été amenés à utiliser dans la description du modèle **element** un pointeur sur ce même modèle.
Supposons que nous cherchions à constituer notre liste chaînée à partir d'informations fournies en données. Deux possibilités s'offrent à nous :

- Ajouter chaque nouvel élément à la fin de la liste. Le parcours ultérieur de la liste se fera alors dans le même ordre que celui dans lequel les données ont été introduites.
- Ajouter chaque nouvel élément au début de la liste. Le parcours ultérieur de la liste se fera alors dans l'ordre inverse de celui dans lequel les données ont été introduites.

Nous avons choisi ici de programmer la seconde méthode, laquelle se révèle légèrement plus simple que la deuxième.
Notez que le dernier élément de la liste (donc, dans notre cas, le premier lu) ne pointera sur rien. Or, lorsque nous chercherons ensuite à utiliser notre liste, il nous faudra être en mesure de *savoir où elle s'arrête*. Certes, nous pourrions, à cet effet, conserver l'adresse de son dernier élément. Mais il est plus simple d'attribuer au champ *suivant* de ce dernier élément une valeur fictive dont on sait qu'elle ne peut apparaître par ailleurs. La valeur **NULL (0)** fait très bien l'affaire.
Ici, nous avons décidé de faire effectuer la création de la liste par une fonction. Le programme principal se contente de réserver l'emplacement d'un pointeur destiné à désigner le premier élément de la liste. Sa valeur effective sera fournie par la fonction **creation**. Dans ces conditions, il est nécessaire que le programme principal lui fournisse, non pas la valeur, mais l'adresse de ce pointeur (du moins si l'on souhaite pouvoir disposer ultérieurement de cette valeur au sein du programme principal).
C'est ce qui justifie la forme de l'en-tête de la fonction **creation** :

```
void creation (struct element * * adeb)
```

dans laquelle **adeb** est effectivement du type « pointeur sur un pointeur sur un élément de type **struct element** ».

```
#include <stdio.h>
#include <stdlib.h>
struct element { int num ;
float x ;
float y ;
struct element * suivant ;
} ;
void creation (struct element * * adeb) ;
main()
{
struct element * debut ;
```

```
creation (&debut) ;
}
void creation (struct element * * adeb)
{
int num ;
float x, y ;
struct element * courant ;
* adeb = NULL ;
while ( printf("numéro x y : "),
scanf ("%d %f %f", &num, &x, &y), num)
{ courant = (struct element *) malloc (sizeof(struct
element)) ;
courant -> num = num ;
courant -> x = x ;
courant -> y = y ;
courant -> suivant = * adeb ;
* adeb = courant ;
}
}
```

Création d'une liste chaînée

FSO.UMPOUJDA.COM

Chapitre 2 : Les chaînes de caractères

En langage C, il n'existe pas de véritable type chaîne, dans la mesure où l'on ne peut pas y déclarer des variables d'un tel type. En revanche, il existe une convention de représentation des chaînes. Celle-ci est utilisée à la fois :

- ✓ par le compilateur pour représenter les chaînes constantes (notées entre doubles quotes) ;
- ✓ par un certain nombre de fonctions qui permettent de réaliser :
  - les lectures ou écritures de chaînes ;
  - les traitements classiques tels que concaténation, recopie, comparaison, extraction de sous-chaîne, conversions...

Mais, comme il n'existe pas de variables de type chaîne, il faudra prévoir un emplacement pour accueillir ces informations. Un tableau de caractères pourra faire l'affaire. C'est d'ailleurs ce que nous utiliserons dans ce chapitre. Mais nous verrons plus tard comment créer dynamiquement des emplacements mémoire, lesquels seront alors repérés par des pointeurs.

1 Représentation des chaînes

1.1 La convention adoptée

En C, une chaîne de caractères est représentée par une suite d'octets correspondant à chacun de ses caractères (plus précisément à chacun de leurs codes), le tout étant terminé par un octet supplémentaire de code nul. Cela signifie que, d'une manière générale, une chaîne de n caractères occupe en mémoire un emplacement de n+1 octets.

1.2 Cas des chaînes constantes

C'est cette convention qu'utilise le compilateur pour représenter les « constantes chaîne » (sous-entendu que vous les introduisez dans vos programmes), sous des notations de la forme : "bonjour"
De plus, une telle notation sera traduite par le compilateur en un pointeur (sur des éléments de type char) sur la zone mémoire correspondante.

Voici un programme illustrant ces deux particularités :

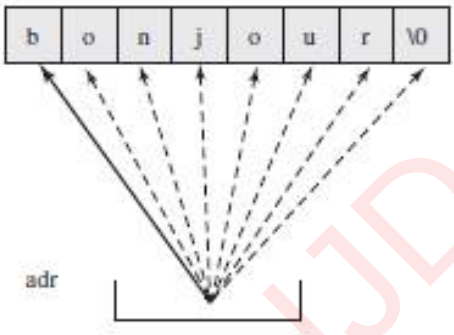
```
#include <stdio.h>
main()
{ char * adr ;
adr = "bonjour" ;
while (*adr)
{ printf ("%c", *adr) ;
adr++ ;
}
}
```

FSO.UMPOUJDA.COM

Convention de représentation des chaînes

La déclaration : ***char * adr ;*** réserve simplement l'emplacement pour un pointeur sur un caractère (ou une suite de caractères). En ce qui concerne la constante: ***"bonjour"*** le compilateur a créé en mémoire la suite d'octets correspondants mais, dans l'affectation : ***adr = "bonjour"*** la notation ***bonjour*** a comme valeur, non pas la valeur de la chaîne elle-même, mais son adresse ; on retrouve là le même phénomène que pour les tableaux.

Voici un schéma illustrant ce phénomène. La flèche en trait plein correspond à la situation après l'exécution de l'affectation : ***adr = "bonjour" ;*** les autres flèches correspondent à l'évolution de la valeur de ***adr***, au cours de la boucle.



1.3 Initialisation de tableaux de caractères

Si vous déclarez, par exemple : ***char ch[20] ;***
Vous ne pourrez pas pour autant transférer une chaîne constante dans ***ch***, en écrivant une affectation du genre : ***ch = "bonjour" ;***
En effet, ***ch*** est une constante pointeur qui correspond à l'adresse que le compilateur a attribuée au tableau ***ch*** ; ce n'est pas une ***lvalue*** ;
En revanche, C vous autorise à initialiser votre tableau de caractères à l'aide d'une chaîne constante. Ainsi, vous pourrez écrire : ***char ch[20] = "bonjour" ;***
Cela sera parfaitement équivalent à une initialisation de ***ch*** réalisée par une énumération de caractères (en n'omettant pas le code zéro – noté ***\0***) :

```
char ch[20] = { 'b','o','n','j','o','u','r','\0' }
```

1.4 Initialisation de tableaux de pointeurs sur des chaînes

Nous avons vu qu'une chaîne constante était traduite par le compilateur en une adresse que l'on pouvait, par exemple, affecter à un pointeur sur une chaîne. Cela peut se généraliser à un tableau de pointeurs, comme dans :

```
char * jour[7] = { "lundi", "mardi", "mercredi", "jeudi",
"vendredi", "samedi", "dimanche" } ;
```

Cette déclaration réalise donc à la fois la création des 7 chaînes constantes correspondant aux 7 jours de la semaine et l'initialisation du tableau ***jour*** avec les 7 adresses de ces 7 chaînes.

FSO.UMPOUJDA.COM

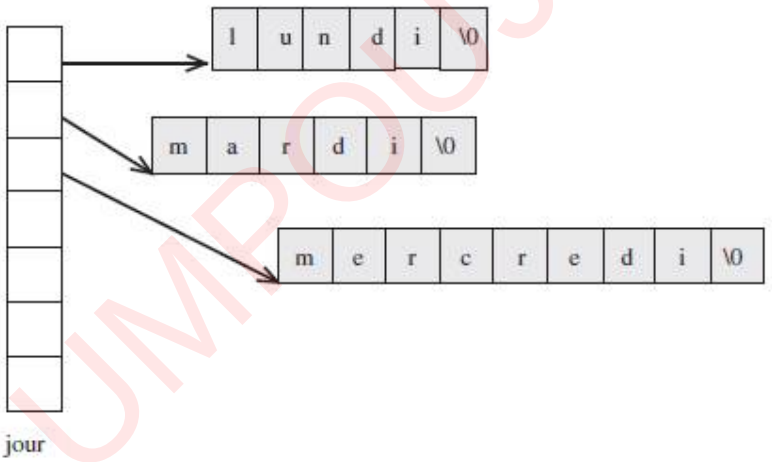
Voici un exemple employant cette déclaration (nous y avons fait appel, pour l'affichage d'une chaîne, au code de format **%s**, dont nous reparlerons un peu plus loin) :

```
main()
{ char * jour[7] = { "lundi", "mardi", "mercredi",
"jeudi","vendredi", "samedi", "dimanche" } ;
int i ;
printf("donnez un entier entre 1 et 7 : ") ;
scanf("%d", &i) ;
printf("le jour numéro %d de la semaine est %s", i,
jour[i-1] ) ;
}
```

Initialisation d'un tableau de pointeurs sur des chaînes

```
donnez un entier entre 1 et 7 : 3
le jour numéro 3 de la semaine est mercredi
```

La situation présentée ne doit pas être confondue avec la précédente. Ici, nous avons affaire à un tableau de sept pointeurs, chacun d'entre eux désignant une chaîne constante. Le schéma ci- après récapitule la situation.



2 Pour lire et écrire des chaînes

Le langage C offre plusieurs possibilités de lecture ou d'écriture de chaînes :

- L'utilisation du code de format ***%s*** dans les fonctions ***printf*** et ***scanf*** ;
- Les fonctions spécifiques de lecture (***gets***) ou d'affichage (***puts***) d'une chaîne (une seule à la fois).

Voyez cet exemple de programme :

```
#include <stdio.h>
main()
{ char nom[20], prenom[20], ville[25] ;
```

FSO.UMPOUJDA.COM

```
printf ("quelle est votre ville : ") ;
gets (ville) ;
printf ("donnez votre nom et votre prénom : ") ;
scanf ("%s %s", nom, prenom) ;
printf ("bonjour cher %s %s qui habite à ", prenom, nom)
;
puts (ville) ;
}
```

Entrées-sorties classiques de chaînes

```
quelle est votre ville : Oujda
donnez votre nom et votre prénom : MASOUDI Hamid
bonjour cher Hamid MASOUDI qui habite à Oujda
```

Les fonctions ***printf*** et ***scanf*** permettent de lire ou d'afficher simultanément plusieurs informations de type quelconque. En revanche, ***gets*** et ***puts*** ne traitent qu'une chaîne à la fois.

3 La lecture au clavier : gets et sscanf

Il est possible de régler la plupart des problèmes des entrées-sorties conversationnelles en travaillant en deux temps :

- Lecture d'une chaîne de caractères par ***gets*** (c'est-à-dire d'une suite de caractères quelconques validés par « return »);
- Décodage de cette chaîne suivant un format, à l'aide de la fonction **sscanf**.

En effet, une instruction telle que :
sscanf (adresse, format, liste_variables) effectue sur l'emplacement dont on lui fournit l'adresse (premier argument de type ***char ****) le même travail que ***scanf*** effectue sur son tampon. La différence est qu'ici nous sommes maître de ce tampon ; en particulier, nous pouvons décider d'appeler à nouveau ***sscanf*** sur une nouvelle zone de notre choix (ou sur la même zone dont nous avons modifié le contenu par ***gets***), sans être tributaire de la position du pointeur, comme cela était le cas avec ***scanf***.

Voici un exemple d'instructions permettant de questionner l'utilisateur jusqu'à ce qu'il ait fourni une réponse satisfaisante

```
#include <stdio.h>
#define LG 80
main()
{
int n, compte ;
char c ;
char ligne [LG+1] ;
do
{ printf ("donnez un entier et un caractère : ") ;
gets (ligne) ;
compte = sscanf (ligne, "%d %c", &n, &c) ;
}
```

```
while (compte < 2 ) ;
printf ("merci pour %d %c\n", n, c) ;
}
```

Contrôle des entrées avec gets et sscanf

```
donnez un entier et un caractère : bof
donnez un entier et un caractère : a 125
donnez un entier et un caractère : 12 bonjour
merci pour 12 b
```

4 Généralités sur les fonctions portant sur des chaînes

C dispose de nombreuses fonctions de manipulation de chaînes, voyons quelques principes généraux. Tout d'abord, rappelons qu'il n'y a pas de véritable type chaîne en C, mais simplement une convention de représentation. On ne peut donc jamais transmettre la valeur d'une chaîne, mais seulement son adresse, ou plus précisément un pointeur sur son premier caractère.

4.1 La fonction strlen

La fonction **strlen** fournit en résultat la longueur d'une chaîne dont on lui a transmis l'adresse en argument. Cette longueur correspond tout naturellement au nombre de caractères trouvés depuis l'adresse indiquée jusqu'au premier caractère de code nul, ce caractère n'étant pas pris en compte dans la longueur. Par exemple, l'expression : **strlen ("bonjour")** vaudra **7** ;
De même, avec : **char * adr = "salut" ;**
L'expression : **strlen (adr)** vaudra **5**.

