LE LANGAGE C
Définition de la norme ANSI

Jean Louis Nebut
IFSIC

Cours C81
Révisé juillet 94

Table des matières

Chapitre 1

INTRODUCTION

1.1 Caractéristiques du langage

Conçu initialement pour écrire le système UNIX, c'est un langage très proche des capacités d'un calculateur des années soixante dix : il permet de mettre en œuvre les opérateurs du matériel (les décalages, les opérations sur les bits) et d'adresser les objets par les fonctions d'adressage de base (adressage direct, indirect, indexé).

Ce n'est pas un langage de haut niveau qui met en avant une stricte notion de type, bien que les variables et les paramètres soient typés.

Il n'y a qu'un seul niveau de déclaration de sous-programmes (qui ne peuvent donc pas s'emboîter), mais plusieurs niveaux de déclaration de variables. L'allocation des variables locales est automatique.

Comme l'assembleur, c'est un langage pour spécialistes : il faut bien comprendre les mécanismes d'adressage, et en particulier l'indirection, il faut avoir assimilé correctement le travail d'un compilateur et d'un éditeur de liens pour découvrir ce que ces logiciels peuvent laisser comme erreurs de programmation.

1.2 Commande de compilation

L'appel du compilateur (**cc** ou **gcc** sous UNIX) pour compiler un programme complet met en jeu trois logiciels :

- d'abord un pré-compilateur, qui joue le rôle d'un macro-processeur (l'équivalent d'un macro-assembleur pour un langage d'assemblage) : substitution de chaînes, expansion des macros, traitement de la compilation conditionnelle. L'option `-E` permet d'arrêter là le traitement (pré-compilation seule, résultat sur la sortie standard).
- puis le compilateur
- enfin l'éditeur de liens, qui engendre l'exécutable `a.out` (ou *xx* avec l'option `-o` *xx*).

Il n'y a pas possibilité de demander un listing de compilation.

1.3 Le niveau lexical

Il y a trois sortes d'entités lexicales : les identificateurs, les nombres et les chaînes. L'alphabet utilisé comprend les caractères du jeu de caractères du calculateur. À l'IFSIC, il s'agit du jeu ASCII à 128 caractères pour les identificateurs, et des jeux ISO-Latin à 256 caractères sous UNIX et ASCII étendu à 256 caractères sous MS/DOS pour les commentaires et les chaînes.

Les minuscules et les majuscules sont considérées comme étant différentes, comme sous UNIX. Par convention (historique), on réserve les majuscules pour les identificateurs définis au niveau du pré-compilateur et pour les identificateurs de type non prédéfini.

1.3.1 Les séparateurs

Les espaces, tabulation et retour de ligne sont ignorés ; ils servent de séparateurs d'entités lexicales.

1.3.2 Les commentaires

C'est une suite de caractères quelconques (incluant les changements de ligne) encadrée par les séquences /* et */

1.3.3 Les identificateurs

Un identificateur est composé d'une lettre, éventuellement suivie d'une suite de lettres et de chiffres. Le caractère souligné est considéré comme une lettre, mais son usage en première position n'est pas conseillé (le compilateur ajoute un souligné devant les identificateurs d'objets externes transmis à l'éditeur de liens). Deux identificateurs doivent se différencier par leurs 31 premiers caractères.

Exemple : *entier_lu dernier_mot mot1 Mot1*
Les deux derniers identificateurs sont différents.

1.3.4 Les nombres

Les entiers peuvent s'écrire en décimal : suite de chiffres dont le premier n'est pas zéro ; en octal, suite de chiffres octaux commençant par un zéro ; ou en hexadécimal, suite de chiffres hexa commençant par 0x. Un nombre négatif est précédé d'un signe moins. Le signe plus unaire a été introduit par la norme C.

Exemple : *129 012 Ox1a ou Ox1A -192*

Les nombres réels comprennent soit un point décimal, soit une partie exposant précédée de la lettre e. Leur syntaxe est :

RÈGLE

La partie entière ou la partie décimale est obligatoire ; le point ou l'exposant est obligatoire.

Exemple : *.18 -3.14156 18e-2*

1.3.5 Les chaînes

Une chaîne est une suite de caractères ne comprenant pas de retour de ligne et encadré par des guillemets. Un guillemet dans une chaîne est noté \"

Exemple :

```
"Ceci est une très longue chaîne qui ne tient pas sur une ligne,\
alors on met un \ devant le retour de ligne pour l'annuler; \
le \" est aussi précédé d'une contre-barre"
```

La norme C accepte également de couper une chaîne par un retour de ligne à condition d'entourer de guillemets toutes les parties de la chaîne : c'est le pré-compilateur qui regroupera les différentes chaînes adjacentes.

Exemple :

```
"Voila le première partie,"
"et voici la seconde."
```

1.3.6 Les symboles

Certains identificateurs servent de mot-clé. Ce sont :

auto	**default**	**float**	**register**	**struct**	**volatile**
break	**do**	**for**	**return**	**switch**	**while**
case	**double**	**goto**	**short**	**typedef**	
char	**else**	**if**	**signed**	**union**	
const	**enum**	**int**	**sizeof**	**unsigned**	
continue	**extern**	**long**	**static**	**void**	

La plupart des caractères non lettres ou chiffres et leurs associations servent également de symboles (opérateurs, délimiteurs...) :

```
+ - * / % < > = ! ? & | ~ . [ ] { } ( ) ^ : , ; #
++ -- << >> == || && != *= -= += %= <= >= &= ^= -> <<= >>=
```

1.4 Structure d'un programme

Il n'y a pas de notion de programme à proprement parler. C ne connait que la notion de sous-programme, restreint à la notion de fonction dans les premières versions, et le compilateur connait la notion de module, ensemble de variables (globales ou locales au module) et de fonctions. L'éditeur de liens lui sait fabriquer un exécutable lorsque l'un des identificateurs d'externes fabriqués par le compilateur s'appelle ***_main***. Ce qu'on appelle habituellement "le programme" est donc en C une procédure dont le nom est ***main***.

La structure la plus simple à donner au compilateur est un module composé d'une seule procédure qui s'appelle ***main***. Elle est la suivante :

```
void main ()
{<les déclarations locales>
 <les instructions>
}
```

Le mot-clé **void** permet de distinguer une procédure d'une fonction. Ainsi, un programme qui se contente d'afficher un message ressemble à :

```
void main ()
{printf ("coucou !") /*Affiche coucou !*/ ;
}
```

Mais ce n'est pas suffisant, car le compilateur ne connait pas le sous-programme d'affichage ***printf*** : il fait partie d'une bibliothèque qu'il faut indiquer à la compilation. Aucune entrée-sortie n'est définie en C, mais il existe plusieurs bibliothèques standard.

1.5 Règles de présentation de la syntaxe

La grammaire du langage est présentée ici sous une notation BNF modifiée. Chaque règle est précédée du mot RÈGLE ; les mots-réservés sont en gras ; les symboles du métalangage sont en écriture haute et ont comme signification :

:	sépare le nom de la règle de la règle elle-même
{ }	parenthèse de groupage

| | alternative

[ ] entité optionnelle

[ ]$^{+}$ entité qui doit être présente une fois ou plus

[ ]* entité qui peut être répétée zéro fois ou plus

Chapitre 2

TYPES, CONSTANTES ET VARIABLES

Les objets C peuvent être de plusieurs types :

- d'un des types de base prédéfinis dans le langage
- d'un type énuméré
- d'un type article
- d'un type union (article en recouvrement)
- ou d'un type défini dans une définition de type

RÈGLE
type :
type de base | type structure | type union |
type énuméré | type défini

On remarque qu'on ne parle pas ici de type tableau ni de type pointeur : les notions syntaxique et sémantique de type sont distinctes en C. On étudie dans ce chapitre les types de base, les types énumérés et les définitions de type.

2.1 Les types de base

Ils sont au nombre de quatre : entier, réel, caractère, et vide.

2.1.1 Les types entiers

Il y a six types entiers, selon la syntaxe suivante, en fonction de la "taille" des valeurs :

RÈGLE
type entier :
{[**signed** | **unsigned**]}{[**short**]|[**long**]} **int**

int valeurs entières signées, en général de la taille des registres de la machine. Sur SUN et HP, les valeurs de type **int** sont codées sur 32 bits et vont de -2^{31} à $+2^{31}-1$. Sur PC (Turbo-C), le codage est sur 16 bits (-2^{15} à $+2^{15}-1$). Abréviation pour **signed int**.

short int entiers courts signés, codés sur deux octets : de -2^{15} à $+2^{15}-1$ ($-32\ 768$ à $+32\ 767$). Pas de différence sur PC. Abréviation pour **signed short int**.

short identique à **short int**

long int entiers longs signés ; codés comme **int** sur 32 bits sur HP, SUN(3 et 4) et PC. Les littéraux de type **long** sont suivis de la lettre l ou L : 0l est le 0 long, 124L est la valeur 124 de type long. Certaines fonctions des bibliothèques prédéfinies ont des paramètres **long**. Abréviation pour **signed long int**.

long identique à **long int**.

unsigned int valeurs entières non signées, de la taille d'un registre. Codées sur 32 bits sur SUN et HP, valeurs de 0 à $2^{32} - 1$

unsigned identique à **unsigned int**

unsigned short int valeurs codées sur deux octets, de 0 à 65 535

unsigned short identique à **unsigned short int**

unsigned long int valeurs longues positives

unsigned long identique à **unsigned long int**

2.1.2 Les types réels

Il y a trois types réels :

float réel codé sur quatre octets "simple précision", ayant au moins sept chiffres significatifs et une puissance de dix de 38 au maximum

double réel codé sur huit octets "double précision", avec quinze chiffres significatifs et une puissance de dix jusqu'à 306

long double pour extension future

Les littéraux réels sont théoriquement toujours considérés comme étant **double**.

2.1.3 Les types caractères

Il y en a deux :

char il sert à coder les valeurs du jeu de caractères ASCII restreint, mais il est en fait identique à **short int** codé sur un octet. Ses valeurs vont donc de −128 à +127, mais les littéraux caractères imprimables ont une notation particulière (plus loin)

unsigned char recouvre les valeurs entières de 0 à 255, permettant ainsi le codage de l'ISO-Latin et de l'ASCII étendu.

Les littéraux de type caractère et qui ont une représentation graphique se notent par cette représentation placée entre apostrophe, sauf pour l'apostrophe, le guillemet et la contrebarre. Les caractères non affichables ont une notation particulière.

notation normale : 'a' ';' 'A'

notation particulière :

'\n'	caractère NL	(nouvelle ligne)
'\r'	caractère RET	(retour chariot)
'\b'	caractère BS	(retour arrière)
'\t'	caractère TAB	(tabulation)
'\v'	caractère VT	(tabulation verticale)
'\f'	caractère FF	(saut de page)
'\"'	caractère guillemet	
'\''	caractère quote	
'\\'	caractère contre barre	

Enfin, tout caractère de code octal nnn ou de code hexa nn peut être dénoté par '\nnn' ou '\xnn'.

2.1.4 Le type void

Le type **void** est le type des sous-programmes qui ne rendent pas de valeurs (type des fonctions qui s'utilisent comme des procédures).

2.2 Le type des valeurs booléennes

Il n'y a pas en C de type booléen. Par contre, les expressions booléennes existent. Par convention, toute valeur entière peut être considérée comme une valeur booléenne, en suivant la règle :

- faux est codé par un entier nul
- vrai est codé par un entier non nul.

2.3 Les types énumérés

Dans un *type énuméré*, le codage des valeurs du type est réalisé par des identificateurs (qui désignent alors des constantes).

RÈGLE
type énuméré :

enum [identificateur] {identificateur [= init]
[, identificateur [= init]]* } ;

Le premier identificateur désigne le type ; quand il est omis, le point virgule doit être précédé des variables du type (type anonyme). Les autres identificateurs sont les valeurs du type ; ces valeurs sont codées par défaut par 0,1,2,..., sauf si une expression d'initialisation vient modifier ces valeurs. Le codage normal (valeur précédente +1) reprend en l'absence d'initialisation.

Exemple :

```
enum jour {lun, mar, mer, jeu, ven, sam, dim} ;
   /* attention, le type  n'est pas jour, mais enum jour */
   /* lun est codé par 0, dim par 6 */
enum priorite {basse = -1, normale, moyenne = 5,
   grande, super = 10} ;
   /* normale "vaut" 0, grande vaut 6 */
```

2.4 Les définitions de type

Quand on a défini le type **enum** *jour* {...}, *jour* n'est pas considéré comme un nouveau type. Une définition d'un type que l'on veut désigner par un identificateur se fait grâce à une déclaration **typedef**

RÈGLE
type défini :
typedef type déclarateur ;

Un déclarateur complète la sémantique du type, et en même temps, donne le moyen de désigner le type. Le déclarateur le plus simple est un identificateur.

Exemple :

```
typedef  int LONGUEUR /* une LONGUEUR est codée par un entier */ ;
typedef  enum {faux, vrai} BOOLEEN ;
```

Le type ***BOOLEEN*** ainsi défini correspond bien à la convention annoncée : ***faux*** est codé par 0, et ***vrai*** par une valeur non nulle, 1 ici. On remarque qu'on ne peut pas affiner les types entiers ou caractères par des intervalles.

2.5 Les constantes

Le langage C ne connait que les constantes littérales. Pour désigner un tel littéral par un identificateur, on utilise le pré-compilateur, en déclarant une macro qui a pour nom l'identificateur choisi, et pour corps le littéral, ou l'expression de constante adéquate. Cette expression doit être calculable à la compilation.

RÈGLE
déclaration de macro pour le pré-compilateur :
define identificateur [(identificateur [, identificateur]*)] texte

La liste d'identificateur entre parenthèse est une liste de paramètres formels pour les macros qui sont des instructions.

Attention !, le # doit être en première colonne, il n'y a pas d'espace avant le **define** dans les anciennes versions de C. Toute occurrence de l'identificateur sera substituée lors de la pré-compilation par le texte qui lui est associé. Le texte peut contenir des blancs, et il se termine par une fin de ligne.

Exemple :

```
# define MAXCAR 20
# define MAXTAB 30
# define PLACETAB (MAXCAR * MAXTAB)
```

On ne peut pas mettre de commentaire sur une ligne **#define** sans le voir réapparaître à chaque substitution de la macro ! Les parenthèses ne sont pas forcément nécessaires, mais elles sont souhaitables (penser à une expression comme ***x/PLACETAB*** : sans les parenthèses, le macro-processeur la remplacera par ***x/20*30***, ce qui n'est pas ce qu'on attend).

2.6 Déclaration de variables

Une déclaration de variable définit à la fois :

- le nom de la variable
- le nom de son type
- son type, qui est obtenu en combinant le nom de son type et une information supplémentaire contenue dans le déclarateur
- sa classe d'allocation, c'est-à-dire comment est allouée la variable.

RÈGLE
déclaration de variable :
[classe] type déclarateur-init [, déclarateur-init]* ;

Comme on l'a déjà dit, le déclarateur le plus simple est un identificateur (le nom de la variable). Un *déclarateur-init* est un déclarateur avec valeur initiale.

RÈGLE
déclarateur-init :
déclarateur [= init]

Exemple :

```
int a, b, c;      /* trois entiers désignés par a, b, c */
enum jour j;   /* j peut prendre les valeurs lun, mar, ... dim */
BOOLEEN trouve, pasla = vrai;
   /* seul pasla est initialisé à vrai */
```

La classe d'allocation sert à la fois à dire où est allouée la variable et quelle est sa portée. Par défaut, une variable déclarée dans une fonction est allouée sur la pile, tandis qu'une variable déclarée en dehors d'une fonction est déclarée dans un segment de données permanent. Cette notion est examinée dans le chapitre concernant la structure des programmes (paragraphe 9.3.1).

Les variables sont initialisées soit explicitement (par = *init*), soit implicitement dans le cas des variables globales (valeur initiale nulle pour les variables de classe **extern** et **static**).

Chapitre 3

LES EXPRESSIONS

Il y a quarante cinq opérateurs répartis en quinze classes de priorités différentes en C. Ces opérateurs permettent une énorme puissance d'expression, mais attention, les parenthèses sont souvent indispensables malgré le nombre des priorités.

Certains opérateurs servent à combiner des valeurs, d'autres servent à réaliser des adressages. On n'étudie dans ce chapitre que les premiers.

Il est nécessaire de consulter le tableau des priorités tout en lisant ce chapitre (page 15).

3.1 Expressions arithmétiques

Les opérateurs sur les types entiers et caractères sont : +, −, *, /, et % (modulo). Seul le moins unaire existe dans les versions de C datant d'avant la norme.

Les opérateurs sur les réels sont : +, −, * et /, plus le moins et le plus unaire. Ainsi, 3/5 donne 0 tandis que 3.0/5 donne 0.6.

Les dépassements de capacité ne sont pas détectés sur les entiers. Les divisions par zéro et les débordements sur les réels sont détectés ou non selon les compilateurs et selon les exécutifs C.

A priorités égales, les opérateurs sont évalués de la gauche vers la droite, mais le compilateur peut tenir compte de la commutativité de l'addition et de la multiplication. Ainsi, dans `f(a) + f(b)`, `f(a)` ou `f(b)` peut être évalué d'abord, ce qui interdit à `f` d'avoir un effet de bord.

Exemple :

```
x + y * 3 - (a + 3 * z)
carlu - 'a' + 10
```

3.2 Expressions logiques

Les opérateurs logiques s'appliquent sur des opérandes entiers positifs puisqu'il n'y a pas de type booléen. Ce sont :

!	négation
&&	et logique
||	ou logique

Les opérateurs && et || sont à circuit court : le deuxième opérande n'est évalué que si c'est indispensable pour le calcul du résultat. Ainsi, dans l'expression `a && b`, `b` n'est évalué que si `a` est vrai (c'est à dire non nul).

Exemple : `!3` vaut `0`
`!(5 || x)` vaut `0` (`!` est plus prioritaire que ||)

3.3 Expressions relationnelles

Les opérateurs de relations n'ont pas tous la même priorité : l'égalité et la différence sont moins prioritaires que les autres. Ce sont :

== et !=	égalité et différence
< <= > >=	comparaisons

Exemple :

```
carlu >= 'a' && carlu <= 'z' ||
carlu >= 'A' && carlu <= 'Z'
```

est vrai si `carlu` est une lettre. Les priorités sont telles que les parenthèses sont inutiles.

3.4 Expressions de manipulation de bits

Les opérandes sont des entiers traités comme des chaînes de bits. Les opérateurs correspondent aux instructions-machine sur les mots.

~ complément à 1 (inversion des bits)

<< décalage à gauche : des zéros entrent à droite

>> décalage à droite, arithmétique pour les opérandes **int**, logique pour les opérandes **unsigned** (extension de signe, ou des zéros entrent à gauche)

& et bit à bit

| ou (inclusif) bit à bit

^ ou exclusif bit à bit

Dans un décalage, l'opérande droit doit être positif et inférieur au nombre de bits d'un mot-machine.

	31	&&	4	vaut 1	31 = 00011111
	31	&	4	vaut 4	4 = 00000100
	31	||	4	vaut 1	
Exemple :	31	|	4	vaut 31	
	31	^	4	vaut 28	
	31	>>	4	vaut 1	
	31	<<	4	vaut 496	

3.5 Les opérateurs d'affectation

En C, une affectation est une expression qui a comme valeur l'expression affectée à l'opérande gauche. Syntaxiquement, l'opérande gauche est également une expression qui désigne une variable. On l'appelle identification dans les règles :

RÈGLE
expression d'affectation :
identification = expression

L'expression de droite et l'expression de gauche (l'identification de la variable) sont évaluées dans n'importe quel ordre, et la variable désignée reçoit la valeur de l'expression de droite, après conversion éventuelle.

Il existe une deuxième forme :

RÈGLE
expression d'affectation :
identification op = expression
op :
+ | - | * | / | % | >> | << | & | ^ | |

identification op= expression

est une abréviation pour :

identification = identification op expression

mais l'identification de la variable n'est calculée qu'une fois ; en particulier, le résultat d'une fonction peut servir dans une identification.

Les opérateurs d'affectation ont une priorité plus faible que tous les opérateurs vus jusqu'ici ; ils sont associatifs de gauche à droite, si bien que :

```
x = y = z
```

est évalué comme :

```
x = (y = z)
```

Exemple :

```
x += 2          /* soit x = x + 2 */
y &= 0xFF       /* masque sur l'octet de poids faible */
x = 3 + y = 2   /* x vaut 5 et y vaut 2 */
```

Si les deux opérandes d'une affectation n'ont pas le même type, la valeur de gauche est convertie dans le type de la variable de droite selon les règles de conversions implicites.

On verra plus loin deux autres opérateurs d'affectation.

3.6 Les conversions implicites

Dans l'évaluation d'une expression, des conversions automatiques ont lieu si deux opérandes d'un opérateur ne sont pas du même type. La règle est que le type "le plus fort" l'emporte, c'est à dire celui qui contient la plus grande valeur absolue :

- s'il y a un **double**, **float** est converti en **double**
- s'il y a un **float**, **int** est converti en **float**
- s'il y a un **long**, **int** est converti en **long**
- s'il y a un **unsigned**, **int** est converti en **unsigned**
- **char**, **short** et **enum** sont convertis en **int**

3.7 Les conversions explicites : le forceur

Un *forceur* est une expression qui transforme une autre expression en une valeur d'un type donné :

RÈGLE
forceur :
(type) expression

Si le type récepteur est "plus fort" que le type de l'expression, il n'y a pas de problème. Dans l'autre sens, il y a troncature (d'un entier en **short** par exemple).

Exemple :

```
y = 3 * sqrt (( double)x) /* sqrt attend un paramètre double */
```

3.8 L'expression conditionnelle

Cette expression utilise l'opérateur ternaire ? :

RÈGLE
expression conditionnelle :
(expression1) ? expression2 : expression3

L'évaluation se fait ainsi :

- évaluation de l'*expression1*
- si elle est non nulle, évaluation de *expression2* qui est la valeur de l'*expression conditionnelle*
- sinon, évaluation de *expression3*, qui est la valeur de l'*expression conditionnelle*.

Exemple :

```
/* calcul du max de a et b */     max = (a > b) ? a : b
/* soit z = 4 */          x = (y = 3 > z) ? z = 2 : z = 1
/* y prend la valeur 3, z la valeur 1 et x la valeur 1 */
```

On verra un autre exemple au paragraphe 4.3.

3.9 Appel de fonction

Un *appel de fonction* est une expression qui a comme valeur la valeur délivrée par la fonction. La désignation de la fonction est syntaxiquement aussi une expression.

RÈGLE
Appel de fonction :
expression ([expression [, expression]*])

Le plus souvent, la fonction est désignée par un identificateur (mais on peut avoir des tableaux de fonctions et des pointeurs sur des fonctions). L'ordre d'évaluation des expressions paramètres effectifs est quelconque. Les paramètres effectifs sont convertis dans le type du paramètre formel, ou, si celui-ci n'est pas connu, les conversions suivantes sont réalisés :

- un paramètre effectif **float** est converti en **double**
- un **char** ou un **short** est converti en **int**
- un **unsigned char** ou **short** est converti en **unsigned int**.

On remarquera qu'un appel de fonction sans paramètre comporte les parenthèses. Si `f` est une fonction de type **void**, l'appel

```
f ; /* il faut écrire f()*/
```

est erroné (le code engendré est faux), mais cela ne provoque pas d'erreur de compilation.

3.10 Priorité des opérateurs

On ne donne pas la grammaire des expressions, qui est fastidieuse. Le tableau suivant présente l'ensemble des opérateurs (certains seront vus dans des chapitres ultérieurs), classés par priorité décroissante (les plus prioritaires en haut, plusieurs opérateurs ayant même priorité étant regroupés entre deux traits horizontaux). La troisième colonne indique le sens de l'associativité de ces opérateurs. En cas de doute, utilisez les parenthèses !

Opérateur	Usage	Associativité
()	Appel de fonction	
[ ]	indexation	
$->$	accès à un champ via pointeur	$\longrightarrow$
.	accès à un champ	
$-$	moins unaire	
$+$	plus unaire	
$++$	incrémentation	
$--$	décrémentation	
!	non logique	
~	complément à un	$\longleftarrow$
*	indirection	
&	prise d'adresse	
sizeof	taille d'un objet	
(type)	forceur	
*	multiplication	
/	division	$\longrightarrow$
%	modulo	
$+$	addition	$\longrightarrow$
$-$	soustraction	
$<<$	décalage gauche	$\longrightarrow$
$>>$	décalage droit	
$<$	plus petit	
$<=$	plus petit ou égal	$\longrightarrow$
$>$	plus grand	
$>=$	plus grand ou égal	
$==$	égalité	$\longrightarrow$
$!=$	différence	
&	et bit à bit	$\longrightarrow$
^	ou exclusif bit à bit	$\longrightarrow$
|	ou bit à bit	$\longrightarrow$
&&	et logique	$\longrightarrow$
||	ou logique	$\longrightarrow$
? :	expression conditionnelle	$\longleftarrow$
=, *=, /=, %=, +=, −=, &=, ^=, |=, <<=, >>=	opérateurs d'affectation	$\longleftarrow$
,	séquence d'expressions	$\longleftarrow$

Chapitre 4

ENTRÉES-SORTIES

Aucune opération d'entrée-sortie n'est définie en C, pas plus que la notion de fichier. Il existe une bibliothèque de fonctions et de constantes qui définit un type fichier et des opérateurs sur l'entrée et la sortie standard d'UNIX (ou MS/DOS). Dans ce chapitre, on ne voit que les opérateurs les plus usuels sur l'entrée et la sortie standard.

L'éditeur de liens se charge d'aller chercher cette bibliothèque si le compilateur le lui demande. Le compilateur a besoin de plusieurs définitions qui sont regroupées dans un fichier du répertoire */usr/include* (sous UNIX). On inclut ce fichier dans un module qui fait des entrées-sorties en plaçant la directive au pré-compilateur :

```
#include <stdio.h>
```

4.1 Lecture-écriture d'un caractère

getchar() Fonction de type **int** qui délivre le caractère suivant lu au clavier (en fait, sur l'entrée standard). La fonction délivre ***EOF***, une constante définie dans le fichier *stdio.h*, lorsque le caractère lu est en fait la marque de fin de fichier. La valeur de ***EOF*** n'est pas en général une valeur du type **char**, d'où le type de la fonction. La fin de fichier s'obtient en tapant un ^D au clavier sous UNIX, ou un ^Z sous MS/DOS.

putchar(c) Fonction qui s'emploie comme une procédure pour ajouter le caractère *c* sur le fichier standard de sortie.

Exemple :

```
char c;              /* On ne teste pas la fin de fichier */
c = getchar();       /* Lecture d'un caractère, avec écho */
putchar('\n');       /* Affichage d'un retour de ligne */
```

4.2 Lecture-écriture d'une ligne

gets() Fonction de type pointeur sur chaîne qui délivre la ligne tapée sur l'entrée-standard ; la marque de fin de ligne est remplacée par une marque de fin de chaîne dans le résultat. La fonction délivre ***NULL*** si la fin de fichier est rencontrée

puts(ch) Affiche la chaîne *ch* sur la sortie standard, en remplaçant la fin de chaîne par une fin de ligne.

4.3 Écriture formatée

printf (format, e_1, e_2, ..., e_n)

affiche les expressions $e_1, e_2, \ldots, e_n$ en fonction du format indiqué (une chaîne) ; les types des expressions sont : entier, réel, caractère ou chaîne. Pour chaque e_i correspond dans le format une spécification de format commençant par un % . ***printf*** affiche la chaîne contenue dans le format en remplaçant chaque spécification de format par l'expression interprétée selon la spécification.