5 Les fonctions de concaténation de chaînes

5.1 La fonction strcat

Voyez cet exemple :

```
#include <stdio.h>
#include <string.h>
main(){
char ch1[50] = "bonjour" ;
char * ch2 = " monsieur" ;
printf ("avant : %s\n", ch1) ;
strcat (ch1, ch2) ;
printf ("après : %s", ch1) ;
}
```

***Fonction* strcat**

```
avant : bonjour
après : bonjour monsieur
```

Notez la différence entre les deux déclarations (avec initialisation) de chacune des deux chaînes **ch1** et **ch2**. La première permet de réserver un emplacement plus grand que la constante chaîne qu'on y place initialement.

L'appel de **strcat** se présente ainsi (nous placerons souvent en regard de la présentation de l'appel d'une fonction le nom du fichier qui en contient le prototype) :
strcat (but, source) ***(string.h)***
Cette fonction recopie la seconde chaîne (**source**) à la suite de la première (**but**), après en avoir effacé le caractère de fin.

5.2 La fonction strncat

Cette fonction dont l'appel se présente ainsi : **strncat (but, source, lgmax)** ***(string.h)*** travaille de façon semblable à **strcat** en offrant en outre un contrôle sur le nombre de caractères qui seront concaténés à la chaîne d'arrivée (**but**).

En voici un exemple d'utilisation :

```
#include <stdio.h>
#include <string.h>
main()
{
char ch1[50] = "bonjour" ;
char * ch2 = " monsieur" ;
printf ("avant : %s\n", ch1) ;
strncat (ch1, ch2, 6) ;
printf ("après : %s", ch1) ;
}
```

***Fonction* strncat**

```
avant : bonjour
après : bonjour monsi
```

6 Les fonctions de comparaison de chaînes

Il est possible de comparer deux chaînes en utilisant l'ordre des caractères définis par leur code.
a) La fonction : **strcmp (chaîne1, chaîne2)** compare deux chaînes dont on lui fournit l'adresse et elle fournit une valeur entière définie comme étant :

- Positive si **chaîne1 > chaîne2** (c'est-à-dire si **chaîne1** arrive après **chaîne2**, au sens de l'ordre défini par le code des caractères) ;
- Nulle si **chaîne1 = chaîne2** (c'est-à-dire si ces deux chaînes contiennent exactement la même suite de caractères) ;
- Négative si **chaîne1 < chaîne2.**

Par exemple (quelle que soit l'implémentation) :

- **strcmp ("bonjour", "monsieur")** est négatif .
- **strcmp ("paris2", "paris10")** est positif.

FSO.UMPOUJDA.COM

b) La fonction : **strncmp (chaîne1, chaîne2, lgmax)** travaille comme **strcmp** mais elle limite la comparaison au nombre maximal de caractères indiqués par l'entier **lgmax**. Par exemple : **strncmp ("bonjour", "bon", 4)** est positif tandis que : **strncmp ("bonjour", "bon", 2)** vaut zéro.
c) Enfin, deux fonctions :

- **stricmp (chaîne1, chaîne2) (*string.h*)**
- **strnicmp (chaîne1, chaîne2, lgmax) (*string.h*)**

travaillent respectivement comme **strcmp** et **strncmp**, mais sans tenir compte de la différence entre majuscules et minuscules (pour les seuls caractères alphabétiques).

7 Les fonctions de copie de chaînes

a) La fonction : **strcpy (but, source)** ***(string.h)*** recopie la chaîne située à l'adresse **source** dans l'emplacement d'adresse **destin**. Là encore, il est nécessaire que la taille du second emplacement soit suffisante pour accueillir la chaîne à recopier, sous peine d'écrasement intempestif. Cette fonction fournit comme résultat l'adresse de la chaîne **but**.
b) La fonction : **strncpy (but, source, lgmax)** ***(string.h)*** procède de manière analogue à **strcpy**, en limitant la recopie au nombre de caractères précisés par l'expression entière **lgmax**.
Notez bien que, si la longueur de la chaîne source est inférieure à cette longueur maximale, son caractère de fin (**\0**) sera effectivement recopié. Mais, dans le cas contraire, il ne le sera pas. L'exemple suivant illustre les deux situations :

```
#include <stdio.h>
#include <string.h>
main(){
char ch1[20] = "xxxxxxxxxxxxxxxxxxx" ;
char ch2[20] ;
printf ("donnez un mot : ") ;
gets (ch2) ;
strncpy (ch1, ch2, 6) ;
printf ("%s", ch1) ;
}
```

Fonctions de recopie de chaînes : **strcpy** ***et*** **strncpy**

```
donnez un mot : bon
bon

_______
donnez un mot : bonjour
bonjouxxxxxxxxxxxxx
```

8 Les fonctions de recherche dans une chaîne

On trouve, en langage C, des fonctions classiques de recherche de l'occurrence dans une chaîne d'un caractère ou d'une autre chaîne (nommée alors sous-chaîne). Elles

fournissent comme résultat **un pointeur de type char *** sur l'information cherchée en cas de succès, et le pointeur nul dans le cas contraire. Voici les principales.

- **strchr (chaîne, caractère)** ***(string.h)*** recherche, dans **`chaîne`**, la première position où apparaît le caractère mentionné.
- **strrchr (chaîne, caractère)** ***(string.h)*** réalise le même traitement que **`strchr`**, mais en explorant la chaîne concernée à partir de la fin. Elle fournit donc la dernière occurrence du caractère mentionné.
- **strstr (chaîne, sous-chaîne)** ***(string.h)*** recherche, dans **`chaîne`**, la première occurrence complète de la sous-chaîne mentionnée.

FSO.UMPOUJDA.COM

Chapitre 3 : Les structures et les énumérations

1 Déclaration d'une structure

Voyez tout d'abord cette déclaration :

```
struct enreg { int numero ;
               int qte ;
               float prix ;
               } ;
```

Celle-ci définit un **modèle de structure** mais ne réserve pas de variables correspondant à cette structure. Ce modèle s'appelle ici **enreg** et il précise le nom et le type de chacun des champs constituant la structure **(numero, qte et prix).**
Une fois un tel modèle défini, nous pouvons déclarer des variables du type correspondant (souvent, nous parlerons de structure pour désigner une variable dont le type est un modèle de structure).
Par exemple : **struct enreg art1 ;** réserve un emplacement nommé **art1** «de type **enreg** » destiné à contenir deux entiers et un flottant.
De manière semblable : **struct enreg art1, art2 ;** réserve deux emplacements **art1** et **art2** du type **enreg.**

2 Utilisation d'une structure

En C, on peut utiliser une structure de deux manières :

- En travaillant individuellement sur chacun de ses champs ;
- En travaillant de manière globale sur l'ensemble de la structure.

2.1 Utilisation des champs d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur « point » (**.**) suivi du nom de champ.
Voici quelques exemples utilisant le modèle **enreg** et les variables **art1** et **art2** déclarées de ce type.
art1.numero = 15 ; affecte la valeur **15** au champ **numero** de la structure **art1**.
printf ("%e", art1.prix) ; affiche, suivant le code format **%e**, la valeur du champ prix de la structure **art1**.
scanf ("%e", &art2.prix) ; lit, suivant le code format **%e**, une valeur qui sera affectée au champ **prix** de la structure **art2**. Notez bien la présence de l'opérateur **&.**
art1.numero++ incrémente de **1** la valeur du champ **numero** de la structure **art1**.

FSO.UMPOUJDA.COM

2.2 Utilisation globale d'une structure

Il est possible d'affecter à une structure le contenu d'une structure définie à partir du **même modèle**. Par exemple, si les structures **art1** et **art2** ont été déclarées suivant le modèle **enreg** défini précédemment, nous pourrons écrire : **art1 = art2 ;**
Une telle affectation globale remplace avantageusement :

```
art1.numero = art2.numero ;
art1.qte = art2.qte ;
art1.prix = art2.prix ;
```

Notez bien qu'une affectation globale n'est possible que si les structures ont été **définies avec le même nom de modèle** ; en particulier, elle sera impossible avec des variables ayant une structure analogue mais définies sous deux noms différents.
L'opérateur d'affectation et **l'opérateur d'adresse &** sont les seuls opérateurs s'appliquant à une structure (de manière globale).

2.3 Initialisations de structures

On retrouve pour les structures les règles d'initialisation qui sont en vigueur pour tous les types de variables, à savoir :

- En l'absence d'initialisation explicite, les structures de classe statique sont, par défaut, initialisées à zéro ; celles possédant la classe automatique ne sont pas initialisées par défaut (elles contiendront donc des valeurs aléatoires).
- Il est possible d'initialiser explicitement une structure lors de sa déclaration. On ne peut toutefois employer que des constantes ou des expressions constantes et cela aussi bien pour les structures statiques que pour les structures automatiques.

Voici un exemple d'initialisation de notre structure **art1**, au moment de sa déclaration : `struct enreg art1 = { 100, 285, 2000 } ;`
Vous voyez que la description des différents champs se présente sous la forme d'une liste de valeurs séparées par des virgules, chaque valeur étant une constante ayant le type du champ correspondant. Là encore, il est possible d'omettre certaines valeurs.

3 Pour simplifier la déclaration de types : définir des synonymes avec *typedef*

La déclaration **typedef** permet de définir ce que l'on nomme en langage C des ***types synonymes*.**
A priori, elle s'applique à tous les types et pas seulement aux structures. C'est pourquoi nous commencerons par l'introduire sur quelques exemples avant de montrer l'usage que l'on peut en faire avec les structures.

3.1 Exemples d'utilisation de *typedef*

La déclaration : **typedef int entier ;** signifie que **entier** est synonyme de **int**, de sorte que les déclarations suivantes sont équivalentes :

FSO.UMPOUJDA.COM

```
int n, p ; entier n, p ;
```

De même : **typedef int * ptr ;** signifie que **ptr** est synonyme de **int ***. Les déclarations suivantes sont équivalentes : **int * p1, * p2 ; ptr p1, p2 ;**

3.2 Application aux structures

En faisant usage de **typedef**, les déclarations des structures **art1** et **art2** du paragraphe 1 peuvent être réalisées comme suit :

```
struct enreg { int numero ;
               int qte ;
               float prix ;
             } ;

typedef struct enreg s_enreg ;
s_enreg art1, art2 ;
```

ou encore, plus simplement :

```
typedef struct { int numero ;
                 int qte ;
                 float prix ;
               } s_enreg ;

s_enreg art1, art2 ;
```

4 Imbrication de structures

Dans nos exemples d'introduction des structures, nous nous sommes limités à une structure simple ne comportant que trois champs d'un type de base. Mais chacun des champs d'une structure peut être d'un type absolument quelconque : pointeur, tableau, structure... Il peut même s'agir de pointeurs sur des structures du type de la structure dans laquelle ils apparaissent.

4.1 Structure comportant des tableaux

Soit la déclaration suivante :

```
struct personne { char nom[30] ;
                  char prenom [20] ;
                  float heures [31] ;
                } employe, courant ;
```

Celle-ci réserve les emplacements pour deux structures nommées **employe** et **courant**. Ces dernières comportent trois champs :

- **nom** qui est un tableau de **30 caractères** ;
- **prenom** qui est un tableau de **20 caractères ;**
- **heures** qui est un tableau de **31 flottants.**

On peut, par exemple, imaginer que ces structures permettent de conserver pour un employé d'une entreprise les informations suivantes :

- nom ;

FSO.UMPOUJDA.COM

- prénom ;
- nombre d'heures de travail effectuées pendant chacun des jours du mois courant.

La notation : **employe.heures[4]** désigne le cinquième élément du tableau **heures** de la structure **employe**. Il s'agit d'un élément de type **float**. Notez que, malgré les priorités identiques des opérateurs **.** et **[]**, leur associativité de gauche à droite évite l'emploi de parenthèses.

De même : **employe.nom[0]** représente le premier caractère du champ **nom** de la structure **employe**.

Par ailleurs : **&courant.heures[4]** représente l'adresse du cinquième élément du tableau **heures** de la structure **courant**.

Notez que, la priorité de l'opérateur **&** étant inférieure à celle des deux autres, les parenthèses ne sont, là encore, pas nécessaires.

Enfin : **courant.nom** représente le champ **nom** de la structure **courant**, c'est-à-dire plus précisément l'adresse de ce tableau.

À titre indicatif, voici un exemple d'initialisation d'une structure de type **personne** lors de sa déclaration :

```
struct personne emp={"Dupont","Jules",{8,7,8, 6, 8, 0, 0, 8}};
```

4.2 Tableaux de structures

Voyez ces déclarations :

```
struct point { char nom ;
               int x ;
               int y ;
             } ;
struct point courbe [50] ;
```

La structure **point** pourrait, par exemple, servir à représenter un point d'un plan, point qui serait défini par son nom (caractère) et ses deux coordonnées.
Notez bien que **point** est un nom de modèle de structure, tandis que **courbe** représente effectivement un tableau de **50 éléments** du type **point**.
Si **i** est un entier, la notation : **courbe[i].nom** représente le nom du point de rang **i** du tableau **courbe**. Il s'agit donc d'une valeur de type **char**. Notez bien que la notation : **courbe.nom[i]** n'aurait pas de sens.
De même, la notation : **courbe[i].x** désigne la valeur du champ **x** de l'élément de rang **i** du tableau **courbe**.
Par ailleurs : **courbe[4]** représente la structure de type **point** correspondant au **cinquième élément** du tableau **courbe**.
Enfin **courbe** est un identificateur de tableau, et, comme tel, désigne son adresse de début. Là encore, voici, à titre indicatif, un exemple d'initialisation (partielle) de notre variable **courbe**, lors de sa déclaration :

```
struct point courbe[50]= { {'A', 10, 25}, {'M', 12, 28},,
{'P', 18,2} };
```

4.3 Structures comportant d'autres structures

Supposez que, à l'intérieur de nos structures **employe** et courant définies dans le paragraphe 4.1, nous ayons besoin d'introduire deux dates : la date d'embauche et la date d'entrée dans le dernier poste occupé. Si ces dates sont elles-mêmes des structures comportant trois champs correspondant au jour, au mois et à l'année, nous pouvons alors procéder aux déclarations suivantes :

```
struct date { int jour ;
              int mois ;
              int annee ;
            } ;

struct personne { char nom[30] ;
                  char prenom[20] ;
                  float heures [31] ;
                  struct date date_embauche ;
                  struct date date_poste ;
                } employe, courant ;
```

Vous voyez que la seconde déclaration fait intervenir un modèle de structure (**date**) précédemment défini.

La notation : **employe.date_embauche.annee** représente l'année d'embauche correspondant à la structure **employe**. Il s'agit d'une valeur de type **int**.
courant.date_embauche représente la date d'embauche correspondant à la structure **courant**. Il s'agit cette fois d'une structure de type **date**. Elle pourra éventuellement faire l'objet d'affectations globales comme dans :
courant.date_embauche = employe.date_poste ;

5 À propos de la portée du modèle de structure

À l'image de ce qui se produit pour les identificateurs de variables, la portée d'un modèle de structure dépend de l'emplacement de sa déclaration :

- Si elle se situe au sein d'une fonction (y compris, la fonction **main**), elle n'est accessible que depuis cette fonction ;
- Si elle se situe en dehors d'une fonction, elle est accessible de toute la partie du fichier source qui suit sa déclaration ; elle peut ainsi être utilisée par plusieurs fonctions.

Voici un exemple d'un modèle de structure nommé **enreg** déclaré à un niveau global et accessible depuis les fonctions **main** et **fct**.

```
struct enreg { int numero ;
               int qte ;
               float prix ;
             } ;

main () { struct enreg x ; .... }
fct ( ....) { struct enreg y, z ; .... }
```

FSO.UMPOUJDA.COM

6 Transmission d'une structure en argument d'une fonction

Jusqu'ici, nous avons vu qu'en C la transmission des arguments se fait par valeur, ce qui implique une recopie de l'information transmise à la fonction. Par ailleurs, il est toujours possible de transmettre la valeur d'un pointeur sur une variable, auquel cas la fonction peut, si besoin est, en modifier la valeur. Ces remarques s'appliquent également aux structures (notez qu'il n'en allait pas de même pour un tableau, dans la mesure où la seule chose qu'on puisse transmettre dans ce cas soit la valeur de l'adresse de ce tableau).

6.1 Transmission de la valeur d'une structure

Voici un exemple simple :

```
#include <stdio.h>
struct enreg { int a ;
               float b ;
             } ;
main() {
struct enreg x ;
void fct (struct enreg y) ;
x.a = 1; x.b = 12.5;
printf ("\navant appel fct : %d %e",x.a,x.b);
fct (x) ;
printf ("\nau retour dans main : %d %e", x.a, x.b);
}
void fct (struct enreg s) {
s.a = 0; s.b=1;
printf ("\ndans fct : %d %e", s.a, s.b);
}
```

Transmission en argument des valeurs d'une structure

```
avant appel fct : 1 1.25000e+01
dans fct : 0 1.00000e+00
au retour dans main : 1 1.25000e+01
```

Naturellement, les valeurs de la structure **x** sont recopiées localement dans la fonction **fct** lors de son appel ; les modifications de **s** au sein de **fct** n'ont aucune incidence sur les valeurs de **x**.

6.2 Transmission de l'adresse d'une structure : l'opérateur ->

Cherchons à modifier notre précédent programme pour que la fonction **fct** reçoive effectivement l'adresse d'une structure et non plus sa valeur. L'appel de **fct** devra donc se présenter sous la forme : **fct (&x) ;**

FSO.UMPOUJDA.COM

Cela signifie que son en-tête sera de la forme :

```
void fct (struct enreg * ads) ;
```

Comme vous le constatez, le problème se pose alors d'accéder, au sein de la définition de **fct**, à chacun des champs de la structure d'adresse **ads**. L'opérateur « **.** » ne convient plus, car il suppose comme premier opérande un nom de structure et non une adresse. Deux solutions s'offrent alors à vous :

- Adopter une notation telle que **(*ads).a** ou **(*ads).b** pour désigner les champs de la structure d'adresse **ads** ;
- Faire appel à un nouvel opérateur noté **->**, lequel permet d'accéder aux différents champs d'une structure à partir de son adresse de début. Ainsi, au sein de **fct**, la notation **ads -> b** désignera le second champ de la structure reçue en argument ; elle sera équivalente à **(*ads).b.**

Voici notre précédent exemple en employant l'opérateur noté **->** :

```
#include <stdio.h>
struct enreg { int a ;
               float b ;
             } ;
main() {
struct enreg x ;
void fct (struct enreg *) ;
x.a = 1; x.b = 12.5;
printf ("\navant appel fct : %d %e",x.a,x.b);
fct (&x) ;
printf ("\nau retour dans main : %d %e", x.a, x.b);
}
void fct (struct enreg * ads)
{
ads->a = 0 ; ads->b = 1;
printf ("\ndans fct : %d %e", ads->a, ads->b);
}
```

Transmission en argument de l'adresse d'une structure

```
avant appel fct : 1 1.25000e+01
dans fct : 0 1.00000e+00
au retour dans main : 0 1.00000e+00
```

7 Transmission d'une structure en valeur de retour d'une fonction

Sachez que C vous autorise à réaliser des fonctions qui fournissent en retour la valeur d'une structure. Par exemple, avec le modèle **enreg** précédemment défini, nous pourrions envisager une situation de ce type :

```
struct enreg fct (...)
{ struct enreg s ; /* structure locale à fct */
.....
```

FSO UMPOUJDA.COM

```
return s ; /* dont la fonction renvoie la valeur */
}
```

Notez bien que **s** aura dû soit être créée localement par la fonction (comme c'est le cas ici), soit éventuellement reçue en argument.
Naturellement, rien ne vous interdit, par ailleurs, de réaliser une fonction qui renvoie comme résultat un pointeur sur une structure.

8 Les énumérations

Un type énumération est un cas particulier de type entier et donc un type scalaire (ou simple). Son seul lien avec les structures présentées précédemment est qu'il forme, lui aussi, un type défini par le programmeur.

8.1 Exemples introductifs

Considérons cette déclaration : **enum couleur {jaune, rouge, bleu, vert} ;**
Elle définit un type énumération nommé **couleur** et précise qu'il comporte quatre valeurs possibles désignées par les identificateurs **jaune, rouge, bleu** et **vert**. Ces valeurs constituent les constantes du type **couleur**.
Il est possible de déclarer des variables de type **couleur** :

```
enum couleur c1, c2 ; /* c1 et c2 sont deux variables */
                      /* de type enum couleur         */
```

Les instructions suivantes sont alors tout naturellement correctes :

```
c1 = jaune ; /* affecte à c1 la valeur jaune */
c2 = c1 ; /* affecte à c2 la valeur contenue dans c1 */
```

Comme on peut s'y attendre, les identificateurs correspondant aux constantes du type **couleur** ne sont pas des ***lvalue*** et ne sont donc pas modifiables :

```
jaune = 3 ; /* interdit : jaune n'est pas une lvalue */
```

8.2 Propriétés du type énumération

- ***Nature des constantes figurant dans un type énumération***

Les constantes figurant dans la déclaration d'un type énumération sont des entiers ordinaires. Ainsi, la déclaration précédente :
enum couleur {jaune, rouge, bleu, vert} ; associe simplement une valeur de type **int** à chacun des quatre identificateurs cités. Plus précisément, elle attribue la valeur **0** au premier identificateur **jaune**, la valeur **1** à l'identificateur **rouge**, etc. Ces identificateurs sont utilisables en lieu et place de n'importe quelle constante entière :

```
int n ;
long p, q ;
.....
n = bleu ; /* même rôle que n = 2 */
p = vert * q + bleu ; /* même rôle que p = 3 * q + 2 */
```

- ***Une variable d'un type énumération peut recevoir une valeur quelconque***

Contrairement à ce qu'on pourrait espérer, il est possible d'affecter à une variable de type énuméré n'importe quelle valeur entière (pour peu qu'elle soit représentable dans le type **int**) :

```
enum couleur {jaune, rouge, bleu, vert} ;
enum couleur c1, c2 ;
.....
c1 = 2 ; /* même rôle que c1 = bleu ; */
c1 = 25 ; /* accepté, bien que 25 n'appartienne pas au */
          /* type type enum couleur */
```

Qui plus est, on peut écrire des choses aussi absurdes que :

```
enum booleen { faux, vrai } ;
enum couleur {jaune, rouge, bleu, vert} ;
enum booleen drapeau ;
enum couleur c ;
c = drapeau ; /* OK bien que drapeau et c ne soit pas
                 d'un même type */
drapeau = 3 * c + 4 ; /* accepté */
```

- ***Les constantes d'un type énumération peuvent être quelconques***

Dans les exemples précédents, les valeurs des constantes attribuées aux identificateurs apparaissant dans un type énumération étaient déterminées automatiquement par le compilateur.
Mais il est possible d'influer plus ou moins sur ces valeurs, comme dans :

```
enum couleur_bis { jaune = 5,rouge,bleu,vert = 12,rose};
/* jaune = 5,rouge = 6, bleu = 7, vert = 12, rose = 13 */
```

Les entiers négatifs sont permis comme dans :

```
enum couleur_ter {jaune = -5,rouge,bleu,vert=12 ,rose } ;
/* jaune = -5,rouge = -4,bleu = -3,vert = 12,rose = 13 */
```

En outre, rien n'interdit qu'une même valeur puisse être attribuée à deux identificateurs différents :

```
enum couleur_ter {jaune=5,rouge,bleu,vert=6,noir,violet};
/* jaune=5,rouge =6,bleu =7,vert =6,noir =7,violet=8 */
```

FSO.UMPOUJDA.COM

Chapitre 4 : La programmation modulaire et les fonctions

Comme tous les langages, C permet de découper un programme en plusieurs parties nommées souvent « modules ». Cette programmation dite modulaire se justifie pour de multiples raisons :

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le programme principal les instructions en décrivant les enchaînements.
- La programmation modulaire permet d'éviter des séquences d'instructions répétitives.

1 La fonction : la seule sorte de module existant en C

Dans certains langages, on trouve deux sortes de modules, à savoir :

- Les **fonctions**, assez proches de la notion mathématique correspondante. Notamment, une fonction dispose d'arguments (en C, comme dans la plupart des autres langages, une fonction peut ne comporter aucun argument) qui correspondent à des informations qui lui sont transmises et elle fournit un unique résultat scalaire (simple) ; désigné par le nom même de la fonction, ce dernier peut apparaître dans une expression. On dit d'ailleurs que la fonction possède une valeur et qu'un appel de fonction est assimilable à une expression.
- Les **procédures** (terme Pascal) ou sous-programmes (terme Fortran ou Basic) qui élargissent la notion de fonction. La procédure ne possède plus de valeur à proprement parler et son appel ne peut plus apparaître au sein d'une expression. Par contre, elle dispose toujours d'arguments. Parmi ces derniers, certains peuvent, comme pour la fonction, correspondre à des informations qui lui sont transmises. Mais d'autres, contrairement à ce qui se passe pour la fonction, peuvent correspondre à des informations qu'elle produit en retour de son appel. De plus, une procédure peut réaliser une action, par exemple afficher un message (en fait, dans la plupart des langages, la fonction peut quand même réaliser une action, bien que ce ne soit pas là sa vocation).

En C, il n'existe qu'une seule sorte de module, nommé **fonction.** De plus, cette fonction pourra prendre des aspects différents. Par exemple :

- La valeur d'une fonction pourra très bien ne pas être utilisée; c'est ce qui se passe fréquemment lorsque vous utilisez **`printf`** ou **`scanf`**. Bien entendu, cela n'a d'intérêt que parce que de telles fonctions réalisent une **action** (ce qui, dans d'autres langages, serait réservée aux sous-programmes ou procédures).
- Une fonction pourra ne fournir aucune valeur.
- Une fonction pourra fournir un résultat non scalaire.
- Une fonction pourra modifier les valeurs de certains de ses arguments (par les pointeurs).