La syntaxe d'une spécification de format est la suivante :

RÈGLE
% [-][L][.d][l]conversion

Conversion indique quelle est la conversion à faire (vers le type chaîne affichable). C'est une lettre, obligatoire :

d ou ***i*** entier signé en base 10

o entier non signé en octal

u entier non signé en base 10

x ou ***X*** entier signé en base 16, lettres en minuscules ou en majuscules

c caractère

s chaîne

f réel imprimé sous forme décimale [–]xxx.yyyyyy (6 chiffres par défaut)

e ou ***E*** réel imprimé sous forme [–]x.yyyyyye±zz (e ou E, 6 chiffres décimaux)

Le signe – après le % indique un cadrage à gauche dans le champ de longueur L (par défaut, cadrage à droite). Si l'expression occupe moins de L caractères, elle est précédée de blancs, ou de zéros si L commence par un zéro. Si l'expression occupe plus de L caractères, le champ est étendu (pas de troncature, sauf éventuellement avec la conversion s). En absence de L, l'expression occupe autant de caractères qu'elle en contient.

d est un entier qui indique le nombre de chiffres après la virgule pour les conversions ***e*** et ***f***, et la longueur de la chaîne à écrire, entraînant éventuellement une troncature, pour la conversion ***s***.

l est la lettre l si l'entier est **long** ou si le réel est **double**, et c'est la lettre h si l'entier est **short**.

Appelée comme une fonction, ***printf*** délivre le nombre de caractères écrits, ou ***EOF*** (!) en cas d'erreur.

Exemple :(1) ***x*** est un entier qui vaut 31, ***y*** un réel qui vaut 135,9385, ***ch*** une chaîne qui vaut "une petite chaîne". L'appel :

```
printf ("Affichage : %d = %x en hexa.%6.2f et : %-10.5s !", x,x,y,x);
```

affichera sur une seule ligne (pas de caractères '\n' dans le format) :
Affichage :␣ 31 = 1f ␣ en hexa. 135.93 et :une ␣ p␣␣␣␣␣ !
(2) Mettre l's du pluriel :

```
printf("Il y a %d chaise%c \n", nbchaise, (nbchaise>1)? 's' : ' ');
```

La variable ***nbchaise*** est affichée avec la spécification ***%d***, et le caractère résultat de l'expresssion conditionnelle est affiché avec la spécification ***%c*** (le s ou rien).

4.4 Lecture formatée

scanf (format, a_1, a_2,..., a_n*)*

lit sur l'entrée standard des valeurs suivant le format indiqué, et les range dans les variables dont les adresses sont $a_1, a_2, \ldots, a_n$. Pour une variable simple ou un article, l'adresse est obtenue en faisant précéder l'identificateur par un "et commercial" (Voir plus loin paragraphe 7.2). Pour chaque a_i doit correspondre dans la chaîne format une spécification de format commençant par un %. La syntaxe d'une spécification de format est la suivante :

%[*][L][l]conversion

Conversion indique quelle est la conversion à faire (depuis le type chaîne lu). C'est une lettre, obligatoire, avec le même codage que pour ***printf***. Cependant, la spécification ***%s*** peut être remplacée par une expression régulière (simplifiée par rapport à celles qu'acceptent les commandes UNIX) qui détermine l'ensemble des caractères admis pour la chîne à lire. L'ensemble des caractères est défini par la notation *[abc]* et ses variantes (*[^abc]*, *[a-c]*). Ainsi, la spécification *%[0-9]* définit un recherche d'entier.

L'étoile indique que la valeur lue suivant la spécification de format n'est pas à affecter à une variable (et donc, la spécification %*... ne "consomme" pas un a_i).

L est la longeur du champ à lire. Par défaut (pas d'entier L), la lecture d'un nombre s'arrête sur le premier caractère qui ne peut pas faire partie d'un nombre.

l est la lettre l si l'entier est **long** ou si le réel est **double**, et c'est la lettre h si l'entier est **short**.

Un format peut contenir des séparateurs de spécification. Il y a deux types de séparateurs, selon la présentation des données à lire :

- sans séparateur, c'est la longueur du champ à lire qui détermine le nombre de caractères à lire ;
- avec un séparateur blanc ou TAB, les champs à lire sont séparés par un ou plusieurs blancs, TAB ou RET ;
- avec un séparateur qui n'est pas le blanc ou TAB, les données doivent être séparées par ce séparateur (qui est ignoré lors de la lecture).

La fonction ***scanf*** délivre le nombre de champs lus et correctement convertis, ou ***EOF*** en fin de fichier.

Exemple :

Soient les déclarations :

```
int u, v; short y; float z; char c;
```

Supposons que l'on lise :
123456 ␣␣ 23.25 RET
1.82,152*.
avec l'instruction :

```
scanf ("%4d%h %*f %f,%d%c", &u, &y, &z, &v, &c);
```

Alors les variables sont initialisées comme suit :

```
u=1234 y=56 z=1.82 v=152 c='*'
```

et le prochain caractère à lire est le point.

4.5 Lectures et tamponnage du clavier

Le comportement apparent des lectures peut varier d'un système à l'autre, selon que les entrées sont tamponnées ou pas. Sous UNIX, et par défaut, les lectures au clavier sont tamponnées : si l'on lit un texte par des *getchar()*, les caractères tapés sont entrés dans un tampon d'UNIX tant qu'on n'a pas tapé le retour de ligne, et si le programme réaffiche aussitôt le caractère lu, on constate que cet affichage n'a pas lieu avant la frappe du RET. Dans l'exemple avec *scanf*, *x* et *y* sont effectivement initialisés, mais le *scanf* ne se terminera que si l'entier 152 est suivi d'un RET.

Que ce soit sous UNIX ou sous MS/DOS, on peut programmer l'absence de tamponnage.

Exemple :

```
/* Fichier somme.c */
/* Ce programme lit deux entiers et affiche leur somme et
   leur différence */
#include <stdio.h>
 void main()
  {  int a, b;
     printf ("Tapez deux entiers : ");
     scanf ("%d %d", &a, &b);
     printf ("Somme =  %d  Différence = %d\n", a + b, a - b);
  }
```

Sous Unix, on compile ce programme par

gcc somme.c

et on l'exécute par :

a.out

On termine par un dernier exemple :

Exemple :

```
/* Ce programme lit des entiers jusqúà la fin de fichier et
   affiche leur somme */
#include <stdio.h>
 void main()
  {  int s, n;
     s = 0;
     printf ("Tapez des entiers, terminez par ^D : ");
      while ( scanf ("%d", &n) != EOF )
        s += n;
     printf ("Somme =  %d\n", s);
  }
```

Chapitre 5

LES INSTRUCTIONS

Toutes les instructions peuvent être étiquetées. Certaines ont un terminateur point-virgule, d'autres pas ; il n'y a pas de séparateur systématique d'instruction.

RÈGLE
instruction :
[identificateur :]
{ instruction-vide | instruction-expression | instruction- tantque |
instruction-répéter | instruction-si | instruction-cas |
instruction-sortie | instruction-retour | instruction-pour |
instruction-continuer | instruction-allerà | instruction-composée }

L'identificateur qui peut préfixer une instruction est l'étiquette de cette expression.

5.1 Instruction-expression

Toute expression terminée par un point-virgule devient une *instruction* : on peut ainsi faire une affectation classique, ou appeler une fonction comme une procédure (on perd alors son résultat) comme on l'a fait dans les exemples précédents avec les fonctions d'entrées-sorties formatées.

RÈGLE
instruction-expression :
expression ;

Exemple :

```
x = a * (b + 5) ;                    /* instruction d'affectation */
printf ("x vaut de'sormais %d \n", x) ; /* appel de procédure */
```

5.2 Instruction composée, ou bloc

L'*instruction-composée*, ou *bloc*, peut comporter des déclarations de variables locales au bloc. La portée de ces variables est l'instruction composée. En général, on déclare des variables locales dans les blocs des fonctions, et pas dans les autres.

RÈGLE
instruction-composée :
{ [déclaration de variable]*

[déclaration de prototype]*
[instruction]+
}

Une *instruction composée* est nécessaire lorsqu'on doit mettre deux instructions ou plus quand la syntaxe en exige une seule. Les prototypes sont vus au paragraphe 9.2.

5.3 Instruction tantque

C'est la boucle avec test de fin en tête : on boucle tant que l'expression est vraie.

RÈGLE
instruction-tantque :
while (expression) instruction

Exemple :

```
/* (1) recopie de l'entrée sur la sortie */
{  int c;
   c = getchar();
   while (c != EOF)
     {putchar(c); c = getchar();}
}
```

On peut déplacer l'affectation `c = getchar()` dans l'expression du tantque, ce qui réduit le corps de la boucle au `putchar()` et supprime l'initialisation. Comme l'opérateur d'affectation est moins prioritaire que l'opérateur d'inégalité, on doit parenthéser.

```
/* (2) idem, en simplifiant */
{  int c;
   while ((c=getchar()) != EOF)
     putchar(c);
}
```

5.4 Instruction répéter

C'est la boucle avec test en fin de boucle : on répète tant que l'expression est vraie (et non pas jusqu'à ce que l'expression devienne vraie).

RÈGLE
instruction-répéter :
do instruction **while** (expression) ;

Le corps de la boucle est syntaxiquement composé là aussi d'une seule instruction.

Exemple :

```
/* lecture d'une réponse */
{  int reponse;
   printf ("Votre choix");
   do { printf ("(0-4)? ");
```

```
        scanf ("%d", &reponse);
      } while (reponse < 0 || reponse > 4);
}
```

5.5 Instruction si

RÈGLE
instruction-si :
if (expression) instruction1 [**else** instruction2]

Dans le cas de si emboîtés, un **else** se raccroche au dernier **if** non encore apparié.

Exemple :

```
/* compter le nombre de lignes du fichier standard d'entrée */
{  int c, nl = 0;
   while ((c = getchar()) != EOF)
       if (c == '\n') nl += 1;
   printf ("il y avait %d lignes\n", nl);
}
```

5.6 Instruction cas

Attention, l'instruction cas du C a une sémantique pas ordinaire, qui oblige en fait à utiliser l'instruction de sortie dans chaque alternative du cas.

RÈGLE
instruction-cas :
switch (expression)
{ [[**case** expression-constante :]$^+$ [instruction]*]
  [**default** : [instruction]*]
}

L'expression qui gouverne le cas est d'abord évaluée, puis convertie dans le type **int** éventuellement, et sa valeur est recherchée successivement (séquentiellement) parmi toutes les étiquettes de cas (qui sont des expressions évaluables à la compilation). Dès qu'une telle étiquette est trouvée, *toutes les instructions* qui suivent (y compris celles des autres alternatives) sont exécutées (!). Le choix **default** est pris si aucune étiquette précédente convient. S'il n'y a pas de **default** et qu'aucune étiquette convient, le cas ne fait rien.

Exemple :

```
/* compter le nombre de voyelles, le nombre de blancs ou TAB,
et le nombre des autres caractères de l'entrée standard */
  {  int c, nbv, nbb, nba;
     nbb = nbv = nba = 0;
     while ((c = getchar()) != EOF)
       switch (c)
         { case 'a' :
           case 'e' :
           case 'i' :
```

```
            case 'o' :
            case 'u' :
              nbv += 1;
               break;
            case ' ' :
            case '\t' :
              nbb += 1;
               break;
            default :
              nba += 1;
          }
    printf("Il y avait %d voyelles, %d blancs et %d autres\n",
           nbv, nbb, nba);
  }
```

5.7 Instruction vide

Comme on peut placer des affectations dans le test d'une boucle, il arrive souvent que le corps d'une boucle soit une *instruction vide*.

RÈGLE
instruction-vide :
;

5.8 Instruction pour

L'*instruction pour* de C n'est pas une instruction de répétition avec gestion automatique d'un indice : c'est simplement une autre forme syntaxique pour l'instruction tantque.

RÈGLE
instruction-pour :
for ([expression1] ; [expression 2] ; [expression3]) instruction

Sa sémantique est donnée par la construction suivante :

```
 expression1;   /* initialisation de la boucle */
 while (expression2)
   { instruction; expression3; }
```

Donc, à part l'expression1, tout est réévalué à chaque itération.

Exemple :

```
/* Calcul de a^b par multiplications successives */
{  int a, b, p;
   p = 1;
   for (i = 0; i < b; i += 1)
   p = p * a;
}
```

On peut grouper plusieurs initialisations dans l'*expression1* lorsque ces initialisations sont des expressions, grâce à l'opérateur de composition séquentielle d'expressions.

5.9 Composition séquentielle d'expressions

RÈGLE
expressions-séquentielles :
expression [, expression]$^+$

Les expressions sont évaluées de gauche à droite, et l'expression résultante a pour valeur la valeur de la dernière expression. Par exemple, la boucle pour précédente peut s'écrire :

Exemple :

```
for (p = 1, i = 0; i < b; p = p * a, i += 1);
/* le corps de la boucle est ici une instruction vide */
```

Enfin, pour traiter un par un les paramètres du programme, disponibles dans le tableau **argv**, on utilise la boucle qui suit. Pour la comprendre, il faut avoir assimilé le chapitre 7.1 et le paragraphe 8.8.

Exemple :

```
while ( ++argv, --argc)
  traiter(argv);
```

5.10 Instruction allerà

Le branchement à une étiquette (l'identificateur de la règle instruction de la page 21) se fait par un classique **goto** :

RÈGLE
instruction allera :
goto étiquette ;

La portée d'une étiquette est le bloc de la fonction qui la définit.

5.11 Instruction de sortie

L'instruction de sortie permet de sortir du bloc d'une boucle ou du bloc d'une instruction cas. Elle permet de réaliser des boucles à sorties multiples.

RÈGLE
instruction-sortie :
break ;

A l'exécution de cette instruction, le bloc le plus externe de la boucle ou du cas est abandonné, et l'exécution reprend juste derrière la boucle ou le cas.