FSO.UMPOUJDA.COM

Ainsi, donc, malgré son nom, en C, la fonction pourra jouer un rôle aussi général que la procédure ou le sous-programme des autres langages.

2 Exemple de définition et d'utilisation d'une fonction en C

Nous proposons un exemple simple de fonction correspondant à l'idée usuelle que l'on se fait d'une fonction, c'est-à-dire recevant des arguments et fournissant une valeur.

```
#include <stdio.h>

/***** le programme principal (fonction main) *****/
main() {

/* déclaration de fonction fexple */
float fexple (float, int, int) ;
float x = 1.5 ;
float y, z ;
int n = 3, p = 5, q = 10 ;

/* appel de fexple avec les arguments x, n et p */
y = fexple (x, n, p) ;
printf ("valeur de y : %e\n", y) ;

/* appel de fexple avec les arguments x+0.5, q et n-1 */
z = fexple (x+0.5, q, n-1) ;
printf ("valeur de z : %e\n", z) ;
}

/*************** la fonction fexple ***************/
float fexple (float x, int b, int c)
{ float val ;

/* déclaration d'une variable "locale" à fexple */
val = x * x + b * x + c ;
return val ;
}
```

Exemple de définition et d'utilisation d'une fonction

Nous y trouvons tout d'abord, de façon désormais classique, un programme principal formé d'un bloc. Mais, cette fois, à sa suite, apparaît la **définition d'une fonction**. Celle-ci possède une structure voisine de la fonction **`main`**, à savoir un en-tête et un corps délimité par des accolades ({ et }). Mais l'en-tête est plus élaboré que celui de la fonction **`main`** puisque, outre le nom de la fonction (**`fexple`**), on y trouve une liste d'arguments (**nom + type**), ainsi que le type de la valeur qui sera fournie par la fonction (on la nomme indifféremment « résultat », « valeur de la fonction », « valeur de retour »...) :

FSO UMPOUJDA.COM

```
float          fexple     (float x,     int b,       int c)
  ↓              ↓            ↓           ↓            ↓
type de la     nom de la    premier     deuxième     troisième
"valeur        fonction     argument    argument     argument
de retour"                (type float) (type int)  (type int)
```

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.
Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration : **`float val ;`** Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type **`float`** nommée **`val`**. On dit que **`val`** est une variable locale à la fonction **`fexple`**, de même que les variables telles que **`n, p, y...`** sont des variables locales à la fonction **`main`** (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de portée qui s'y attache.
L'instruction suivante de notre fonction **`fexple`** est une affectation classique (faisant toutefois intervenir les valeurs des arguments **`x, n`** et **`p`**).

Enfin, l'instruction **`return val`** précise la valeur que fournira la fonction à la fin de son travail.
En définitive, on peut dire que **`fexple`** est une fonction telle que **`fexple(x,b,c)`** fournisse la valeur de l'expression $x^2 + bx + c$. Notez bien l'aspect arbitraire du nom des arguments ; on obtiendrait la même définition de fonction avec, par exemple :

```
float fexple(float z, int coef, int n)
{/* déclaration d'une variable "locale" à fexple */
float val ;
val = z * z + coef * z + n ;
return val ;
}
```

Examinons maintenant la fonction **`main`**. Vous constatez qu'on y trouve une déclaration : **`float fexple (float, int, int) ;`**
Elle sert à prévenir le compilateur que **`fexple`** est une fonction et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration.
Quant à l'utilisation de notre fonction **`fexple`** au sein de la fonction **`main`**, elle est classique et comparable à celle d'une fonction prédéfinie telle que **`scanf`** ou **`sqrt`**. Ici, nous nous sommes contentés d'appeler notre fonction à deux reprises avec des arguments différents.

3 Quelques règles

3.1 Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des «**arguments muets** », ou encore « **arguments formels** » ou « **paramètres formels** ».

FSO.UMPOUJDA.COM

Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.
Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des «**arguments effectifs** » (ou encore « **paramètres effectifs** »). Comme le laisse deviner l'exemple précédent, on peut utiliser n'importe quelle expression comme argument effectif ; au bout du compte, c'est la valeur de cette expression qui sera transmise à la fonction lors de son appel. Notez qu'une telle « liberté » n'aurait aucun sens dans le cas des paramètres formels : il serait impossible d'écrire un en-tête de **`fexple`** sous la forme **`float fexple (float a+b, ...)`** pas plus qu'en mathématiques vous ne définiriez une fonction f par $f(x + y) = 5$!

3.2 L'instruction return

Voici quelques règles générales concernant cette instruction.

- L'instruction **`return`** peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction **`fexple`** précédente d'une manière plus simple :

```
float fexple (float x, int b, int c)
{
      return (x * x + b * x + c) ;
}
```

- L'instruction **`return`** peut apparaître à plusieurs reprises dans une fonction, comme dans cet autre exemple :

```
double absom (double u, double v)
{
      double s ;
      s = a + b ;
      if (s>0) return (s) ;
      else return (-s)
}
```

Notez bien que non seulement l'instruction **`return`** définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée (n'oubliez pas qu'en C tous les modules sont des fonctions, y compris le programme principal). Nous verrons qu'une fonction peut ne fournir aucune valeur : elle peut alors disposer de une ou plusieurs instructions **`return` sans expression**, interrompant simplement l'exécution de la fonction ; mais elle peut aussi dans ce cas ne comporter aucune instruction **`return`**, le retour étant alors mis en place automatiquement par le compilateur à la fin de la fonction.

- Si le type de l'expression figurant dans **`return`** est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion.

Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. C'est d'ailleurs ce que nous avons fait fréquemment avec **`printf`** ou **`scanf`**. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de

FSO.UMPOUJDA.COM

calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire !).

3.3 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé **void**. Par exemple, voici l'en-tête d'une fonction recevant un argument de type **int** et ne fournissant aucune valeur :
void sansval (int n) et voici quelle serait sa déclaration :
void sansval (int) ;

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction **return**. Certains compilateurs ne détecteront toutefois pas l'erreur.
Quand une fonction ne reçoit aucun argument, on place le mot-clé **void** (le même que précédemment, mais avec une signification différente !) à la place de la liste d'arguments (attention, en C++, la règle sera différente : on se contentera de ne rien mentionner dans la liste d'arguments). Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type **float** (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :
float tirage (void)

Sa déclaration serait très voisine(elle ne diffère que par la présence du point-virgule!):
float tirage (void) ;

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son en-tête sera de la forme :
void message (void) et sa déclaration sera : **void message (void) ;**

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction **affiche_carres** qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments et une fonction **erreur** qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

```
#include <stdio.h>
main(){
void affiche_carres (int,int);/*prototype de affiche_carres*/
void erreur (void) ; /* prototype de erreur */
int debut = 5, fin = 10 ;
.....
affiche_carres (debut, fin) ;
.....
if (...) erreur () ;
}
void affiche_carres (int d, int f){ int i ;
for (i=d ; i<=f ; i++)
printf ("%d a pour carré %d\n", i, i*i) ;
```

```
}
void erreur (void)
{ printf ("*** erreur ***\n") ; }
```

4 Les fonctions et leurs déclarations

4.1 Les différentes façons de déclarer (ou de ne pas déclarer) une fonction

Dans notre précédant exemple, nous avions fourni la définition de la fonction **fexple** **après** celle de la fonction **main**. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float x, int b, int c)
{
....
}
main()
{
float fexple (float,int,int);/*déclaration de la fonc.fexple */
.....
y = fexple (x, n, p) ;
.....
}
```

En toute rigueur, dans ce cas, la déclaration de la fonction **fexple** (ici, dans **main**) est facultative, car, lorsqu'il traduit la fonction **main**, le compilateur connaît déjà la fonction **fexple**.

4.2 Où placer la déclaration d'une fonction

La tendance la plus naturelle consiste à placer la déclaration d'une fonction à l'intérieur des déclarations de toute fonction l'utilisant ; c'est ce que nous avons fait jusqu'ici. Et, de surcroît, dans tous nos exemples précédents, la fonction utilisatrice était la fonction **main** elle-même !
Dans ces conditions, nous avions affaire à une déclaration locale dont la portée était limitée à la fonction où elle apparaissait.
Mais il est également possible d'utiliser des déclarations globales, en les faisant apparaître **avant la définition de la première fonction**. Par exemple, avec :

```
float fexple (float, int, int) ;
main()
{ .....
}
void f1 (...)
{ .....
}
```

La déclaration de **fexple** est connue à la fois de **main** et de **f1**.

4.3 À quoi sert la déclaration d'une fonction

Nous avons vu que la déclaration d'une fonction est plus ou moins obligatoire et qu'elle peut être plus ou moins détaillée. Malgré tout, nous vous avons recommandé d'employer toujours la forme la plus complète possible qu'on nomme prototype. Dans ce cas, un tel prototype peut être utilisé par le compilateur, et cela de deux façons complètement différentes.

a) Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.

b) Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction **fexple**, un appel tel que : **fexple (n+1, 2*x, p)** sera traduit par :

- L'évaluation de la valeur de l'expression **n+1** (en **int**) et sa conversion en **float**,
- L'évaluation de la valeur de l'expression **2*x** (en **float**) et sa conversion en **int** ; il y a donc dans ce dernier cas une conversion dégradante.

5 Retour sur les fichiers en-tête

Nous avons déjà dit qu'il existe un certain nombre de fichiers d'extension **.h**, correspondant chacun à une classe de fonctions. On y trouve, entre autres choses, les prototypes de ces fonctions. Ce point se révèle fort utile :

- D'une part pour effectuer des contrôles sur le nombre et le type des arguments mentionnés dans les appels de ces fonctions,
- D'autre part pour forcer d'éventuelles conversions auxquelles on risque de ne pas penser.

À titre d'illustration de ce dernier aspect, supposez que vous ayez écrit ces instructions :

```
float x, y ;
.....
y = sqrt (x) ;
.....
```

sans les faire précéder d'une quelconque directive **#include**.
Elles produiraient alors des **résultats faux**. En effet, il se trouve que la fonction **sqrt** s'attend à recevoir un argument de type **double** (ce qui sera le cas ici, compte tenu des conversions implicites), et elle fournit un résultat de type **double**. Or, lors de la traduction de votre programme, le compilateur ne le sait pas. Il attribue donc d'office à **sqrt** le type **int** et il met en place une conversion de la valeur de retour (laquelle sera en fait de type **double**) en **int**.

On se trouve en présence des conséquences habituelles d'une mauvaise interprétation de type. Un premier remède consiste à placer dans votre module la déclaration :
`double sqrt(double) ;` mais encore faut-il que vous connaissiez de façon certaine le type de cette fonction. Une meilleure solution consiste à placer, en début de votre programme, la directive :
`#include <math.h>` laquelle incorporera automatiquement le prototype approprié (entre autres choses).

6 En C, les arguments sont transmis par valeur

Nous avons déjà eu l'occasion de dire qu'en C les arguments d'une fonction étaient transmis par valeur. Cependant, dans les exemples que nous avons rencontrés dans ce chapitre, les conséquences et les limitations de ce mode de transmission n'apparaissaient guère. Or voyez cet exemple :

```
#include <stdio.h>
main()
{ void echange (int a, int b) ;
int n=10, p=20 ;
printf ("avant appel : %d %d\n", n, p) ;
echange (n, p) ;
printf ("après appel : %d %d", n, p)
}
void echange (int a, int b)
{
int c ;
printf ("début echange : %d %d\n", a, b) ;
c = a ;
a = b ;
b = c ;
printf ("fin echange : %d %d\n", a, b) ;
}
```

Conséquences de la transmission par valeur des arguments

```
avant appel : 10 20
début echange : 10 20
fin echange : 20 10
après appel : 10 20
```

La fonction **`echange`** reçoit deux valeurs correspondant à ses deux arguments muets **`a`** et **`b`**. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs **`n`** et **`p`**.
En effet, lors de l'appel de **`echange`**, il y a eu transmission de la valeur des expressions **`n`** et **`p`**. On peut dire que ces valeurs ont été recopiées localement dans la fonction **`echange`** dans des emplacements nommés **`a`** et **`b`**. C'est effectivement sur ces copies qu'a travaillé la fonction **`echange`**, de sorte que les valeurs des variables **`n`** et **`p`** n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

FSO.UMPOUJDA.COM

Ce mode de transmission semble donc interdire a priori qu'une fonction produise une ou plusieurs valeurs en retour, autres que celle de la fonction elle-même. Or, il ne faut pas oublier qu'en C tous les modules doivent être écrits sous forme de fonction. Autrement dit, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction. Nous verrons que ce problème possède plusieurs solutions, à savoir :

- Transmettre en argument la valeur de l'adresse d'une variable. La fonction pourra éventuellement agir sur le contenu de cette adresse.
- Utiliser des variables globales ; cette deuxième solution devra toutefois être réservée à des cas exceptionnels, compte tenu des risques qu'elle présente (effets de bords).

7 Les variables globales

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.
En fait, en C, plusieurs fonctions (dont, bien entendu le programme principal **`main`**) peuvent partager des variables communes qu'on qualifie alors de **globales**.

7.1 Exemple d'utilisation de variables globales

Voyez cet exemple de programme.