Exemple :

```
/* on regarde si l'entrée standard contient une sonnerie */
{ int c; enum {rien, sonnerie} s = rien;
  while ((c = getchar()) != EOF)
    if (c == '\007') {s = sonnerie; break;}
  if (s == sonnerie)
   printf("Sonnerie dans l'entre'e standard\n");
}
```

5.12 Instruction continuer

Cette instruction est peu utilisée et peu utilisable : placée dans le corps d'une boucle, elle termine l'itération en cours et provoque un rebouclage sur le test de la boucle.

RÈGLE
instruction-continuer :
    continue ;

5.13 Exercice

Écrivez un programme simulant une calculette munie des opérateurs plus, moins, multiplier, diviser et reste, fonctionnant sur des entiers, et ne sachant calculer qu'une formule à 2 opérandes (et un seul opérateur) ; c'est donc une boucle sans fin qui :

- lit le premier opérande, l'opérateur, puis le second opérande ;
- affiche le résultat, ou *** si le calcul est impossible.

Chapitre 6

LES TABLEAUX

C ne définit pas un type tableau, mais on déclare des variables tableaux grâce au déclarateur de tableau.

6.1 Déclaration d'un tableau

Rappelons la syntaxe de déclaration d'une variable (paragraphe 2.6).

RÈGLE
[classe] type déclarateur [= init] ;

Pour déclarer un tableau, on place un *déclarateur de tableau* en partie déclarateur :

RÈGLE
déclarateur-de-tableau :
déclarateur [[expression-constante]]

Dans ces conditions, le type de la déclaration de variable est le type des éléments du tableau (*type de base* ou *type structure*, ou *type énuméré*, ou *type défini*), et *l'expression-constante* (une expression évaluable à la compilation) indique le nombre d'éléments du tableau. Si le déclarateur est un identificateur, alors c'est un simple tableau (une dimension), et l'identificateur désigne le tableau.

Exemple :

```
int t[10];                /* tableau de 10 entiers */
char lettres [2*26];      /* tableau de 52 caractères */
```

Si le déclarateur est lui-même un déclarateur de tableau à une dimension, on obtient un tableau à deux dimensions, qu'on considère comme un tableau de tableaux.

Exemple :

```
float mat [N][N];             /* matrice carrée */
char lesmots [NBMOTS][LONGMAXMOT];
    /* tableau à NBMOTS entrées, dont chacune est un tableau
     de LONGMAXMOT caractères */
```

On verra au paragraphe 7.8 ce qui se passe si on omet la taille du tableau dans le déclarateur.

6.2 Identificateur de tableau

Dans une expression, la valeur dénotée par un identificateur de variable simple est la valeur de cette variable. Il n'en est pas de même pour un identificateur de tableau : dans une expression, la valeur dénotée par un identificateur de tableau est l'adresse du premier élément du tableau (ou la valeur d'un pointeur sur le tableau ; voir plus loin). Celà explique qu'il n'y a pas en C d'opérateur d'affectation de tableaux. Si `x` et `y` sont deux tableaux "similaires" (déclarés par exemple par **int** `x[N], y[N] ;)` l'expression `x = y` ne veut rien dire puisque `x` est considéré syntaxiquement comme une identification (expression qui identifie une variable) alors que c'est une adresse, une adresse constante bien sûr.

6.3 Opérateur d'indexation

La numérotation des éléments d'un tableau est toujours la même : le premier élément a le numéro 0, le second le numéro 1, et si le tableau est à n élément, le dernier est numéroté n-1.

L'opérateur d'accès sélectif est le crochet.

RÈGLE
expression-indicée :
expression1 [expression2]

expression2 est éventuellement convertie dans le type entier. Si `i` est sa valeur, et si *expression1* est un identificateur de tableau d'éléments de type `T`, ou une expression de type pointeur vers un type `T`, alors la valeur de l'expression indicée est la valeur du ième élément de type `T` qui suit l'élément pointé par *expression1*.

En clair, `t[i]` a pour valeur le i+unième élément du tableau `t`.

Exemple :

```
/* lire et afficher un tableau de 5 entiers */
{  int t[5] ;  int i ;
   for (i = 0 ; i < 5 ; i += 1)
     scanf ("%d", &t[i]) ;        /* adresse du ième élément */
   for (i = 0 ; i < 5 ; i += 1)
     printf ("%5d", t[i]) ;       /* valeur du ième élément */
}
```

Compte tenu de la remarque sur ce que dénote un identificateur de tableau, `t` dénote l'adresse du premier élément, et a donc la même valeur que `&t[0]`. Par conséquent, l'adresse `&t[i]` passée en paramètre à la fonction `scanf` a donc la valeur `t+i`. La boucle de lecture peut s'écrire aussi :

```
for (i = 0 ; i < 5 ; i += 1)
   scanf ("%d", t + i) ;
```

On verra plus loin que l'opérateur + ici n'ajoute pas la valeur `i` à `t`, mais la valeur `i` fois la taille d'un élément de `t` à `t`.

6.4 Initialisation d'un tableau

On peut initialiser un tableau lors de sa déclaration, avec les restrictions suivantes si l'on utilise un vieux compilateur : sa classe d'allocation doit être **static** ou ce doit être une variable globale (l'initialisation des variables locales à allocation dans la pile a été introduit dans C lors du processus de normalisation du langage). L'initialisation d'un tableau à une dimension consiste à énumérer entre accolades les valeurs des (premiers) éléments du tableau.

RÈGLE
init :
expression-constante | { init [, init]* }

Exemple :

```
int t[5] = {3, 3} ; /* t[0] et t[1] sont initialisés à 3 */
int m[3][2] = {{1, 1}, {2}, {3, 3}} ;
    /* m[1][1] n'est pas initialisé */
```

6.5 Les chaînes

Toute variable de la forme **char** *c[n]* peut être considérée comme une chaîne. Il n'y a pas de représentation de chaîne définie en C, mais la bibliothèque de manipulation de chaînes impose que tout tableau de caractères qui est une chaîne contienne un caractère de code nul, qui est le marqueur de fin de chaîne.

Ainsi, la constante chaîne *"chaîne"* est représentée par un tableau de 7 caractères, qui contient "c h a i n e \0".

Comme une chaîne est un tableau, ni plus ni moins, il n'y a pas d'opérateurs spécifiques dans le langage : on ne peut pas en particulier affecter de chaînes entre elles, ni déclarer une chaîne avec une initialisation à une constante chaîne ! Il faudrait en effet logiquement écrire :

```
char ch[7] = {'c','h','a','i','n','e','\0'};
```

pour déclarer et initialiser la variable *ch*. On verra une autre possibilité au paragraphe 7.8.

L'affectation *ch1 = "chaîne"* est interdite puisque *ch* est considérée comme une constante-adresse ; il faut utiliser la fonction de la bibliothèque *strcpy* :
strcpy (ch1,ch) ou *strcpy (ch1,"chaîne")*. Par contre, la fonction *scanf* admet la spécification de format chaîne (*%s*). Lire la chaîne *ch* se fera par :

```
scanf ("%s", ch)
```

sans mettre l'opérateur & puisque *ch* est une adresse.

Les fonctions de la bibliothèque sur les chaînes sont données au paragraphe 11.1.

Le chapitre suivant reprend la notion de tableau vu sous l'angle des pointeurs.

6.6 Exercice

Écrivez un programme qui :

- lit des entiers jusqu'à rencontrer une marque de fin de fichier ;
- ordonne au fur et à mesure ces entiers ;
- affiche les entiers en ordre croissant.

Prévoyez le cas où vous ne lisez pas à temps la marque de fin de fichier.

Chapitre 7

LES POINTEURS

7.1 Définition

Un pointeur est un objet dont les valeurs servent à désigner d'autres objets d'un même type. Les valeurs d'un pointeur sont les adresses de ces autres objets. Les pointeurs sont à usages multiples en C :

- ils font partie intégrante du mécanisme d'indexation, qui est en fait une variante de l'adressage indirect
- ils sont nécessaires pour passer des paramètres résultats à une fonction
- ils permettent de faire de l'allocation dynamique et de construire des structures de listes.

7.2 Opérateurs de prise d'adresse

L'opérateur de prise d'adresse est un opérateur unaire qui s'applique à une identification de variable.

RÈGLE
expression d'adresse :
& identification

Cet opérateur ne s'applique pas à des constantes, ni à des variables de classe **register**, ni à des champs de bits (paragraphe 10.8).

Exemple :

```
int x;
int t[N];
 /* &x a pour valeur l'adresse de x
    &t[3] a pour valeur l'adresse du quatrième élément de t
    &t est interdit (t est une constante)
 */
```

7.3 Déclarateur de pointeur

Comme pour les tableaux, il n'existe pas de type pointeur. Une variable pointeur est définie par un *déclarateur de pointeur.*

RÈGLE
déclarateur-de-pointeur :
  * déclarateur

Ainsi la déclaration suivante :

```
int *p ;
```

déclare une variable `p` qui est un pointeur sur des objets de type entier. Ici, `p` est un déclarateur simple réduit à un identificateur. Ce peut être aussi un déclarateur de tableau, pour déclarer un tableau de pointeurs ou un pointeur de tableaux. Afin de pouvoir interpréter une déclaration telle que :

```
int *p[N] ; /* pointeur de tableaux ou tableau de pointeurs ? */
```

on utilise les caractéristiques des opérateurs * et [ ], priorité et associativité. Les crochets sont associatifs à droite et sont plus prioritaires que l'étoile, associative à gauche. La déclaration précédente s'interprète donc comme un tableau `p` à `N` éléments qui sont des pointeurs d'entier.

Pour déclarer un pointeur de tableaux, on a recours aux parenthèses :

```
int (*p)[N] ;
```

déclare une variable `p` qui est un pointeur sur un tableau de `N` entiers.

7.4 Opérateur d'adressage indirect

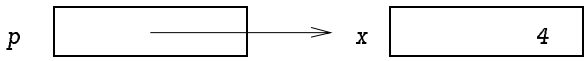
Le pointeur va servir à faire des indirections, et à obtenir des valeurs par indirection.

RÈGLE
expression-par-indirection :
  * expression

L'expression doit être du type pointeur vers un type `T`, et le résultat de l'expression par indirection est la valeur de type `T` repérée par la valeur de l'expression.

Exemple :

```
int x = 4 ;  int y ;
int *p = &x ;
     /* p est un pointeur d'entier initialisé à l'adresse de x */
     /* l'affectation x = y peut s'écrire : */
     *p = y
```



On peut substituer `*p` à toute occurrence de `x` après l'initialisation de `p` par l'adresse de `x`.
mettre `x` à zéro : `*p = 0`
augmenter `x` de `y` : `*p += y`

7.5 Opérateurs sur les pointeurs

Un pointeur pointe sur des objets d'un type donné. Il existe cependant une valeur commune à tous les pointeurs : la valeur `NULL`, définie dans le fichier `stdio.h`, et qui est une valeur de pointeur ne pointant sur rien.

Les affectations de pointeurs ayant même type d'objets pointés sont possibles ; si les objets pointés n'ont pas même type, ou s'ils ont même type mais que ce type n'est pas connu (à l'intérieur d'une fonction compilée séparément par exemple), on doit utiliser un *forceur*.

Les opérations arithmétiques suivantes sont définies sur les pointeurs :

- addition d'un entier : correspond à un déplacement dans le type pointé ; l'entier est d'abord multiplié par la taille d'un élément du type pointé avant l'addition
- soustraction d'un entier : même interprétation que l'addition
- soustraction de deux pointeurs : fournit le nombre d'éléments situés entre les deux pointeurs
- comparaison de deux pointeurs.

7.6 Type d'un pointeur universel

Il est quelquefois nécessaire d'avoir des pointeurs d'un type non déterminé. Un tel pointeur est de type prédéfini **void** * :

void * *pu* ;

déclare un pointeur `pu` pointant sur un type indéterminé.

7.7 Pointeurs et indexation

L'indexation est une forme particulière de l'indirection. Ainsi, l'expression indicée :

```
expression1 [expression2]
```

est calculée comme étant :

```
*(expression1 + (expression2))
```

le plus étant le plus sur les pointeurs (multiplication préalable de `expression2` par la taille des éléments pointés, donc par la taille des éléments du tableau), et `expression1` étant de type pointeur (un identificateur de tableau a pour valeur l'adresse du tableau).

Ainsi, l'expression `t[i]` est équivalente à l'expression `*(t+i)`, et `t[i]` est effectivement transformée par le compilateur en `*(t+i)`. Il en est de même pour un tableau à plusieurs dimensions. Soit

```
typedef  ... T;          /* type T de taille n */
T gt[P][Q];
   /* tableau gt de P tableaux de Q éléments de type T */
```

Alors, `gt[i][j]` est évalué comme `*(gt[i]+j)` ; comme `gt[i]` est un tableau (de `Q` éléments), le signe + est l'opérateur additif sur les pointeurs, et `j` est multiplié par la taille d'un élément de `gt[i][j]`, soit `n`, avant l'addition. Puis `gt[i]` est lui même évalué comme `*(gt+i)`, et `i` est multiplié par `Q*n`, puisque `gt[i]` est un élément de taille `Q*n`. L'expression `*(gt+i+j)` a donc pour valeur la valeur de l'élément d'adrese `gt+Q*n*i+n*j`.

7.8 Déclaration de pointeur et déclarateur de tableau

Il ressort de l'exposé précédent que les notions de pointeurs et de tableaux sont très proches l'une de l'autre. En particulier, les deux déclarateurs suivants :

```
char t[];
char *P;
```

déclarent tous les deux une variable pointeur sur un caractère. Si l'on fait pointer ces variables sur des chaînes (dont le premier élément entre autre est bien de type caractère) :

```
t = "une";
P = "deux";
```

alors on peut utiliser indifféremment la notation indexée ou l'adressage indirect pour référencer un élément de ces chaînes :

`t[1]` vaut '`n`', `*t` vaut '`u`'
`p[2]` vaut '`u`', `*p` vaut '`d`'

On fera bien la différence cependant entre la déclaration d'un tableau : réservation d'un certain nombre d'éléments, et la déclaration d'un pointeur, qui n'alloue pas les éléments. On utilisera un déclarateur de pointeur quand on ne connait pas la taille du tableau :

```
char *ch;          /* simple pointeur sur une chaîne */
   ...
ch = "longueur ligne à calculer!";
   /* le pointeur pointe désormais sur une chaîne */
```

On peut déclarer également :

```
char ch[] = "longueur...";
   /* ch est un pointeur qui pointe sur la chaîne... */
```

Exemple :

Soient deux chaînes `c1` et `c2`. On veut recopier `c1` dans `c2`.

```
char c1[N]; char c2[M];
   ...
{  int i = 0; /* version1 */
     do { c2[i] = c1[i]; i += 1;
        } while c2[i] != '\0';
}

/* en utilisant un tantque, : */
{  int i = 0; /* version2 */
   while ((c2[i] = c1[i]) != '\0') i += 1;
}
/* le test != '\0' est redondant puisqúil signifie
```

```
   "différent de faux"*/

{  int i = 0 ; /*  version3 */
   while (c2[i] = c1[i]) i += 1 ;
}

/*  en considérant c1 et c2 comme des pointeurs : */
{  while (*c2 = *c1) {c2 += 1 ; c1 += 1}
}
/*  cette version 4 est séduisante, mais elle plante ici car c1 et c2
   sont des constantes (adresse). Elle convient si c1 et c2 sont
   des paramètres seulement ! */
```

7.9 Les opérateurs d'incrémentation et de décrémentation

Les opérations qui consistent à ajouter une unité à une variable étant très fréquentes, C a introduit deux opérateurs pour réaliser cette opération. Appliqués à des pointeurs, ajouter un s'entend ajouter une fois la taille des objets pointés.

RÈGLE
expression-d'incrémentation :
$\{ ++ \mid -- \}$ identification | identification $\{ ++ \mid -- \}$

Ces opérateurs ont une priorité identique à l'étoile de l'indirection (et une associativité de droite à gauche) et sont donc moins prioritaires que les crochets. Leur sémantique est la suivante :

`++x` s'évalue en additionnant un à `x`, puis délivre la nouvelle valeur de `x`.

`x++` délivre la valeur de `x`, puis additionne un à `x`.

Ces opérateurs s'utilisent surtout avec des pointeurs . On fera très attention aux priorités et aux parenthésages.

Exemple :

- la boucle de la version 4 de l'exercice précédent peut s'écrire :

  while `(*c2++ = *c1++)` ;

  L'associativité étant de droite à gauche, le ++ porte sur `c2` (ou `c1)` : c'est donc `c2` (ou`c1)` qui sera augmenté de un, mais pas avant d'avoir utilisé la valeur actuelle de `c2` (ou `c1)`, pour faire une affectation indirecte.
- considérons maintenant l'expression `(*x)++` dans laquelle `x` est un pointeur. Ici, le ++ porte sur `*x` puisqu'il y a des parenthèses. Cette expression délivre donc la valeur `*x` puis `*x` est augmenté de un.

7.10 Tableaux et pointeurs

On peut optimiser l'utilisation des tableaux avec des pointeurs, en utilisant le fait que le nom d'un tableau est l'adresse de son premier élément. Ainsi, on a déjà vu que `tab[i]` et `*(tab+i)` étaient équivalents.

Considérons par exemple la boucle suivante :

for `(i = 0; i < N; i++) tab[i] = X;`

On peut l'écrire aussi :

```
for (i = 0; i < N; i++) *(tab + i) = X;
```

En utilisant un pointeur sur le début du tableau, on a :

```
for (ptab = tab, i = 0; i < N; i++) *ptab++ = X;
```

Mais `i`, qui ne sert plus qu'à tester la fin, peut être supprimé :

```
for (ptab = tab; ptab < tab + N; ptab++) *ptab = X;
```

On peut sortir les deux expressions invariantes de la boucle, qui devient le **while** suivant :

```
ptab = tab; fin = tab + N;
while (ptab < fin) *ptab++ = X;
```

7.11 Exercices

1. Reprenez l'exemple de l'accès à un entier par indirection du paragraphe 7.4. Déclarez un pointeur `pp` pour accéder à l'entier `x` par une double indirection. Écrivez un programme qui montre que vous pouvez modifier `x` par double indirection (via `pp`).

2. Écrivez un programme qui affiche les indices des valeurs nulles du tableau d'entiers `tab`, initialisé à sa déclaration, et sans déclarer de variable de type entier.

Chapitre 8

LES FONCTIONS

On n'aborde pas dans ce chapitre les problèmes de la compilation séparée. On rappelle que la notion de sous-programme est réalisée en C par la notion de fonction.

8.1 Déclaration de fonction

Comme tout objet, une fonction se déclare avant de s'utiliser. Cependant, les fonctions ne peuvent être emboîtées, et elles se déclarent toutes au même niveau que la fonction ***main*** qui sert de programme (et qui peut être absente en cas de compilations séparées).

On a le choix entre deux syntaxes pour déclarer une fonction : la syntaxe de la norme C, et la vieille syntaxe, qui prévaut encore dans la majorité des ouvrages sur C.

RÈGLE
déclaration-de-fonction :
[classe] [type] déclarateur ([liste-de-paramètres])
[déclaration-de-paramètres] bloc |
[classe] type déclarateur ([déclaration-des-paramètres]) bloc

La première ligne correspond à la vieille déclaration : le type peut être omis, et les paramètres sont déclarés en deux temps. Dans ce cas là, la liste de paramètres est une simple énumération des identificateurs des paramètres formels, séparés par une virgule.

Une déclaration de paramètre est analogue à une déclaration de variable sans classe, avec cette différence que les paramètres tableaux à une dimension sont déclarés comme des pointeurs (*p ou p[ ]). La déclaration des paramètres est une liste de déclarations de paramètres séparées par des points virgules dans la vieille version, et séparées par des virgules dans la norme.

Exemple :

```
/* fonction sans paramètre qui ne délivre rien */
/* anciennement : */
    proc1()
      { <son bloc> }
/* selon la norme : */
     void proc1( void)
      { <son bloc> }

/* fonction qui délivre un entier et qui a trois paramètres :
  deux entiers et un tableau de caractères */
/* anciennement : */
    proc2(n, m, t)
```

```
      int n, m;  char t[];
     { <son bloc> }
/* selon la norme : */
     int proc2( int n,  int m,  char t[])
      { <son bloc> }
```

8.2 Visibilité des objets

Sont visibles dans une fonction :

- ses paramètres formels
- ses variables locales
- les variables globales locales au module qui déclare la fonction (variables déclarées en dehors des fonctions, mais dans la même entité de compilation)
- les variables globales exportées par d'autres modules et importées par le module qui déclare la fonction. Ces deux derniers cas sont étudiés dans le chapitre sur la compilation séparée (paragraphe 9.3.1).

Les règles habituelles ont cours : un identificateur local homonyme d'un identificateur plus global cache cet identificateur global. Les variables locales et les paramètres formels des fonctions sont alloués automatiquement à l'activation de la fonction, et disparaissent quand la fonction se termine.

8.3 Valeur délivrée par une fonction

Une fonction se termine lorsque son exécution arrive sur l'accolade fermante de son bloc. Si arrivée là, aucune instruction **return** n'a été exécutée, la fonction délivre une valeur indéfinie (ce peut être normal dans le cas d'une fonction de type **void** qui s'utilise comme une procédure). La valeur délivrée par une fonction est évaluée par une instruction de retour.

Les types des fonctions sont les types de bases, les types pointeurs, et, depuis la norme, les types structures (et bien sûr les types définis qui sont de ces types-là).

RÈGLE
instruction-retour :
return [expression] ;

Sans expression, l'instruction de retour termine l'exécution de la fonction sans délivrer de valeur ; avec une expression, elle termine l'exécution et délivre la valeur de l'expression.

8.4 Mode de transmission des paramètres

Il y a un seul mode de transmission de paramètre en C, le mode *par valeur*. Le paramètre formel est donc une variable locale qui est initialisée par une affectation avec le paramètre effectif (une expression), au moment de l'activation de la fonction.

Exemple :

```
void lignede( int nb,  char car)  /* affiche nb caractères car */
  {  for ( ; nb > 0; --nb) putchar(car);
  }
```

Comme *nb* est une variable locale, on peut détruire sa valeur.

Exemple :

```
int max( int a, int b)  /* max de a et de b */
  {  return (a > b )? a : b;
  }

void AfficherVecteur( float v[], int n)
    /* affiche sur une ligne les n valeurs du vecteur v */
  {  int i;
     for (i = 0; i < n; ++i) printf("%6.2f", v[i]);
    printf("\n");
  }
```

Lorsqu'un paramètre est un tableau, comme dans l'exemple prédédent, le paramètre formel reçoit l'adresse du tableau au moment de l'appel. Comme cette adresse est aussi une valeur de pointeur sur des réels, la notation *float ** pour le paramètre *v* est équivalente à *float v[]*. Si le tableau a plusieurs dimensions, le compilateur a besoin de connaitre la taille de la (des) dernière(s) dimensions pour mener à bien ses calculs d'adresses (voir le paragraphe 7.7).

Exemple :

```
void AfficherMatrice( float m[N][N])  /* ou float m[][N] */
    /* affiche la matrice m, N étant une constante */
  {  int i, j;
     for (i = 0; i < N; ++i)
       {  for (j = 0; j < N; ++j) printf("%6.2f", m[i][j]);
         printf("\n");
       }
  }
```

On peut donner à présent la grammaire de la déclaration des paramètres.

RÈGLE
déclaration-des-paramètres :
  type [déclarateur] [, type déclarateur]* [, ...]

Le cas des points de suspension indique que le nombre des paramètres effectifs peut être variable. La fonction doit avoir un moyen de connaitre ce nombre.

> **Exemple :**La fonction *printf* est déclarée dans la bibliothèque par
>
> **int** *printf*(**char** * *format*, ...)
>
> et l'on sait que le format contient autant de spécifications de format que d'expressions placées en paramètre effectif.

L'absence de déclarateur correspond au cas où il n'y a aucun paramètre (le type est **void**).

RÈGLE
déclarateur :
  identificateur | * déclarateur |
  (déclarateur) | déclarateur [[exp.constante]]

8.5 Les paramètres résultat

La transmision par valeur des paramètres correspond bien à l'idée d'une fonction qui ne réalise pas d'effet de bord. Mais on sait bien qu'on a souvent besoin d'écrire un sous-programme qui transforme la valeur d'un objet.

On peut remarquer tout d'abord que le problème est résolu en ce qui concerne les tableaux : comme le paramètre formel est initialisé avec l'adresse du tableau effectif, toute manipulation sur le paramètre formel équivaut à une manipulation sur le paramètre effectif.

Exemple :

```
void LireVecteur( float v[], int n)  /* lit les n valeurs de v */
  {  int i ;
     for (i = 0; i < n ; ++i) scanf("%f", &v[i]) ;
  }
```

Par contre, une variable simple qu'on veut modifier par un sous-programme ne peut être passée par valeur, puisque c'est sa valeur avant l'appel qui est copiée dans le paramètre effectif : on doit donc transmettre son adresse, et indiquer dans la déclaration de paramètres que le paramètre est un pointeur.

Exemple :

```
void permuter( int *x, int *y)   /* permute x et y par indirection */
  {  int aux ;
     aux = *x ; *x = *y ; *y = aux ;
  }
     /* appel : permuter(&a, &b); pour permuter a et b */
```

8.6 Paramètres tableaux et pointeurs

Reprenons sous la forme de sous-programme la copie d'une chaîne dans une autre ; dans une version traditionnelle, nous écrirons :

```
void CopieChaine( char ch1[], char ch2[])
  {  int i = 0 ;
     while ((ch1[i] = ch2[i]) != '\0') ++i ;
  }
```

La version avec pointeur est plus concise :

```
void CopieChaine( char ch1[], char ch2[])
  {  while (*ch1++ = *ch2++) ;
  }
```

Dans cette version, la valeur du pointeur sur les deux chaînes, transmise en entrée, est augmentée à chaque tour de boucle pour pointer successivement sur toutes les cases de `ch2`. L'utilisation des pointeurs doit se faire néanmoins avec une grande vigilance. Reprenons le sous-programme `LireVecteur`, et transformons-le (sans précaution) en une fonction délivrant un vecteur.

```
float * LireVecteur( int n)          /* Attention, erreurs */
      /* Lit et délivre un vecteur à n entrées */
  {  float v[] ;
     for ( ; n > 0 ; --n) scanf("%f", &v[n]) ;
     return v ;
  }
```

Apparemment, la programmation est classique : déclaration d'un tableau local de taille inconnue `v`, puis lecture de ce tableau, et délivrance du tableau local. Il y a deux grosses erreurs :

- la déclaration **float** `v[ ]` ne déclare pas un tableau, mais seulement un pointeur sur un éventuel tableau de flottants. Le compilateur ne dira rien, et l'exécution de la fonction risque même de bien se passer.
- l'instruction retour délivre la valeur du pointeur `v`, qui, si l'exécution s'est effectivement bien passée (c'est à dire, si on a pu sans dommage empiler n réels non prévus sur la pile d'exécution), est une variable locale qui disparait à la fin de l'activation de la fonction.

Si l'on voulait effectivement retourner la valeur du pointeur, on devrait déclarer le tableau `v` avec la classe d'allocation `static`.

8.7 Les fonctions en paramètre

On ne peut pas passer une fonction en paramètre. Par contre, on peut transmettre un pointeur sur une fonction (de même qu'on peut fabriquer un tableau de pointeurs sur des fonctions). Prenons un exemple tiré de la bibliothèque C.