```
#include <stdio.h>
int i ;
main()
{ void optimist (void) ;
for (i=1 ; i<=5 ; i++)
optimist() ;
}
void optimist(void)
{ printf ("il fait beau %d fois\n", i) ;
}
```

Exemple d'utilisation d'une variable globale

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

La variable **`i`** a été déclaré en dehors de la fonction **`main`**. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à **`i`** des valeurs qui se trouvent utilisées par la fonction **`optimist`**.
Notez qu'ici la fonction **`optmist`** se contente d'utiliser la valeur de **`i`** mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute

FSO.UMPOUJDA.COM

variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsque vous aurez à écrire des fonctions susceptibles de modifier la valeur de certaines variables, il sera beaucoup plus judicieux de prévoir d'en transmettre l'adresse en argument. En effet, dans ce cas, l'appel de la fonction montrera explicitement quelle est la variable qui risque d'être modifiée et, de plus, ce sera la seule qui pourra l'être.

7.2 La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur déclaration (n'oubliez pas que, pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois). Ainsi, voyez, par exemple, ces instructions :

```
main()
{
....
}
int n ;
float x ;
fct1 (...)
{
....
}
fct2 (...)
{
....
}
```

Les variables **n** et **x** sont accessibles aux fonctions **fct1** et **fct2**, mais pas au programme principal. En pratique, bien qu'il soit possible effectivement de déclarer des variables globales à n'importe quel endroit du programme source qui soit extérieur aux fonctions, on procédera rarement ainsi. En effet, pour d'évidentes raisons de lisibilité, on préférera regrouper les déclarations de toutes les variables globales au début du programme source.

7.3 La classe d'allocation des variables globales

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lors de l'édition de liens. On traduit cela en disant qu'elles font partie de la **classe d'allocation statique**.
De plus, ces variables se voient **initialisées à zéro**, avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

FSO.UMPOUJDA.COM

8 Les variables locales

À l'exception de l'exemple du paragraphe précédent, les variables que nous avions rencontrées jusqu'ici n'étaient pas des variables globales. Plus précisément, elles étaient définies au sein d'une fonction (qui pouvait être **`main`**). De telles variables sont dites **locales** à la fonction dans laquelle elles sont déclarées.

8.1 La portée des variables locales

Les variables locales ne sont connues qu'à l'intérieur de la fonction où elles sont déclarées. **Leur portée est donc limitée à cette fonction.**
Les variables locales n'ont aucun lien avec des variables globales de même nom ou avec d'autres variables locales à d'autres fonctions. Voyez cet exemple :

```
int n ;
main()
{
int p ;
....
}
fct1 ()
{
int p ;
int n ;
}
```

La variable **`p`** de **`main`** n'a aucun rapport avec la variable **`p`** de **`fct1`**. De même, la variable **`n`** de **`fct1`** n'a aucun rapport avec la variable globale **`n`**. Notez qu'il est alors impossible, dans la fonction **`fct1`**, d'utiliser cette variable globale **`n`**.

8.2 Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent. Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant. On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »).
D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

FSO.UMPOUJDA.COM

8.3 Les variables locales statiques

Il est toutefois possible de demander d'attribuer un emplacement permanent à une variable locale et qu'ainsi sa valeur se conserve d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot-clé **static** (le mot *static* employé sans indication de type est équivalent à **static int**) . En voici un exemple :

```
#include <stdio.h>
main()
{ void fct(void) ;
int n ;
for ( n=1 ; n<=5 ; n++)
fct() ;
}
void fct(void)
{ static int i ;
i++ ;
printf ("appel numéro : %d\n", i) ;
}
```

Exemple d'utilisation de variable locale statique

```
appel numéro : 1
appel numéro : 2
appel numéro : 3
appel numéro : 4
appel numéro : 5
```

La variable locale **i** a été déclarée de classe « **statique** ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**.
Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée **i** qui n'aurait alors aucun rapport avec la variable **i** de **fct**.

8.4 Le cas des fonctions récursives

Le langage C autorise la récursivité des appels de fonctions. Celle-ci peut prendre deux aspects :

- Récursivité directe: une fonction comporte, dans sa définition, au moins un appel à elle-même,
- Récursivité croisée: l'appel d'une fonction entraîne celui d'une autre fonction qui, à son tour, appelle la fonction initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux fonctions).

Voici un exemple fort classique (d'ailleurs inefficace sur le plan du temps d'exécution) d'une fonction calculant une factorielle de manière récursive :

FSO.UMPOUJDA.COM

```
long fac (int n)
{
if (n>1) return (fac(n-1)*n) ;
else return(1) ;
}
```

Fonction récursive de calcul de factorielle

Il faut bien voir qu'alors chaque appel de **fac** entraîne une allocation d'espace pour les variables locales et pour son argument **n** (apparemment, **fct** ne comporte aucune variable locale ; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de **fac**, à l'intérieur de **fac**, provoque une telle allocation, sans que les emplacements précédents soient libérés. Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction **return** que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

9 Utilisation de pointeurs sur des fonctions

En C, comme dans la plupart des autres langages, il n'est pas possible de placer le nom d'une fonction dans une variable. En revanche, on peut y définir une variable destinée à pointer sur une fonction, c'est-à-dire à contenir son adresse.
De plus, en C, le nom d'une fonction (employé seul) est traduit par le compilateur en l'adresse de cette fonction. On retrouve là quelque chose d'analogue à ce qui se passait pour les noms de tableaux, avec toutefois cette différence que les noms de fonctions sont externes (ils subsisteront dans les modules objet).
Ces deux remarques offrent en C des possibilités intéressantes. En voici deux exemples.

9.1 Paramétrage d'appel de fonctions

Considérez cette déclaration : **int (*adf) (double, int) ;**
Elle spécifie que : **(*adf)** est une fonction à deux arguments (de type **double** et **int**) fournissant un résultat de type **int**,

Donc que : **adf** est un pointeur sur une fonction à deux arguments (**double** et **int**) fournissant un résultat de type **int**.
Si, par exemple, **fct1** et **fct2** sont des fonctions ayant les prototypes suivants :

```
int fct1 (double, int) ;
int fct2 (double, int) ;
```

les affectations suivantes ont alors un sens :

```
adf = fct1 ;
adf = fct2 ;
```

Elles placent, dans **adf**, l'adresse de la fonction correspondante (**fct1** ou **fct2**).
Dans ces conditions, il devient possible de programmer un « appel de fonction variable » (c'est-à-dire que la fonction appelée peut varier au fil de l'exécution du programme) par une instruction telle que : **(*adf) (5.35, 4) ;**

FSO.UMPOUJDA.COM

Celle-ci, en effet, appelle la fonction dont l'adresse figure actuellement dans **adf**, en lui transmettant les valeurs indiquées **(5.35 et 4).** Suivant le cas, cette instruction sera donc équivalente à l'une des deux suivantes :

```
fct1 (5.35, 4) ;
fct2 (5.35, 4) ;
```

9.2 Fonctions transmises en argument

Supposez que nous souhaitions écrire une fonction permettant de calculer l'intégrale d'une fonction quelconque suivant une méthode numérique donnée. Une telle fonction que nous supposerons nommée **integ** posséderait alors un en-tête de ce genre : **float integ (float(*f)(float), ...)**

Le premier argument muet correspond ici à l'adresse de la fonction dont on cherche à calculer l'intégrale. Sa déclaration peut s'interpréter ainsi :

(*f)(float) est de type **float**, **(*f)** est donc une fonction recevant un argument de type **float** et fournissant un résultat de type **float**, **f** est donc un pointeur sur une fonction recevant un argument de type **float** et fournissant un résultat de type **float**.

Au sein de la définition de la fonction **integ**, il sera possible d'appeler la fonction dont on aura ainsi reçu l'adresse de la façon suivante : **(*f) (x)**

Notez bien qu'il ne faut surtout pas écrire **f(x)**, car **f** désigne ici un pointeur contenant l'adresse d'une fonction, et non pas directement l'adresse d'une fonction.

L'utilisation de la fonction **integ** ne présente pas de difficultés particulières. Elle pourrait se présenter ainsi :

```
main()
{
float fct1(float), fct2(float) ;
.....
res1 = integ (fct1, .....) ;
.....
res2 = integ (fct2, .....) ;
.....
}
```

10 Les différents types de variables, leur portée et leur classe d'allocation

Nous avons déjà vu différentes choses concernant les classes d'allocation des variables et leur portée. Ici, nous nous proposons de faire le point après avoir introduit quelques informations supplémentaires.

10.1 La portée des variables

On peut classer les variables en quatre catégories en fonction de leur portée (ou espace de validité). Nous avons déjà rencontré les trois premières que sont : les variables globales, les variables globales cachées et les variables locales à une fonction. En outre, il est possible de définir des **variables locales à un bloc**. Elles se

FSO UMPOUJDA.COM

déclarent en début d'un bloc de la même façon qu'en début d'une fonction. Dans ce cas, la portée de telles variables est limitée au bloc en question. Leur usage est, en pratique, peu répandu.

10.2 Les classes d'allocation des variables

Il est également possible de classer les variables en trois catégories en fonction de leur classe d'allocation. Là encore, nous avons déjà rencontré les deux premières, à savoir:

- **La classe statique** : elle renferme non seulement les variables globales (quelles qu'elles soient), mais aussi les variables locales faisant l'objet d'une déclaration **`static`**. Les emplacements mémoire correspondants sont alloués une fois pour toutes au moment de l'édition de liens,

- **La classe automatique** : par défaut, les variables locales entrent dans cette catégorie. Les emplacements mémoire correspondants sont alloués à chaque entrée dans la fonction où sont définies ces variables et ils sont libérés à chaque sortie.

En toute rigueur, il existe une classe un peu particulière, à savoir la **classe registre :** toute variable entrant a priori dans la classe automatique peut être déclarée explicitement avec le qualificatif **register**. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un registre pour y ranger la variable ; cela peut amener quelques gains de temps d'exécution.
Bien entendu, cette possibilité ne peut s'appliquer qu'aux variables scalaires.

Remarque :

L**e cas des fonctions.** La fonction est considérée par le langage C comme un objet global. C'est ce qui permet d'ailleurs à l'éditeur de liens d'effectuer correctement son travail. Il faut noter toutefois qu'il n'est pas nécessaire d'utiliser une déclaration **extern** pour les fonctions définies dans un fichier source différent de celui où elles sont appelées (mais le faire ne constitue pas une erreur).
En tant qu'objet global, la fonction peut voir sa portée limitée au fichier source dans lequel elle est définie à l'aide d'une déclaration **static** comme dans : **static int fct (...)**

10.3 Tableau récapitulatif

Voici un tableau récapitulant la portée et la classe d'allocation des différentes variables suivant la nature de leur déclaration (la colonne « Type » donne le nom qu'on attribue usuellement à de telles variables).

FSO UMPOUJDA.COM

Type de variable	Déclaration	Portée	Classe d'allocation
Globale	en dehors de toute fonction	-la partie du fichier source suivant sa déclaration, -n'importe quel fichier source, avec **`extern`**.	**Statique**
Globale cachée	en dehors de toute **Statique** fonction, avec l'attribut **`static`**	uniquement la partie du fichier source suivant sa déclaration	
Locale «rémanente »	au début d'une fonction, avec l'attribut **`static`**	la fonction	
Locale à une fonction	au début d'une fonction	la fonction	**Automatique**
Locale à un bloc	au début d'un bloc	le bloc	

Type, portée et classe d'allocation des variables

11 Initialisation des variables

Nous avons vu qu'il était possible d'initialiser explicitement une variable lors de sa déclaration. Ici, nous allons faire le point sur ces possibilités, lesquelles dépendent en fait de la classe d'allocation de la variable concernée.

11.1 Les variables de classe statique

Ces variables sont permanentes. Elles sont initialisées une seule fois avant le début de l'exécution du programme.
Elles peuvent être initialisées explicitement lors de leur déclaration. Nous verrons que cela s'applique également aux tableaux ou aux structures. Bien entendu, les valeurs servant à ces initialisations ne pourront être que des constantes ou des expressions constantes (c'est-à-dire calculables par le compilateur). N'oubliez pas que les constantes symboliques définies par **`const`** ne sont pas considérées comme des expressions constantes. En l'absence d'initialisation explicite, ces variables seront initialisées à zéro.

11.2 Les variables de classe automatique

Ces variables ne sont pas initialisées par défaut. En revanche, comme les variables de classe statique, elles peuvent être initialisées explicitement lors de leur déclaration.
Dans ce cas, lorsqu'il s'agit de variables scalaires (ce qui est le cas de toutes celles rencontrées jusqu'ici, mais ne sera pas le cas des tableaux ou des structures), la norme vous autorise à utiliser comme valeur initiale non seulement une valeur constante, mais également n'importe quelle expression (dans la mesure où sa valeur est définie

FSO.UMPOUJDA.COM

au moment de l'entrée dans la fonction correspondante, laquelle, ne l'oubliez pas, peut être la fonction **main**).
En voici un exemple :