La bibliothèque standard (*stdlib.h*) contient une fonction de tri de tableau par le tri rapide de Hoare, déclarée par :

```
void qsort( void *arr,  int n,  int size,
           int (*comp_fn)( void*,  void*)) ;
```

qui trie le tableau `arr` (**void *** : tableau de n'importe quoi), de longueur `n`, dont les éléments font `size` caractères, et dont la relation d'ordre est définie par le pointeur `comp_fn` sur une fonction à deux paramètres pointeurs quelconques et délivrant un entier. La fonction de comparaison admet comme paramètres deux éléments `e1` et `e2` du tableau, et délivre un entier négatif si `e1` $<$ `e2`, nul si `e1` = `e2`, et positif si `e1` $>$ `e2`. Pour utiliser `qsort` pour comparer un tableau `t` de `n` entiers, on déclarera la fonction :

```
int comp_int( void *e1,  void *e2)  /* pour être conforme à la
                                       déclaration de comp_fn */
  {  int premier = *( int *)e1 ;     /* forceur obligatoire ! */
     int second = *( int *)e2 ;
     return premier - second ;
  }
```

et on appelera `qsort` ainsi :

```
qsort (t, n, sizeof(int), comp_int) ;
```

On remarquera deux choses :

- on n'a pas mis de & devant le paramètre effectif *comp_int* : le compilateur trouve lui même qu'il faut transmettre un pointeur sur *comp_int* ;
- dans *qsort*, on a déclaré le paramètre **comp_fn* par un prototype de fonction qu'on verra dans le chapitre sur la structuration des programmes.

8.8 Les arguments de *main*

Lorsque la fonction *main* est appelée, l'interpréteur du langage de commande lui transmet les arguments de la ligne de commande. Pour les récupérer dans le programme, il faut déclarer deux ou trois paramètres selon les besoins :

- le premier, *argc*, est un entier qui dénombre les arguments transmis.
- le second, *argv*, est un tableau de pointeurs sur des chaînes, tel que *argv[0]* pointe sur l'argument 0 (le nom de l'exécutable appelé), *argv[argc - 1]* pointe sur le dernier argument et *argv[argc]* vaut *NULL*.
- le troisième, *env*, est un tableau de pointeurs sur des chaînes de la forme "nom = valeur", tableau qui donne les noms et valeurs des variables de l'environnement.

Exemple :

```
/* Engendre un fichier de nom <parametre 1> qui contient
  <parametre 2> lignes de <parametre 3>
    Enregistré dans le fichier "fichierde"
*/

#include <stdio.h>
 void main ( int argc,  char *argv[])
  { FILE *f ;             /* fichier f */
    int i ;  int N ;
    if ( argc == 3)
    { N = atoi(argv[2]) ;  /* atoi transforme une chaine en entier */
      f = fopen(argv[1], "w") ;  /* ouverture en écriture de f */
       for (i = 0 ; i < N ; ++i)   /* remplissage des N lignes */
        fprintf(f, "%s\n", argv[3]) ;
      fclose(f) ;
    }
    else printf("Donnez 3 paramètres SVP\n") ;
  }
```

On pourra appeler ce programme par

```
fichierde etoile 4 \*
```

pour fabriquer le fichier *étoile* de 4 lignes contenant chacune une étoile. Voir paragraphe 11.4.1 pour l'ouverture du fichier *f*.

8.9 Exercices

1. Anticipant légèrement sur le chapitre suivant, on vous demande d'écrire un module de gestion rudimentaire du temps. Le temps y est matérialisé par trois variables globales : *heures* pour l'heure, *minutes* pour les minutes, et *secondes* pour les secondes. Le module comprend les 3 opérateurs suivants :

- la procédure **`Afficher_heure`** qui affiche (correctement) *Il est ... heure(s) ... minute(s) ... seconde(s)* ;
- la procédure **`Etablir_heure`** qui a 3 paramètres h, m, s pour initialiser l'heure ;
- la procédure **`Tic`** qui fait avancer l'heure d'une seconde.

Vous écrirez un programme pour tester ce module. Note : le module consiste en un fichier qui déclare l'heure en variable globale, puis les 3 opérateurs. On y incluera ici le programme principal.

2. On veut crypter un texte selon le principe du décalage des lettres : tout caractère qui n'est pas une lettre est inchangé, toute lettre est *décalée* dnas l'alphabet d'une distance égale à la clé de cryptage. Vous écrirez :
   - une procédure qui admet en premier paramètre la clé du cryptage et qui crypte le caractère transmis en deuxième paramètre ; le deuxième paramètre sert en entrée comme en résultat ;
   - une procédure qui admet en premier paramètre la clé du cryptage et qui crypte le texte transmis en deuxième paramètre dans un tableau ;
   - un programme qui déclare un texte (global), le crypte, et affiche le résultat du cryptage.

Chapitre 9

STRUCTURATION DES PROGRAMMES

La composition de déclarations de type, de variables et de fonctions permet de fabriquer des modules. Un programme peut se composer de la seule fonction ***main***, d'un module comportant des sous-programmes et la fonction ***main***, ou utiliser plusieurs modules compilés séparément.

9.1 Composition d'un module

Un module est une entité de compilation.

RÈGLE
module :
[directive-au-pré-compilateur]* [déclaration-de-type]*
[déclaration-de-variable]* [déclaration-de-prototype]*
[déclaration-de-fonction]*

Le langage n'impose aucun ordre dans les déclarations, contrairement à la règle précédente, mais si l'on fait référence à un objet déclaré plus loin, le compilateur risque de prendre des initiatives (du genre type par défaut) désastreuses.

Les directives au pré-compilateur servent à inclure des fichiers (de bibliothèque ou non), à définir des constantes ou des macro-instructions (voir paragraphe 12).

Les déclarations de type ont une portée qui est le module ; elles ne sont pas exportables. Pour exporter des types, on crée en général un fichier comportant des déclarations de types, et on l'inclut en tête de chaque module qui en a besoin par une directive au pré-compilateur.

Les déclarations de variables et de fonctions peuvent définir des objets locaux ou globaux.

9.1.1 Variables locales du module

Une variable locale au module est de la classe d'allocation **static**. Après la déclaration :

static int `t[100]` ;

le tableau `t` est une variable globale vis-à-vis des fonctions définies dans le module, mais c'est une variable locale au module en ce sens qu'elle est inaccessible depuis les autres modules.

9.1.2 Variables globales

Toute variable déclarée en dehors d'une fonction et qui n'a pas d'attribut de classe est une variable globale : elle est automatiquement exportée par le compilateur.

9.1.3 Variables externes

Pour utiliser dans un module `A` une variable globale d'un module `B`, il faut l'importer explicitement. Celà consiste à déclarer localement à `A` la variable globale de `B` et à lui donner la classe **extern** : la variable ne sera pas localisée dans le module `A`, mais elle sera accessible depuis `A`.

9.1.4 Fonctions locales au module

Comme pour les variables locales du module, les fonctions locales ont l'attribut **static** ; elles ne sont pas accessibles depuis les autres modules.

9.1.5 Fonctions globales

Toutes les fonctions déclarées dans un module sans attribut de classe sont *a priori* des fonctions globales que tout module peut utiliser à condition de l'importer.

9.1.6 Fonctions externes

Pour utiliser dans un module `A` une fonction globale définie dans un module `B`, il faut l'importer explicitement. Cependant, cette importation ne se fait pas de la même façon que pour les variables externes. Le compilateur doit en effet connaitre non seulement le type de la fonction, mais aussi le nombre et le type de ses paramètres. L'importation consiste alors à donner une déclaration du *prototype* de la fonction, sans lui mettre l'attribut de classe **extern** (mais on peut le mettre quand même).

9.2 Prototype de fonction

Un prototype de fonction est analogue à une déclaration de fonction dans laquelle le bloc est remplacé par un point virgule ; de plus, le nom des paramètres formels peut être omis.

RÈGLE

déclaration-de-prototype :

[**extern**] type déclarateur (type déclarateur-abstrait [, type déclarateur-abstrait]*) ;

déclarateur-abstrait :

identificateur | * déclarateur-abstrait | (déclarateur-abstrait) | déclarateur-abstrait [[expression-constante]] | déclaration-de-prototype

Exemple :

```
/* Les prototypes des fonctions lignede et max du paragraphe 8.4
   peuvent s'écrire : */
void lignede( int n, char ch);
int max( int a, int b);
```

Le nom du paramètre formel n'est pas utilisé par le compilateur, et il peut être omis, contrairement à ce qui se passe en Pascal :

```
void lignede( int, char);
int max( int, int);
```

version qui semble préférable.

Il est à noter que les anciens compilateurs (hors norme) n'acceptent aucune déclaration de paramètre dans les prototypes. Il aurait fallu écrire :

```
void lignede();
int max();
```

ou même simplement :

```
lignede();
max();
```

puisque le type par défaut est le type **int** (mais ce genre d'écriture est à proscrire...).

9.3 Ordre des déclarations dans un module

Théoriquement, l'ordre est quelconque : on peut commencer par déclarer la fonction **main**, puis les autres fonctions, du moment que toute variable référencée a été précédée de sa déclaration. En fait, un tel module, organisé sans précaution, ne fonctionnera correctement que si toutes les fonctions sont de type **int** ou **void** : lorsque le compilateur compile un appel de fonction, deux cas se produisent :

- ou il connait déjà la fonction, parce que le module contient plus haut soit la déclaration de la fonction, soit la déclaration du prototype de la fonction ;
- ou il ne connait pas encore la fonction, parce qu'elle est définie plus loin, ou parce qu'elle n'est pas définie du tout (fonction externe).

Dans le premier cas, le code de récupération de la valeur de la fonction sera correct. Dans le second, le compilateur engendrera une récupération d'un entier, car il suppose que la fonction viendra plus tard, et il prendra la valeur par défaut pour un type de fonction. Concrètement, l'ordre devrait être le suivant :

- le commentaire sur le module
- les directives au pré-compilateur
- les déclarations des variables globales et les déclarations des variables externes (importées)
- les déclarations des prototypes de toutes les fonctions utilisées dans le module (importées ou non)
- les déclarations des fonctions, dans n'importe quel ordre.

9.3.1 Composition d'une fonction

Le bloc d'une fonction peut contenir des déclarations de variables et des déclarations de prototype. Les variables déclarées dans une fonction peuvent être :

- locales (classe **auto**, classe par défaut)
- rémanentes (classe **static**), c'est à dire qu'elles conservent leur valeur d'un appel à l'autre (ce sont des variables allouées avec les variables locales du module)
- externes (classe **extern**).

Si une fonction est seule dans son module à importer telle variable globale et telle fonction externe, il est conseillé d'y placer les déclarations d'externes et de prototype correspondant plutôt que de les placer en tête du module (voir l'exemple du paragraphe 10.4).

9.3.2 Résumé sur l'organisation des données

Le tableau de la figure 9.1 synthétise les durées de vie et les visibilités des variables en fonction de leur localisation, de leur classe et de leur sorte.

Sorte de variable	Localisation	Classe	Durée de vie	Visibilité
variable globale	module		programme	module
variable de module	module	**static**	programme	module
variable locale	fonction	**auto** ou rien	bloc	bloc
variable rémanente	fonction	**static**	programme	bloc
variable importée	module	**extern**	programme	module
variable importée	fonction	**extern**	programme	bloc

FIG. 9.1: Les sortes de variables

9.3.3 Attribut des variables

La classe d'une variable locale, omise la plupart du temps puisque c'est **auto** par défaut, peut prendre une valeur d'attribut. Il y a trois attributs possibles :

register cet attribut est une sorte de pragma qui indique au compilateur qu'on souhaiterait que la variable en question soit allouée dans un registre. L'opérateur de mise d'adresse (&) est interdit sur une variable qui a cet attribut.

const Cet attribut indique au compilateur que la valeur initiale obligatoirement fournie lors de la déclaration ne variera pas, et donc qu'il peut optimiser en conséquence.

volatile Cet attribut indique au compilateur que la variable "consomme" sa valeur à chaque affectation, et lui interdit de faire les optimisations classiques. L'exemple classique (le seul ?) est celui d'un pointeur de caractère qui désigne un port d'entrée-sortie : si l'on envoie successivement deux commandes sur le port, il serait regrettable que le compilateur supprime la première affectation sous prétexte qu'elle sera annihilée par la seconde ; l'exemple qui suit fonctionne sur une machine type 68000, où les entrées-sorties se font par affectation des ports (il ne fonctionnerait pas sur un 80x86, qui a des instructions particulières d'entrées-sorties).

```
volatile char *port_de_commande;
 ...
*port_de_commande = 0x1e;  /* reset */
*port_de_commande = 0x20;  /* autre commande */
```

L'attribut **register** peut également être donné à un paramètre formel (déclaré selon l'ancienne méthode).

9.4 Exercice

Écrivez un module qui gère une pile d'éléments de type tableau `chaine` (de caractères) défini dans le module. Il exporte :

- le type `chaine` ;
- le prédicat `pilevide` ;

- le constructeur *empiler(ch)* qui empile *ch* ;
- l'accesseur *sommet* qui délivre la chaîne en sommet ;
- le modificateur *depiler(ch)* qui extrait *ch*.

Par ailleurs, vous écrirez un programme ne faisant pas partie du module, qui lit une phrase au clavier, et qui la réaffiche à l'envers.

Chapitre 10

LES ARTICLES

Les articles en C s'appelent des *structures*. Un objet de type structure est un objet composé de plusieurs champs qui peuvent être de type différent et qui sont dénotés par un identificateur.

10.1 Le type structure

Chaque occurrence d'un type structure est précédée du mot-clé **struct**.

RÈGLE
type structure :
struct [identificateur] [{ [déclaration-de-champ ;]$^{+}$ }]

L'identificateur est le nom du type ; s'il est omis, il s'agit d'un type anonyme. Une déclaration de champ est analogue à une déclaration de variable sans classe. On a donc, dans une première approche :

RÈGLE
déclaration-de-champ :
type déclarateur [, déclarateur]*

Exemple :

```
struct date          /* déclaration du type struct date */
  { short j, m;
    int a;
  };
typedef enum {lun, mar, mer, jeu, ven, sam, dim} JOUR;
typedef struct
  { JOUR j;
    struct date d;
  } UNE_DATE;
```

La première déclaration définit le type **struct** *date* composé de trois entiers de tailles différentes, la dernière définit le type *UNE_DATE* comme étant le type (anonyme) structure composée d'un champ de type énuméré et d'un champ de type **struct** *date*.

10.2 Opérateurs sur les types structures

Depuis la norme C, on peut affecter deux articles de même type, et donc passer un article en paramètre.

L'autre opérateur permet la sélection d'un composant, par la notation pointée habituelle. Le dernier opérateur est présenté plus loin.

RÈGLE
expression-champ-d'article :
expression.identificateur-de-champ

Exemple :

```
UNE_DATE aujourdhui, demain ;  /* cf paragraphe précédent */
aujourdhui.j = mar ;
aujourdhui.d.a = 1989 ;
aujourdhui.d.m = 5 ;
aujourdhui.d.j = 9 ;
```

10.3 Articles en paramètre

L'opérateur point est plus prioritaire que l'opérateur de prise d'adresse : par contre, l'opérateur d'indirection et l'opérateur de prise d'adresse ont même priorité, et sont associatifs de droite à gauche. D'où la procédure de lecture suivante :

```
void LireDate( struct date *d)  /* lit jj-mm-aa */
  { scanf("%hd %*c %hd %*c %d", &(*d).j, &(*d).m, &(*d).a) ;
  }

  /* Appel possible : */

  LireDate(&aujourdhui.d) ;
```

10.4 Fonctions de type article

Depuis la norme C, une fonction peut être de type article. L'exemple suivant est une fonction qui a en paramètre une date et qui calcule la date du lendemain.

Exemple :

```
UNE_DATE lendemain(UNE_DATE aujourdhui)
  { UNE_DATE demain ;
     int nbjoursdumois( struct date d) ;
                    /* délivre le nombre de jours du mois d.m */
    JOUR nomdusuivant(JOUR j) ;  /* délivre le nom du suivant de j */

     if (aujourdhui.d.j != nbjoursdumois(aujourdhui.d))
      { /* simplement passer au jour suivant */
        demain.d.j = aujourdhui.d.j + 1 ;
        demain.d.m = aujourdhui.d.m ;
        demain.d.a = aujourdhui.d.a ;
      }
     else if (aujourdhui.d.m == 12)
      { /* changement d'année */
        demain.d.j = 1 ; demain.d.m = 1 ;
        demain.d.a = aujourdhui.d.a + 1 ;
```

```
      }
     else  /* changement de mois */
      { demain.d.j = 1; demain.d.a = aujourdhui.d.a;
        demain.d.m = aujourdhui.d.m + 1;
      }
    demain.j = nomdusuivant(aujourdhui.j);
     return demain;
  }

JOUR nomdusuivant(JOUR j)
  { return (JOUR)(((int)j + 1) % 7);
  }

int nbjoursdumois(struct date d)
       /* délivre le nombre de jours du mois d.j */
  { int anneebissextile(int annee);   /* vrai ou faux */
     short nbjours[12] =
      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

     if (d.m == 2 && anneebissextile(d.a))
       return 29;
     else return nbjours[d.m - 1];
          /* tableau indexé à partir de zéro */
  }

int anneebissextile(int annee)
  { return ((annee % 4 == 0 && annee % 100 != 0)
           || annee % 400 == 0);
  }
```

10.5 Initialisation des articles

On peut, comme pour les tableaux, initialiser les articles lors de leur déclaration : les valeurs fournies iront dans les premiers champs. Par exemple :

```
UNE_DATE d = {lun, {3, 5}};
```

initialise le nom du jour, le numéro du jour et le mois de la variable `d`

10.6 Pointeurs sur les articles

Avant la norme C, les affectations de structures étaient interdites : on était amené à déclarer de nombreux pointeurs sur des structures (paramètres, valeurs des fonctions). La notation d'accès à un champ `c` d'un article repéré par le pointeur `p` est lourde, rappelons là : c'est `(*p).c`. Cette notation est simplifiée grâce à l'opérateur –>, beaucoup plus lisible : `(*p).c` est équivalent à `p->c`.

RÈGLE
expression-champ-d'article :
expression.identificateur-de-champ | expression–>identificateur-de-champ

Dans le premier cas, l'expression doit désigner un article ; dans le deuxième cas, l'expression doit désigner un pointeur sur un article.

La procédure *Liredate* des paragraphes précédents peut s'écrire alors :

```
void Liredate( struct date *d)
  { scanf("%hd %*c %hd %*c %d", &d->j, &d->m, &d->a)
  }
```

10.7 Taille d'un article : opérateur sizeof

L'opérateur **sizeof** délivre la taille en octet occupée par une variable (locale ou globale, mais pas un paramètre formel ni une variable externe), par les objets d'un type, ou par une expression. Attention, bien que sa syntaxe semble l'assimiler à une fonction, c'est bien un opérateur, et qui de plus est évalué à la compilation.

RÈGLE
expression-de-taille :
sizeof expression | **sizeof** (identificateur-de-type)

On ne peut pas faire l'hypothèse que **sizeof**(*article*) soit la somme des taille de chaque champ de *article*, à cause des éventuels problèmes d'alignement.

Exemple :

sizeof (*UNE_DATE*) est une constante qui est le nombre d'octets occupés par un objet de type *UNE_DATE*.
sizeof *nbjours* / **sizeof** (**short**) est une constante qui vaut 12.

10.8 Les articles tassés

Les articles vus dans les paragraphes précédents sont alloués en fonction du type des champs : un booléen déclaré en **int** occupera un mot. Il est possible de demander une allocation au bit près, avec la notation dite de champ de bits. La syntaxe complète de la déclaration de champ permet d'indiquer par une expression constante la taille en bit du champ.

RÈGLE
déclaration-de-champ :
type déclarateur [, déclarateur]* |
type [identificateur] : expression [, type [identificateur] : expression]*

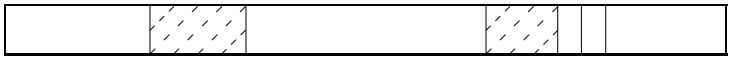
Le type d'un champ de bit ne peut être qu'entier ; le type **signed int** peut avoir des effets bizarres, aussi utilise-t-on pratiquement toujours le type **unsigned int**. Sans identificateur, le champ de bit sert à combler une zone de bits non utilisée : en effet, les champs sont alloués en séquence, sans modification de l'ordre donné dans la structure. Une constante 0 indique que l'on veut cadrer le champ suivant sur une frontière adressable (l'octet sur une machine à octets, le mot sur une machine à mots,...)

Exemple :

```
struct entree
  { unsigned int type :4;    /* entier sur 4 bits */
    unsigned int vide :1;    /* un booléen */
    unsigned int trou :1;    /* un booléen */
```

```
    unsigned int :0;          /* on saute jusqúà la frontière */
    short int index;          /* champ non tassé */
    unsigned int :3;          /* trou inutilisé de 3 bits */
    unsigned int sens :5;     /* entier sur 5 bits */
}
```

Une telle structure pourrait être implantée comme ceci, sur trois octets :



mais elle pourrait tout aussi bien être implantée dans l'autre sens (le champ *type* d'abord...). Les opérations sur les champs de bits sont les opérations sur les entiers, mais l'opérateur de prise d'adresse n'est pas autorisé : on n'a pas non plus de pointeurs sur les champs de bits.

10.9 Les unions

Une union permet de faire des équivalences de type, c'est à dire qu'elle permet de considérer une même zone mémoire selon différentes interprétations. La syntaxe d'une union est la même que celle d'un article, en remplaçant **struct** par **union**. La variable *mot* suivante :

```
union unmot
  { int entier;
    char car[4];
    void * p;
  } mot;
```

est allouée sur un seul mot machine (on suppose une machine 32 bits ici). Si l'on fait référence à *mot.entier*, on obtient une valeur interprétée comme étant un entier, tandis que *mot.car[1]* sera interprété comme étant un caractère et *mot.p* comme étant un pointeur de n'importe quoi. Contrairement à Pascal, rien n'indique quelle est l'interprétation valide à un instant donné.

Si les différents champs n'ont pas la même taille, la zone allouée permet d'implanter le plus grand. En cas d'initialisation lors de la déclaration la valeur doit être du type du premier champ :

```
union unmot m = 25;  /* = "abcd" serait refusé */
```

10.10 Exercice

Reprenez le module du chapitre précédent : il déclare *une* pile et ses opérateurs ; transformez-le pour qu'il déclare un *type abstrait* pile. Dans ces conditions, "la" pile du module n'existe plus, seul son *modèle* existe : c'est l'utilisateur qui déclare la variable pile.

Le type *PILE* est bien entendu un article. Comme on ne peut pas initialiser les champs d'un type article, vous ajouterez un opérateur au type *PILE*, *init*, pour initialiser la pile à vide. Tous les opérateurs du module ont un nouveau (et premier) paramètre : "la" pile sur laquelle ils opèrent. Reprenez également le programme.

Chapitre 11

LES FONCTIONS DES BIBLIOTHÈQUES STANDARD

Plusieurs domaines de la programmation ne sont pas couverts par le langage, mais par ses bibliothèques, qui sont standardisées par la norme : les manipulations de chaînes, de fichiers, l'allocation dynamique. Ce chapitre présente quelques unes des fonctions les plus courantes de ces bibliothèques. Pour pouvoir les utiliser, il faut obligatoirement déclarer leur prototype dans le module qui les appelle : pour celà, il suffit d'inclure le fichier de déclarations correspondant qui se trouve dans le répertoire */usr/include* dans le cas Unix.

11.1 Les manipulations de chaînes

Rappelons qu'une chaîne est un tableau de caractères contenant une valeur de chaîne composée de caractères quelconques et terminée par le caractère de code nul. Le fichier des déclarations de prototype s'inclut par : `#include <string.h>`.

char `*strcat (char *dest, char *source);` Délivre `dest`, pointeur sur la chaîne qui est la concaténation des chaînes `dest` (d'abord) et `source` (ensuite).

char `*strchr (char *ch, char c);` Délivre un pointeur sur la première apparition de `c` dans la chaîne `ch`, ou `NULL`.

int `strcmp (char *ch1, char *ch2);` Compare les chaînes `ch1` et `ch2`, et délivre :

$$\begin{array}{lcl} ch1 < ch2 & \rightarrow & -1 \\ ch1 = ch2 & \rightarrow & 0 \\ ch1 > ch2 & \rightarrow & +1 \end{array}$$

char `*strcpy (char *dest, char *source);` Copie `source` dans `dest` et délivre `dest`.

int `strlen (char *ch);` Délivre la longueur effective de `ch`, caractère `NULL` exclu.

char `*strstr (char *ch, char *ssch);` Délivre le pointeur sur la première sous-chaîne `ssch` contenue dans `ch`, ou `NULL`.

char `*strncat (char *dest, char *source, unsigned n);` Comme `strcat`, mais limité aux `n` premiers caractères de `dest`.

char `*strncpy (char *dest, char *source, unsigned n);` Comme `strcpy`, mais ne copie que les `n` premiers (s'il y en a `n`) caractères de `source`.

int *strncmp* **(char** **ch1,* **char** **ch2,* **unsigned** *n);* Comme *strcmp*, mais ne compare au plus que *n* caractères.

char **strlwr* **(char** **ch);* Délivre *ch* convertie en minuscules.

char **strupr* **(char** **ch);* Délivre *ch* convertie en majuscules ;

11.2 Les manipulations de zones de mémoire

Ces fonctions travaillent sur des zones de type quelconque (pointeurs de type **void** *) et s'utilisent conjointement avec des forceurs. C'est une généralisation des fonctions sur les chaînes. Elles sont déclarées également dans le fichier *string.h* (en théorie tout du moins, car elles sont déclarées dans *memory.h* sur le HP). Sur les SUN, on les obtient à partir de la bibliothèque *System V* en faisant l'édition de liens avec l'option *-L /usr/5lib* qui ira chercher le fichier */usr/5lib/libc.a*.

void **memchr* **(void** **z,* **unsigned char** *c,* **unsigned** *n);* Recherche le premier *c* parmi les *n* premiers caractères de la zone *z*, et délivre un pointeur sur *c*.

int *memcmp* **(void** **z1,* **void** **z2,* **unsigned** *n);* Compare les *n* premiers caractères des zones *z1* et *z2*, et délivre 0 si les zones sont égales, une valeur positive si $z1 > z2$, une valeur négative si $z1 < z2$

void **memcpy* **(void** **z1,* **void** **z2,* **unsigned** *n);* Copie *n* caractères de *z2* dans *z1* et délivre *z1*.

void **memmove* **(void** **z1,* **void** **z2,* **unsigned** *n);* Comme *memcpy*, mais fonctionne même si les zones se chevauchent.

void **memset* **(void** **z,* **unsigned char** *c,* **unsigned** *n);* Délivre *z* et positionne ses *n* premiers caractères à la valeur *c*.

Exemple : Pour copier deux tableaux *a* et *b* d'éléments d'un type *T* quelconque, on fera :

```
memcpy ((void *)a, (void *)b, sizeof (a));
```

11.3 Les fonctions de test ou de transformation de caractère

Ces fonctions sont souvent implémentées par des macros du pré-compilateur. Elles rendent un entier à considérer comme un booléen, sauf pour les deux fonctions de transformation.

int *isalpnum* **(int** *c);* *c* est il un caractère aphanumérique ?

int *isalpha* **(int** *c);* *c* est il un caractère alphabétique ?

int *iscntrl* **(int** *c);* *c* est il un caractère de contrôle ?

int *isdigit* **(int** *c);* *c* est il un caractère chiffre ?

int *islover* **(int** *c);* *c* est il un caractère lettre minuscule ?

int *isupper* **(int** *c);* *c* est il un caractère lettre majuscule ?

int *isspace* **(int** *c);* *c* est il un caractère blanc, TAB, RET, ... ?

int *isxdigit* **(int** *c);* *c* est il un caractère hexadécimal ?

int *tolower* **(int** *c);* Délivre *c* si ce n'est pas une lettre, sinon la lettre *c* en minuscule.

int *toupper* **(int** *c);* Délivre *c* si ce n'est pas une lettre, sinon la lettre *c* en majuscule.