```
#include <stdio.h>
int n ;
main()
{ void fct (int r) ;
int p ;
for (p=1 ; p<=5 ; p++)
{ n = 2*p ;
fct(p) ;
}
}
void fct(int r)
{
int q=n, s=r*n ;
printf ("%d %d %d\n", r,q,s) ;
}
```

Initialisation de variables de classe automatique

N'oubliez pas que ces variables automatiques se trouvent alors initialisées à chaque appel de la fonction dans laquelle elles sont définies.

12 Les arguments variables en nombre

Dans tous nos précédents exemples, le nombre d'arguments fournis au cours de l'appel d'une fonction était prévu lors de l'écriture de cette fonction.
Or, dans certaines circonstances, on peut souhaiter réaliser une fonction capable de recevoir un nombre d'arguments susceptible de varier d'un appel à un autre. C'est d'ailleurs ce que nous faisons (sans même y penser) lorsque nous utilisons des fonctions comme **printf** ou **scanf** (en dehors du premier argument, qui représente le format, les autres sont en nombre quelconque)
Le langage C permet de résoudre ce problème à l'aide des fonctions particulières **va_start** et **va_arg**. La seule contrainte à respecter est que la fonction doit posséder certains arguments fixes (c'est-à-dire toujours présents), leur nombre ne pouvant être inférieur à un. En effet, comme nous allons le voir, c'est le dernier argument fixe qui permet, en quelque sorte, d'initialiser le parcours de la liste d'arguments.

12.1 Premier exemple

Voici un premier exemple de fonction à arguments variables : les deux premiers arguments sont fixes, l'un étant de type **int**, l'autre de type **char**. Les arguments suivants, de type **int**, sont en nombre quelconque et l'on a supposé que le dernier d'entre eux était **-1.** Cette dernière valeur sert donc, en quelque sorte, de sentinelle.
Par souci de simplification, nous nous sommes contentés, dans cette fonction, de lister les valeurs de ces différents arguments (fixes ou variables), à l'exception du dernier.

FSO.UMPOUJDA.COM

```
#include <stdio.h>
#include <stdarg.h>
void essai (int par1, char par2, ...)
{
va_list adpar ;
int parv ;
printf ("premier paramètre : %d\n", par1) ;
printf ("second paramètre : %c\n", par2) ;
va_start (adpar, par2) ;
while ( (parv = va_arg (adpar, int) ) != -1)
printf ("argument variable : %d\n", parv) ;
}
main()
{
printf ("premier essai\n") ;
essai (125, 'a', 15, 30, 40, -1) ;
printf ("\ndeuxième essai\n") ;
essai (6264, 'S', -1) ;
}
```

Arguments en nombre variable délimités par une sentinelle

```
premier essai
premier paramètre : 125
second paramètre : a
argument variable : 15
argument variable : 30
argument variable : 40
deuxième essai
premier paramètre : 6264
second paramètre : S
```

Vous constatez la présence, dans l'en-tête, des deux noms des paramètres fixes **par1** et **par2**, déclarés de manière classique ; les trois points servent à spécifier au compilateur l'existence de paramètres en nombre variable.
La déclaration : **va_list adpar** précise que **adpar** est un identificateur de liste variable. C'est lui qui nous servira à récupérer, les uns après les autres, les différents arguments variables.
Comme à l'accoutumée, une telle déclaration n'attribue aucune valeur à **adpar**. C'est effectivement la fonction **va_start** qui va permettre de l'initialiser à l'adresse du paramètre variable.
Notez bien que cette dernière est déterminée par **va_start** à partir de la connaissance du nom du dernier paramètre fixe.
Le rôle de la fonction **va_arg** est double :

- D'une part, elle fournit comme résultat la valeur trouvée à l'adresse courante fournie par **adpar** (son premier argument), suivant le type indiqué par son second argument (ici **int**).
- D'autre part, elle incrémente l'adresse contenue dans **adpar**, de manière que celle-ci pointe alors sur l'argument variable suivant.

FSO.UMPOUJDA.COM

Ici, une instruction **while** nous permet de récupérer les différents arguments variables, sachant que le dernier a pour valeur **-1**.
Enfin, la norme ANSI prévoit que la macro **va_end** doit être appelée avant de sortir de la fonction concernée. Si vous manquez à cette règle, vous courrez le risque de voir un prochain appel à la fonction conduire à un mauvais fonctionnement du programme.

12.2 Second exemple

La gestion de la fin de la liste des arguments variables est laissée au bon soin de l'utilisateur ; en effet, il n'existe aucune fonction permettant de connaître le nombre effectif de ces arguments. Cette gestion peut se faire :

- Par sentinelle, comme dans notre précédent exemple,
- Par transmission, en argument fixe, du nombre d'arguments variables.

Voici un exemple de fonction utilisant cette seconde technique. Nous n'y avons pas prévu d'autres arguments fixes que celui spécifiant le nombre d'arguments variables.

```
#include <stdio.h>
#include <stdarg.h>
void essai (int nbpar, ...)
{ va_list adpar ;
int parv, i ;
printf ("nombre de valeurs : %d\n", nbpar) ;
va_start (adpar, nbpar) ;
for (i=1 ; i<=nbpar ; i++)
{ parv = va_arg (adpar, int) ;
printf ("argument variable : %d\n", parv) ;
}
}
main()
{ printf ("premier essai\n") ;
essai (3, 15, 30, 40) ;
printf ("\ndeuxième essai\n") ;
essai (0) ;
}
```

Arguments variables dont le nombre est fourni en argument fixe

```
premier essai
nombre de valeurs : 3
argument variable : 15
argument variable : 30
argument variable : 40
deuxième essai
nombre de valeurs : 0
```

FSO.UMPOUJDA.COM

Chapitre 5 : Les fichiers

Le terme de fichier désigne plutôt un ensemble d'informations situé sur une «mémoire de masse » telle que le disque ou la disquette. Nous verrons toutefois qu'en C, comme d'ailleurs dans d'autres langages, tous les périphériques, qu'ils soient d'archivage (disque, disquette...) ou de communication (clavier, écran, imprimante...), peuvent être considérés comme des fichiers. Ainsi, en définitive, les **entrées-sorties conversationnelles apparaîtront comme un cas particulier de la gestion de fichiers.**
Rappelons que l'on distingue traditionnellement deux techniques de gestion de fichiers :

- Accès séquentiel consiste à traiter les informations séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent (ou apparaîtront) dans le fichier ;
- Accès direct consiste à se placer immédiatement sur l'information souhaitée, sans avoir à parcourir celles qui la précèdent.

1 Création séquentielle d'un fichier

Voici un programme qui se contente d'enregistrer séquentiellement dans un fichier une suite de nombres entiers saisis au clavier.

```
#include <stdio.h>
main()
{
char nomfich[21] ;
int n ;
FILE * sortie ;
printf ("nom du fichier à créer : ") ;
scanf ("%20s", nomfich) ;
sortie = fopen (nomfich, "w") ;
do { printf ("donnez un entier : ") ;
scanf ("%d", &n) ;
if (n) fwrite (&n, sizeof(int), 1, sortie) ;
}
while (n) ;
fclose (sortie) ;
}
```

Création séquentielle d'un fichier d'entiers

Nous avons déclaré un tableau de caractères **nomfich** destiné à contenir, sous forme d'une chaîne, le nom du fichier que l'on souhaite créer. La déclaration :

```
FILE * sortie ;
```

signifie que **sortie** est un pointeur sur un objet de type **FILE**. Ce nom désigne en fait un modèle de structure défini dans le fichier **stdio.h** (par une instruction **typedef**, ce qui explique l'absence du mot **struct**).
N'oubliez pas que cette déclaration ne réserve qu'un emplacement pour un pointeur. C'est la fonction **fopen** qui créera effectivement une telle structure et qui en fournira l'adresse en résultat.

FSO.UMPOUJDA.COM

La fonction **fopen** est ce que l'on nomme une fonction d'ouverture de fichier. Elle possède deux arguments :

- Le nom du fichier concerné, fourni sous forme d'une chaîne de caractères ; ici, nous avons prévu que ce nom ne dépassera pas **20** caractères (le chiffre 21 tenant compte du caractère **\0**) ; notez qu'en général ce nom pourra comporter une information (chemin, répertoire...) permettant de préciser l'endroit où se trouve le fichier.
- Une indication, fournie elle aussi sous forme d'une chaîne, précisant ce que l'on souhaite faire avec ce fichier. Ici, on trouve **w** (abréviation de **write**) qui permet de réaliser une ouverture en écriture. Plus précisément, si le fichier cité n'existe pas, il sera créé par **fopen**. S'il existe déjà, son ancien contenu deviendra inaccessible. Autrement dit, après l'appel de cette fonction, on se retrouve dans tous les cas en présence d'un fichier vide.

Le remplissage du fichier est réalisé par la répétition de l'appel :

fwrite (&n, sizeof(int), 1, sortie) ;

La fonction **fwrite** possède quatre arguments précisant :

- l'adresse d'un bloc d'informations (ici **&n**) ;
- la taille d'un bloc, en octets : ici **sizeof(int)** ; notez l'emploi de l'opérateur **sizeof** qui assure la portabilité du programme ;
- le nombre de blocs de cette taille que l'on souhaite transférer dans le fichier (ici **1**) ;
- l'adresse de la structure décrivant le fichier (**sortie**).

Notez que, d'une manière générale, **fwrite** permet de transférer plusieurs blocs consécutifs de même taille à partir d'une adresse donnée.
Enfin, la fonction **fclose** réalise ce que l'on nomme une fermeture de fichier. Elle force l'écriture sur disque du tampon associé au fichier. En effet, chaque appel à **fwrite** provoque un entassement d'informations dans le tampon associé au fichier. Ce n'est que lorsque ce dernier est plein qu'il est « vidé » sur disque. Dans ces conditions, on voit qu'après le dernier appel de **fwrite** il est nécessaire de forcer le transfert des dernières informations accumulées dans le tampon.

2 Liste séquentielle d'un fichier

Voici maintenant un programme qui permet de lister le contenu d'un fichier quelconque tel qu'il a pu être créé par le programme précédent.

```
#include <stdio.h>
main()
{
char nomfich[21] ;
int n ;
FILE * entree ;
printf ("nom du fichier à lister : ") ;
```


FSO.UMPOUJDA.COM

```
scanf ("%20s", nomfich) ;
entree = fopen (nomfich, "r") ;
while ( fread (&n, sizeof(int), 1, entree), !
feof(entree) )
printf ("\n%d", n) ;
fclose (entree) ;
}
```

Liste séquentielle d'un fichier

Les déclarations sont identiques à celles du programme précédent. En revanche, on trouve cette fois, dans l'ouverture du fichier, l'indication **r** (abréviation de **read**). Elle précise que le fichier en question ne sera utilisé qu'en lecture. Il est donc nécessaire qu'il existe déjà (nous verrons un peu plus loin comment traiter convenablement le cas où il n'existe pas).

La lecture dans le fichier se fait par un appel de la fonction **fread** :

fread (&n, sizeof(int), 1, entree)

dont les arguments sont comparables à ceux de **fwrite**. Mais, cette fois, la condition d'arrêt de la boucle est : **feof (entree)**
Celle-ci prend la valeur vrai (c'est-à-dire **1**) lorsque la fin du fichier a été rencontrée. Notez bien qu'il n'est pas suffisant d'avoir lu le dernier octet du fichier pour que cette condition prenne la valeur vraie. Il est nécessaire d'avoir **tenté de lire au-delà** (contrairement à ce qui se passe, par exemple, en Turbo Pascal, dans lequel la détection de fin de fichier fonctionne en quelque sorte par anticipation) ; c'est ce qui explique que nous ayons examiné cette condition après l'appel de **fread** et non avant.

Remarque :

On pourrait remplacer la boucle **while** par la construction (moins concise) suivante :

```
do
{ fread (&n, sizeof(int), 1, entree) ;
if ( !feof(entree) ) printf ("\n%d", n) ;
}
while ( !feof(entree) ) ;
```

N'oubliez pas que le premier argument des fonctions **fwrite** et **fread** est une adresse. Ainsi, lorsque vous aurez affaire à un tableau, il faudra utiliser simplement son nom (sans le faire précéder de **&),** tandis qu'avec une structure il faudra effectivement utiliser l'opérateur **&** pour en obtenir l'adresse. Dans ce dernier cas, même si l'on ne cherche pas à rendre son programme portable, il sera préférable d'utiliser l'opérateur **sizeof** pour déterminer avec certitude la taille des blocs correspondants.
fread fournit le nombre de blocs effectivement lus (et non pas le nombre d'octets lus). Ce résultat peut être inférieur au nombre de blocs demandés soit lorsque l'on a rencontré une fin de fichier, soit lorsqu'une erreur de lecture est apparue. Dans notre précédent exemple d'exécution, **fread** fournit toujours **1**, sauf la dernière fois où elle fournit **0**.

FSO.UMPOUJDA.COM

3 L'accès direct

Les fonctions **fread** et **fwrite** lisent ou écrivent un certain nombre d'octets dans un fichier, à partir d'une position courante. Cette dernière n'est rien d'autre qu'un «pointeur » dans le fichier, c'est-à-dire un nombre précisant le rang du prochain octet à lire ou à écrire. (Le terme de « pointeur » n'a pas exactement le même sens que celui de pointeur tel qu'il apparaît en langage C. En effet, il ne désigne pas, à proprement parler, une adresse en mémoire, mais un emplacement dans un fichier. Pour éviter des confusions, nous parlerons de « pointeur de fichier »). Après chaque opération de lecture ou d'écriture, ce pointeur se trouve incrémenté du nombre d'octets transférés. C'est ainsi que l'on réalise un accès séquentiel au fichier.
Mais il est également possible d'agir directement sur ce pointeur de fichier à l'aide de la fonction **fseek**. Cela permet ainsi de réaliser des lectures ou des écritures en n'importe quel point du fichier, sans avoir besoin de parcourir toutes les informations qui précèdent. On peut ainsi réaliser ce que l'on nomme généralement un « accès direct ».

3.1 Accès direct en lecture sur un fichier existant

Exemple :

```
#include <stdio.h>
main()
{ char nomfich[21] ;
int n ;
long num ;
FILE * entree ;
printf ("nom du fichier à consulter : ") ;
scanf ("%20s", nomfich) ;
entree = fopen (nomfich, "r") ;
while ( printf (" numéro de l'entier recherché : "),
scanf ("%ld", &num), num )
{ fseek (entree, sizeof(int)*(num-1), SEEK_SET) ;
fread (&n, sizeof(int), 1, entree) ;
printf (" valeur : %d \n", n) ;
}
fclose (entree) ;
```

Accès direct en lecture sur un fichier existant

Le programme ci-dessus permet d'accéder à n'importe quel entier d'un fichier.
La principale nouveauté réside essentiellement dans l'appel de la fonction `fseek` :
`fseek ( entree, sizeof(int)*(num-1), SEEK_SET) ;`
Cette dernière possède trois arguments :

- Le fichier concerné (désigné par le pointeur sur une structure de type **FILE**, tel qu'il a été fourni par **fopen**) ;
- Un entier de type **long** spécifiant la valeur que l'on souhaite donner au pointeur de fichier. Il faut noter que l'on dispose de trois manières d'agir

effectivement sur le pointeur, le choix entre les trois étant fait par l'argument suivant ;

- Le choix du mode d'action sur le pointeur de fichier : il est défini par une constante entière. Les valeurs suivantes sont prédéfinies dans <**stdio.h**> :
  - ✓ **SEEK_SET** (en général **0**) : le second argument désigne un déplacement (en octets) depuis le **début du fichier** ;
  - ✓ **SEEK_CUR** (en général **1**) : le second argument désigne un déplacement exprimé à partir de la **position courante** ; il s'agit donc en quelque sorte d'un déplacement relatif dont la valeur peut, le cas échéant, être négative ;
  - ✓ **SEEK_END** (en général **2**) : le second argument désigne un déplacement depuis la **fin du fichier**.

Ici, il s'agit de donner au pointeur de fichier une valeur correspondant à l'emplacement d'un entier (**sizeof(int)** octets) dont l'utilisateur fournit le rang. Il est donc naturel de donner au troisième argument la valeur **0**. Notez, au passage, la formule : **sizeof(int) * (num-1)** qui se justifie par le fait que nous avons convenu que, pour l'utilisateur, le premier entier du fichier porterait le rang **1** et non **0**.

3.2 Les possibilités de l'accès direct

L'accès direct facilite et accélère les opérations de mise à jour d'un fichier. Il permet de remplir un fichier de façon quelconque. Ainsi, nous pourrions constituer notre fichier d'entiers en laissant l'utilisateur fournir ces entiers dans l'ordre de son choix, comme dans cet exemple de programme :

```
#include <stdio.h>
main()
{ char nomfich[21] ;
FILE * sortie ;
long num ;
int n ;
printf ("nom fichier : ") ;
scanf ("%20s",nomfich) ;
sortie = fopen (nomfich, "w") ;
while (printf("\nrang de l'entier : "),
scanf("%ld",&num), num)
{ printf ("valeur de l'entier : ") ;
scanf ("%d", &n) ;
fseek (sortie, sizeof(int)*(num-1), SEEK_SET) ;
fwrite (&n, sizeof(int), 1, sortie) ;
}
fclose(sortie) ;
```

Création d'un fichier en accès direct

FSO.UMPOUJDA.COM

4 Les entrées-sorties formatées et les fichiers de texte

Les fichiers concernés par les opérations de formatage sont alors ce qu'on appele des « fichiers de type texte » ou encore des « fichiers de texte ». Ce sont des fichiers que vous pouvez manipuler avec un éditeur ou un traitement de texte quelconques ou, encore plus simplement, lister par les commandes appropriées du système d'exploitation (**`TYPE`** ou **`PRINT`** sous **DOS**, **`pr`** ou **`more`** sous **UNIX**...). Dans de tels fichiers, chaque octet représente un caractère. Généralement, on y trouve des caractères de fin de ligne (`\n`), de sorte qu'ils apparaissent comme une suite de lignes. Les fonctions permettant de travailler avec des fichiers de texte ne sont rien d'autre qu'une généralisation de celles que nous avons déjà rencontrées pour les entrées-sorties conversationnelles.

Nous nous contenterons donc d'en fournir une brève liste :
fscanf (fichier, format, liste_d'adresses)
fprintf (fichier, format, liste_d'expressions)
fgetc (fichier)
fputc (entier, fichier)
fgets (chaîne, lgmax, fichier)
fputs (chaîne, fichier)

La signification de leurs arguments est la même que pour les fonctions conversationnelles correspondantes. Seule **`fgets`** comporte un argument entier (**`lgmax`**) de contrôle de longueur. Il précise le nombre maximal de caractères (y compris le **`\0`** de fin) qui seront placés dans la chaîne.
Leur valeur de retour est la même que pour les fonctions conversationnelles. Cependant, il nous faut apporter quelques indications supplémentaires qui ne se justifiaient pas pour des entrées-sorties conversationnelles (mais qui auraient un intérêt en cas de simple « redirection » des entrées-sorties), à savoir que la valeur de retour fournie par **`fgetc`** est du type **`int`** (et non, comme on pourrait le croire, de type **`char`**). Lorsque la fin de fichier est atteinte, cette fonction fournit la valeur **`EOF`** (constante prédéfinie dans **`<stdio.h>`** – en général **`-1`**). La fin de fichier n'est détectée que lorsque l'on cherche à lire un caractère alors qu'il n'y en a plus de disponible, et non pas, dès que l'on a lu le dernier caractère (ce qui, ici, serait assez mal approprié puisque alors on ne pourrait obtenir à la fois le code de ce caractère et une indication de fin de fichier). D'autre part, notez bien que cette convention fait, en quelque sorte, double emploi avec la fonction **`feof`**.
D'une manière générale, toutes les fonctions présentées ci-dessus fournissent une valeur de retour bien définie en cas de fin de fichier ou d'erreur.

5 Les différentes possibilités d'ouverture d'un fichier

Dans nos précédents exemples, nous n'avons utilisé que les modes **`w`** et **`r`**. Nous vous fournissons ici la liste des différentes possibilités offertes par **`fopen`**.
`r` : ***lecture*** seulement ; le fichier doit exister.

w : ***écriture*** seulement. Si le fichier n'existe pas, il est créé. S'il existe, son (ancien) contenu est perdu.
a : ***écriture en fin de fichier*** (append). Si le fichier existe déjà, il sera étendu. S'il n'existe pas, il sera créé – on se ramène alors au mode **w**.
r+ : ***mise à jour*** (lecture et écriture). Le fichier doit exister. Notez qu'alors il n'est pas possible de réaliser une lecture à la suite d'une écriture ou une écriture à la suite d'une lecture, sans positionner le pointeur de fichier par **fseek**. Il est toutefois possible d'enchaîner plusieurs lectures ou écritures consécutives (de façon séquentielle).
w+ : ***création pour mise à jour***. Si le fichier existe, son (ancien) contenu sera détruit. S'il n'existe pas, il sera créé. Notez que l'on obtiendrait un mode comparable à **w+** en ouvrant un fichier vide (mais existant) en mode **r+**.
a+ : ***extension et mise à jour***. Si le fichier n'existe pas, il sera créé. S'il existe, le pointeur sera positionné en fin de fichier.
t ou **b** : lorsque l'implémentation distingue les fichiers de texte des autres, il est possible d'ajouter l'une de ces deux lettres à chacun des 6 modes précédents. La lettre **t** précise que l'on a affaire à un fichier de texte ; la lettre **b** précise que l'on a affaire à un fichier binaire. (On dit aussi que **t** correspond au mode « translaté », pour spécifier que certaines substitutions auront lieu).

6 Les fichiers prédéfinis

Un certain nombre de fichiers sont connus du langage C, sans qu'il soit nécessaire ni de les ouvrir ni de les fermer :

- **stdin :** unité d'entrée (par défaut, le clavier) ;
- **stdout** : unité de sortie (par défaut, l'écran) ;
- **stderr :** unité d'affichage des messages d'erreurs (par défaut, l'écran).

On trouve parfois également :

- **stdaux** : unité auxiliaire ;
- **stdprt :** imprimante

Les deux premiers fichiers correspondent aux unités standard d'entrée et de sortie d'un programme. Lorsque vous exécutez un programme depuis le système, vous pouvez éventuellement rediriger ces fichiers. Par exemple, la commande système suivante (valable à la fois sous **UNIX** et sous **DOS**) :

```
TRUC <DONNEES >RESULTATS
```

exécute le programme **TRUC**, en utilisant comme unité d'entrée le fichier **DONNEES** et comme unité de sortie le fichier **RESULTATS**.
Dans ces conditions, une instruction telle que, par exemple, **fgetchar** deviendrait équivalente à **fgetc(fich)** où **fich** serait un flux obtenu par appel à **fopen**. De même, **scanf(...)** deviendrait équivalent à **fscanf(fich, ..)**, etc.
Notez bien qu'au sein du programme même il n'est pas possible de savoir si un fichier prédéfini a été redirigé au moment du lancement du programme ; autrement dit, lorsqu'une fonction comme **fgetchar** ou **scanf** lit des informations, elle ne peut absolument pas savoir si ces dernières proviennent du clavier ou d'un fichier.

FSO UMPOUJDA.COM

Chapitre 6 : La compilation séparée et directives du processeur

En C, il est possible de compiler séparément plusieurs programmes (fichiers) source et de rassembler les modules objet correspondants au moment de l'édition de liens. D'ailleurs, dans certains environnements de programmation, la notion de *projet* permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable.
Cette notion de projet fait intervenir précisément les fichiers à considérer ; généralement, il est possible de demander de créer le programme exécutable, en ne recompilant que les sources ayant subi une modification depuis leur dernière compilation.
Notez que, à partir du moment où l'on parle de compilation séparée, il existe au moins (ou il a existé) deux programmes source ; dans la suite, nous supposerons qu'ils figurent dans des fichiers, de sorte que nous parlerons toujours de fichier source.

1 La portée d'une variable globale - la déclaration extern

A priori, la portée d'une variable globale semble limitée au fichier source dans lequel elle a été définie. Ainsi, supposez que l'on compile séparément ces deux fichiers source :

source 1	source 2

`int x ;`
`main()`
`{`
`.....`
`}`
`fct1()`
`{`
`.....`
`}`
	`fct2()`
	`{`
	`.....`
	`}`
	`fct3()`
	`{`
	`.....`
	`}`

Il ne semble pas possible, dans les fonctions **`fct2`** et **`fct3`** de faire référence à la variable globale **`x`** déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).
En fait, le langage C prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé **`extern`**. Ainsi, en faisant précéder notre second fichier source de la déclaration :
`extern int x ;`
Il devient possible de mentionner la variable globale **`x`** (déclarée dans le premier fichier source) dans les fonctions **`fct2`** et **`fct3`**.

FSO.UMPOUJDA.COM

2 Les variables globales et l'édition de liens

Supposons que nous ayons compilé les deux fichiers source précédents et voyons d'un peu plus près comment l'éditeur de liens est en mesure de rassembler correctement les deux modules objet ainsi obtenus. En particulier, examinons comment il peut faire correspondre au symbole **x** du second fichier source l'adresse effective de la variable **x** définie dans le premier.

D'une part, après compilation du premier fichier source, on trouve, dans le module objet correspondant, une indication associant le symbole **x** et son adresse dans le module objet. Autrement dit, contrairement à ce qui se produit pour les variables locales, pour lesquelles ne subsiste aucune trace du nom après compilation, le nom des variables globales continue à exister au niveau des modules objet. On retrouve là un mécanisme analogue à ce qui se passe pour les noms de fonctions, lesquels doivent bien subsister pour que l'éditeur de liens soit en mesure de retrouver les modules objet correspondants.

D'autre part, après compilation du second fichier source, on trouve dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom **x** provient de l'extérieur et qu'il faudra en fournir l'adresse effective.

Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable **x** et de la reporter dans le second module objet.

Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objet correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

3 Les variables globales cachées- la déclaration static

Il est possible de « cacher » une variable globale, c'est-à-dire de la rendre inaccessible à l'extérieur du fichier source où elle a été définie (on dit aussi « rendre confidentielle» au lieu de « cacher » ; on parle alors de « variables globales confidentielles »). Il suffit pour cela d'utiliser la déclaration **static** comme dans cet exemple :

```
static int a ;
main()
{
.....
}
fct()
{
.....
}
```

FSO.UMPOUJDA.COM

Sans la déclaration **static**, **a** serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de **a** ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par **extern**.
Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle de la première source.
Cette possibilité de cacher des variables globales peut s'avérer précieuse lorsque vous êtes amené à développer un ensemble de fonctions d'intérêt général qui doivent partager des variables globales. En effet, elle permet à l'utilisateur éventuel de ces fonctions de ne pas avoir à se soucier des noms de ces variables globales puisqu'il ne risque pas alors d'y accéder par mégarde. Cela généralise en quelque sorte à tout un fichier source la notion de variable locale telle qu'elle était définie pour les fonctions.
Ce sont d'ailleurs de telles possibilités qui permettent de développer des logiciels en C, en utilisant une philosophie orientée objet.

4 Le préprocesseur

Le « préprocesseur ». Il s'agit d'un programme qui est exécuté automatiquement avant la compilation et qui transforme votre fichier source à partir d'un certain nombre de directives. Ces dernières, contrairement à ce qui se produit pour les instructions du langage C, sont écrites sur des lignes distinctes du reste du programme; elles sont toujours introduites par un mot précis commençant par le caractère #. Parmi ces directives, nous avons déjà utilisé **#include** et **#define**.
Nous nous proposons ici d'étudier les diverses possibilités offertes par le préprocesseur, à savoir :

- l'incorporation de fichiers source (directive **#include**) ;
- la définition de symboles et de macros (directive **#define**) ;
- la compilation conditionnelle.

4.1 La directive #include

Elle permet d'incorporer, avant compilation, le texte figurant dans un fichier quelconque. Jusqu'ici, nous n'avions incorporé de cette manière que les contenus de fichiers en-tête requis pour le bon usage des fonctions standard. Mais cette directive peut s'appliquer également à des fichiers de votre cru. Cela peut s'avérer utile, par exemple pour écrire une seule fois les déclarations communes à plusieurs fichiers source différents, en particulier dans le cas des prototypes de fonctions.
Rappelons que cette directive possède deux syntaxes voisines :

#include <nom_fichier> recherche le fichier mentionné dans un emplacement (chemin, répertoire) défini par l'implémentation.

#include "nom_fichier" recherche le fichier mentionné dans le même emplacement (chemin, répertoire) que celui où se trouve le programme source.
Généralement, la première est utilisée pour les fichiers en-tête correspondant à la bibliothèque standard, tandis que la seconde l'est plutôt pour les fichiers que vous créez vous-même.

FSO.UMPOUJDA.COM

4.2 La directive #define

Elle offre en fait deux possibilités :

- définition de symboles (c'est sous cette forme que nous l'avons employée jusqu'ici) ;
- définition de macros.

4.2.1 Définition de symboles

Une directive telle que : **#define nbmax 5** demande de substituer au symbole **nbmax** le texte **5**, et cela chaque fois que ce symbole apparaîtra dans la suite du fichier source.
Une directive : **#define entier int** placée en début de programme, permettra d'écrire en français les déclarations de variables entières. Ainsi, par exemple, ces instructions :

```
entier a, b ;
entier * p ;
```

seront remplacées par :

```
int a, b ;
int * p ;
```

Il est possible de demander de faire apparaître dans le texte de substitution un symbole déjà défini. Par exemple, avec ces directives :

```
#define nbmax 5
....
#define taille nbmax + 1
```

Chaque mot **taille** apparaissant dans la suite du programme sera systématiquement remplacé par **5+1**. Notez bien que **taille** ne sera pas remplacé exactement par **6** mais, étant donné que le compilateur accepte les expressions constantes là où les constantes sont autorisées, le résultat sera comparable (après compilation). Il est même possible de demander de substituer à un symbole un texte vide. Par exemple, avec cette directive : **#define rien** tous les symboles `rien` figurant dans la suite du programme seront remplacés par un texte vide. Tout se passera donc comme s'ils ne figuraient pas dans le programme.
Nous verrons qu'une telle possibilité n'est pas aussi fantaisiste qu'il y paraît au premier abord puisqu'elle pourra intervenir dans la compilation conditionnelle.

Voici quelques derniers exemples vous montrant comment résumer en un seul mot une instruction C :

```
#define bonjour printf("bonjour")
#define affiche printf("resultat %d\n", a)
#define ligne printf("\n")
```

Notez que nous aurions pu inclure le point-virgule de fin dans le texte de substitution.

FSO.UMPOUJDA.COM

D'une manière générale, la syntaxe de cette directive fait que le symbole à remplacer ne peut contenir d'espace (puisque le premier espace sert de délimiteur entre le symbole à substituer et le texte de substitution). Le texte de substitution, quant à lui, peut contenir autant d'espaces que vous le souhaitez puisque c'est la fin de ligne qui termine la directive. Il est même possible de le prolonger au-delà, en terminant la ligne par \ et en poursuivant sur la ligne suivante.
Vous voyez que, compte tenu des possibilités d'imbrication des substitutions, cette instruction s'avère très puissante au point qu'elle pourrait permettre à celui qui le souhaiterait de réécrire totalement le langage C (on pourrait, par exemple, le franciser). Cette puissance a toutefois ses propres limites dans la mesure où tout abus dans ce sens conduit inexorablement à une perte de lisibilité des programmes.

Remarque :

Si vous introduisez, par mégarde, un signe = dans une directive **`#define`**, aucune erreur ne sera, bien sûr, détectée par le préprocesseur lui-même. Par contre, en général, cela conduira à une erreur de compilation. Ainsi, par exemple, avec :
`#define N = 5`
une instruction telle que : **`int t[N] ;`** deviendra, après traitement par le préprocesseur : **`int t[= 5] ;`**
laquelle est manifestement erronée. Notez bien, toutefois, que, la plupart du temps, vous ne connaîtrez pas le texte généré par le préprocesseur et vous serez simplement en présence d'un diagnostic de compilation concernant apparemment l'instruction **`int t[N].`** Le diagnostic de l'erreur en sera d'autant plus délicat.
Une autre erreur aussi courante que la précédente consiste à terminer (à tort) une directive **`#include`** par un point-virgule. Les considérations précédentes restent valables dans ce cas.

4.2.2 Définition de macros

La définition de macros ressemble à la définition de symboles mais elle fait intervenir la notion de paramètres. Par exemple, avec cette directive :
`#define carre(a) a*a`
le préprocesseur remplacera dans la suite tous les textes de la forme : **`carre(x)`** dans lesquels **`x`** représente en fait un symbole quelconque par : **`x*x`**
Par exemple :

`carre(z)`	deviendra	**`z*z`**
`carre(valeur)`	deviendra	**`valeur*valeur`**
`carre(12)`	deviendra	**`12*12`**

La macro précédente ne disposait que d'un seul paramètre, mais il est possible d'en faire intervenir plusieurs en les séparant, classiquement, par des virgules. Par exemple, avec :

`#define dif(a,b) a-b`

`dif(x,z)`	deviendrait	**`x-z`**
`dif(valeur+9,n)`	deviendrait	**`valeur+9-n`**

Là encore, les définitions peuvent s'imbriquer. Ainsi, avec les deux définitions précédentes, le texte : **`dif(carre(p),carre(q))`** sera, dans un premier temps, remplacé par : **`dif(p*p,q*q)`** puis, dans un second temps, par : **`p*p-q*q`**

FSO.UMPOUJDA.COM

Néanmoins, malgré la puissance de cette directive, il ne faut pas oublier que, dans tous les cas, il ne s'agit que de substitution de texte. Il est souvent nécessaire de prendre quelques précautions, notamment lorsque le texte de substitution fait intervenir des opérateurs. Par exemple, avec ces instructions :

```
#define     DOUBLE(x)      x + x
.....
DOUBLE(a)/b
DOUBLE(x+2*y)
DOUBLE(x++)
```

Le texte généré par le préprocesseur sera le suivant :

```
a + a/b
x+2*y + x+2*y
x++ + x++
```

Vous constatez que, si le premier appel de macro conduit à un résultat correct, le deuxième ne fournit pas, comme on aurait pu l'escompter, le double de l'expression figurant en paramètre. Quant au troisième, il fait apparaître ce que l'on nomme souvent un « effet de bord ». En effet, la notation : **DOUBLE(x++)** conduit à incrémenter deux fois la variable **x**. De plus, elle ne fournit pas vraiment son double.
Par exemple, si **x** contient la valeur **5**, l'exécution du programme ainsi généré conduira à **calculer 5+6.**
Le premier problème, lié aux priorités relatives des opérateurs, peut être facilement résolu en introduisant des parenthèses dans la définition de la macro. Ainsi, avec :

```
#define DOUBLE(x) ((x)+(x))
...
DOUBLE(a)/b
DOUBLE(x+2*y)
DOUBLE(x++)
```

Le texte généré par le préprocesseur sera :

```
((a)+(a))/b
((x+2*y)+(x+2*y))
((x++)+(x++))
```

Les choses sont nettement plus satisfaisantes pour les deux premiers appels de la macro **DOUBLE**. Par contre, bien entendu, l'effet de bord introduit par le troisième n'a pas pour autant disparu. Par ailleurs, il faut savoir que les substitutions de paramètres ne se font pas à l'intérieur des chaînes de caractères. Ainsi, avec ces instructions :

```
#define   AFFICHE(y)   printf("valeur de y %d",y)
...
AFFICHE(a) ;
AFFICHE(c+5) ;
```

le texte généré par le préprocesseur sera :

```
printf("valeur de y %d",a) ;
printf("valeur de y %d",c+5) ;
```

FSO.UMPOUJDA.COM

Remarque :

Dans la définition d'une macro, il est impératif de ne pas prévoir d'espace dans la partie spécifiant le nom de la macro et les différents paramètres. En effet, là encore, le premier espace sert à délimiter la macro à définir. Par exemple, avec :

```
#define somme (a,b) a+b
...
z = somme(x+5) ;
```

le préprocesseur générerait le texte :

```
z = (a,b) a+b(x+5) ;
```

4.3 La compilation conditionnelle

La construction ci-après :

```
#if condition
.....
#else
.....
#endif
```

permet d'incorporer l'une des deux parties du texte, suivant la valeur de la condition indiquée. En voici un exemple d'utilisation :

```
#define CODE 1
.....
#if CODE == 1
     instructions 1
#endif
#if CODE == 2
     instructions 2
#endif
```

Ici, ce sont les **instructions 1** qui seront incorporées par le préprocesseur. Mais il s'agirait **des instructions 2** si nous remplacions la première directive par :

```
#define CODE 2
```

Notez qu'il existe également une directive **#elif** qui permet de condenser les choix imbriqués. Par exemple, nos précédentes instructions pourraient s'écrire :

```
#define CODE 1
.....
#if CODE == 1
          instructions 1
#elif CODE == 2
          instructions 2
#endif
```

FSO.UMPOUJDA.COM

D'une manière générale, la condition mentionnée dans ces directives **#if** et **#elif** peut faire intervenir n'importe quels symboles définis pour le préprocesseur et des opérateurs relationnels, arithmétiques ou logiques. Ces derniers se notent exactement de la même manière qu'en langage C.
En outre, il existe un opérateur noté **defined**, utilisable uniquement dans les conditions destinées au préprocesseur (**if** et **elif**). Ainsi, par exemple on pourrait également s'écrire :

```
#define MISEAUPOINT
.....
#if defined(MISEAUPOINT)
        instructions 1
#else
        instructions 2
#endif
```

D'une manière générale, les directives de test de la valeur d'une expression peuvent s'avérer précieuses :

- pour introduire dans un fichier source des instructions de mise au point que l'on pourra ainsi introduire ou supprimer à volonté du module objet correspondant. Par une intervention mineure au niveau de source lui-même, il est possible de contrôler la présence ou l'absence de ces instructions dans le module objet correspondant, et ainsi, de ne pas le pénaliser en taille mémoire lorsque le programme est au point.
- pour adapter un programme unique à différents environnements. Les paramètres définissant l'environnement sont alors exprimés dans des symboles du préprocesseur.

FSO.UMPOUJDA.COM