11.3.1 Les conversions

Les conversions de chaînes en nombres et vice versa sont parfois décrites dans le fichier *stdlib.h* (pas sur SUN ni HP).

double *atof* **(char** **ch***)** ; conversion chaîne *ch* en flottant double

int *atoi* **(char** **ch***)** ; conversion chaîne *ch* en entier

long *atol* **(char** **ch***)** ; conversion chaîne *ch* en entier long

char **itoa* **(int** *e*, **char** **ch*, **int** *base***)** ; convertit *e* dans la chaîne *ch* suivant la *base*, et délivre un pointeur sur *ch*

char **ltoa* **(long** *el*, **char** **ch*, **int** *base***)** ; idem, pour convertir l'entier long *el*.

int *sprintf* **(char** **tampon*, **char** **format*, ...**)** ; Analogue à *printf*, mais range le résultat dans le *tampon* et non à l'écran.

int *sscanf* **(char** **tampon*, **char** **format*, ...**)** ; Analogue à *scanf*, mais extrait les données du *tampon* et non du clavier.

11.4 Les entrées sorties et les fichiers

Pour la bibliothèque C, un fichier est toujours un flot de caractères inorganisé, et muni d'un accès séquentiel et d'un accès direct (au niveau du caractère). Les fichiers s'utilisent avec l'insertion du fichier *stdio.h*. On y trouve les définitions des constantes *NULL* et *EOF* déjà rencontrées, ainsi que la définition d'un type *FILE* (une structure). La plupart des opérateurs sur les fichiers ont un paramètre de type pointeur sur *FILE*. Aussi, pour "déclarer" un fichier *f*, on déclare en fait :

```
FILE *f ;
```

Trois fichiers sont prédéfinis : *stdin*, *stdout*, *stderr* (entrée, sortie et erreur standard).

11.4.1 Ouverture et Fermeture

L'opération d'ouverture réalise à la fois l'assignation avec un fichier externe, et l'ouverture proprement dite.

*FILE *fopen* **(char** **nom*, **char** **mode***)** ; Ouvre le fichier externe dont le *nom* est le premier paramètre (une chaîne), avec le *mode* donné en second paramètre, et délivre un pointeur de fichier, ou *NULL* si le fichier externe n'est pas trouvé. Le mode est aussi une chaîne :

r	ouverture en lecture
w	ouverture en écriture (efface le fichier s'il existe déjà)
a	ouverture en allongement (écriture en fin de fichier)
r+	ouverture en lecture pour mise à jour
w+	analogue à w, mais permet de lire.
a+	ouverture en lecture et en allongement.

A ces six modes, on peut ajouter un *"b"* final si le système a la notion de fichier binaire. Pour MS/DOS, celà empêche qu'un \n soit copié sous la forme des deux caractères NL+RET.

Exemple :

```
{ FILE *f, *fopen( char *,  char *) ;
  if ((f = fopen("../alpha"", "r")) == NULL)
    printf("Pas l'accès sur ../alpha \n") ;
```

```
  else
     ...
}
```

*FILE *freopen (***char** **nouveau,* **char** **mode, FILE *ancien) ;* Ferme le fichier *ancien* et ouvre le fichier externe *nouveau* en l'assignant à *ancien*, et délivre *ancien* si tout va bien, ou *NULL* sinon. Cette fonction sert surtout à réassigner dynamiquement les fichiers standard :

Exemple :

```
if (freopen("../alpha", "r", stdin) == NULL)
   printf("Pas d'accès sur ../alpha\n");
else   /* stdin redirigé sur ../alpha */
   ...
```

int *fclose (FILE *f)* Ferme *f* et délivre 0 si tout va bien, *EOF* dans le cas contraire.

11.4.2 Fin de fichiers et erreurs

int *feof (FILE *f)* Si la fin de fichier est atteinte pour *f*, délivre vrai (valeur $\neq 0$) et faux sinon.

int *ferror (FILE *f)* Délivre vrai si une erreur est détectée sur *f*, et faux sinon.

void *clearerr (FILE *f)* Efface les indicateurs de fin de fichier et d'erreur du fichier *f*

11.4.3 Accès séquentiel par caractère

int *fgetc (FILE *f) ;*

int *getc (FILE *f) ;* lit un caractère sur *f*, et délivre *EOF* si la fin de fichier est atteinte ou si une erreur arrive ; l'instruction *c = getc (stdin) ;* est équivalent à *c = getchar () ;*

int *fputc (***char** *c, FILE *f) ;*

int *putc (***char** *c, FILE *f) ;* Ajoute *c* à *f*, délivre le caractère écrit ou *EOF* en cas d'erreur. *putc (c, stdout) ;* est équivalent à *putchar (c) ;*.

int *ungetc (***int***c, FILE *f) ;* Renvoit le dernier caractère lu (le paramètre *c*) sur le fichier *f* : ce caractère sera celui que lira le *getc* suivant. On ne peut pas faire deux *unget* successifs sans intercaler une lecture. Cette fonction délivre *c* si elle s'est bien déroulée, et *EOF* sinon. On ne peut pas renvoyer *EOF* (ce n'est pas un caractère).

11.4.4 Accès séquentiel par ligne complète

char **fgets (***char** **ch,* **int** *n, FILE *f) ;* lit les caractères sur *f*, jusqu'à ce que ou bien *n - 1* caractères soient lus, ou bien une fin de ligne soit rencontrée. Les caractères lus sont rangés dans *ch*. Si la fin de ligne a été lue, elle est stockée dans *ch* ou *NULL* en cas d'erreur ou de fin de fichier. Dans le même ordre d'idée, on se rappellera la fonction *gets* sur l'entrée standard (paragraphe 4.2).

int *fputs (***char** **ch, FILE *f) ;* Ecrit sur *f* les caractères de la chaîne *ch*, le caractère nul non compris, et sans ajouter de fin de ligne. Délivre la dernier caractère écrit ou *EOF* en cas d'erreur. Voir également *puts* pour la sortie standard (paragraphe 4.2).

11.4.5 Accès séquentiel avec format

int *fprintf (FILE *f,* **char** **format, ...) ;* Analogue au *printf* pour la sortie standard (paragraphe 4.3). Délivre le nombre de caractères écrits ou une valeur négative en cas d'erreur.

int *fscanf (FILE *f,* **char** **format, ...) ;* Analogue au *scanf* pour l'entrée standard. Délivre le nombre de valeurs assignées aux variables paramètres, ou *EOF* si la fin de fichier est atteinte avant de pouvoir lire la première valeur.

11.4.6 Accès direct

int *fseek (FILE *f,* **long** *dep,* **int** *mode) ;* Déplace la fenêtre du fichier *f* de *dep* caractères, en commençant au début du fichier si *mode = SEEK_SET* (constante qui vaut 0), à la position courante si *mode = SEEK_CUR* (1), ou en partant de la fin du fichier si *mode = SEEK_END* (2). On peut lire ensuite par les fonctions d'accès séquentiel. La fonction délivre *EOF* en cas d'erreur, et 0 sinon.

long *ftell (FILE * f) ;* Délivre la valeur du déplacement de la fenêtre de *f*, ou *-1L* en cas d'erreur.

11.4.7 Manipulation des fichiers

int *remove (FILE *f) ;* Détruit le fichier externe associé à *f*. Délivre faux en cas d'erreur.

int *rename (FILE *ancien, FILE *nouveau) ;* Renomme ancien par nouveau. Délivre faux en cas d'erreur.

11.4.8 Allocation dynamique

Les fonctions d'allocation dynamique permettent l'allocation, la surallocation et la libération de blocs de mémoire repérés par un pointeur.

void **malloc (***unsigned** *taille) ;* Alloue un bloc de *taille* caractères et délivre une valeur de pointeur sur le bloc, ou *NULL* en cas d'impossibilité.

void **realloc (***void** **p,* **unsigned** *taille) ;* Permet une surallocation pour le bloc repéré par *p* et précédemment crée par *malloc*.

void **free (***void** **p) ;* libère le bloc repéré par *p* et créé par *malloc* ou modifié par *realloc*.

A titre d'exemple, on donne ici une structure rudimentaire de liste et une fonction d'ajout en tête.

Exemple :

```
struct elem      /* un entier + le lien vers le suivant */
  { int n ;
    struct elem *suivant ;
  } ;
struct elem *tete = NULL ;    /* tête de liste */
void ajout_en_tete( int x)     /* ajoute x en tête de liste */
  { struct elem *p ;           /* pointeur de création */
    p = ( struct elem *) malloc ( sizeof ( struct elem)) ;
        /* forceur obligatoire, car malloc est de type void * */
    p->n = x ; p->suivant = tete ; tete = p ;
  }
```

11.4.9 Fonctions Mathématiques

On utilise les fonctions mathématiques en incluant le fichier *math.h*. Les arguments sont de type double (valeurs en radians pour les angles). On trouve : *acos, cos, asin, sin, atan, tan, exp, log, log10, sqrt, fabs* pour les flottants et *abs* pour les entiers,...

11.4.10 Relations avec UNIX

On devrait trouver dans *stdlib.h* plusieurs fonctions dont les suivantes qui permettent de manipuler certains objets UNIX :

void *exit* (**int** *n*) ; délivre le code retour *n* pour le shell et termine l'exécution de la fonction en cours.

char **getenv* (**char** **ch*) ; Délivre un pointeur sur la valeur d'une variable de l'environnement dont le nom est dans *ch*. Ainsi, pour connaitre le répertoire personnel, on fera :

```
repertoire = getenv ("HOME") ;
```

int *system* (**char** **commande*) ; Donne la *commande* à UNIX pour qu'il l'exécute.

11.5 Exercice

On reprend encore le module de gestion de piles abstraites. On veut cette fois que le modèle de piles ne fixe pas la taille de la pile. Rajoutez donc le paramètre *taille* au constructeur *init*, et allouez dynamiquement la pile dans ce constructeur.

Chapitre 12

LE PRÉ-COMPILATEUR

Comme son nom l'indique, le pré-compilateur analyse les sources qu'on soumet au compilateur avant leur compilation, et traite les directives qui lui sont adressées. Une directive commence par le caractère dièse (#) en première colonne. Il a d'autres actions qui aident le compilateur : il joint les lignes terminées par une contre-barre, il ôte les commentaires, il concatène les chaînes adjacentes, et il tient à jour quelques variables qui lui sont propres.

12.1 Directive define

On a déjà présenté cette directive au paragraphe 2.5 pour définir des constantes. Rappelons sa syntaxe :

RÈGLE
déclaration de macro pour le pré-compilateur :
define identificateur [(identificateur [, identificateur]*)] texte

Sans paramètre et sans texte, ***define*** sert simplement à définir un identificateur (voir paragraphe 12.2). Avec les parenthèses et un texte, la directive sert à déclarer une macro-instruction paramétrée par les identificateurs entre parenthèses. Comme le texte peut commencer par une parenthèse, il ne peut y avoir d'espace entre le nom de la macro et la parenthèse ouvrante.

Exemple :

```
# define and &&
# define P1600
# define VOIR(x) printf("-->%d\n", x)
```

La première définition donne un substitut à l'opérateur &&, la deuxième définit la "variable" ***P1600*** sans lui donner de valeur, et la troisième définit la macro ***VOIR(x)*** qui affiche l'entier passé en paramètre.

La nouvelle norme a introduit deux opérateurs pour cette directive.

12.1.1 Opérateur

Un paramètre formel précédé du dièse dans une définition de macro indique que le paramètre effectif doit être entouré de guillemets. Si le paramètre effectif contient lui-même un guillemet, ce dernier sera correctement traité (\" à la place de ").

Exemple :

```
# define VOIR(x) printf(#x " = %d\n", x)
```

L'appel :

```
VOIR(indice)
```

sera transformé en :

```
printf("indice" " = %d\n", indice)
```

puis en :

```
printf("indice = %d\n", indice)
```

12.1.2 Opérateur

Moins utile, cet opérateur placé devant ou derrière un paramètre formel indique que la valeur du paramètre effectif doit être collé à l'entité lexicale qui précède ou qui suit la paramètre, de façon à ne produire qu'une entité lexicale au lieu de deux.

Exemple :

```
# define OUVRIRF(f,n) f = open("temp##n", "w") ;
```

L'appel :

```
OUVRIRF(fd,3)
```

sera transformé en :

```
fd = open("temp3", "w") ;
```

12.2 Directives de compilation conditionnelle

Les directives **#if, #ifdef, #ifndef, #else, #elif** et **#endif** permettent de faire de la compilation conditionnelle sur des expressions de constantes (**#if**) ou sur le fait qu'un identificateur du pré-compilateur est défini ou non défini (**#ifdef, #ifndef**). L'expression **#ifdef** *PC* est équivalente à l'expression **#if defined** *PC*.

Exemple :

```
# if UC == PC
#   define LG 16
# elif UC == CYBER
#   define LG 48
# else
#   define LG 32
# endif
```

Ici, l'action compilée est en fait encore une directive (définition du nombre de bits dans un mot d'un PC, du Control Data du CICB, et des minis), mais ç'aurait pu être une séquence d'instructions C.

12.3 Directive d'inclusion

On a également déjà utilisé la directive **#include** : elle sert à inclure le fichier nommé en argument. Le nom du fichier est entouré de chevrons comme dans :

```
#include <stdio.h>
```

lorsque le fichier appartient à la bibliothèque C et qu'il se trouve dans le répertoire */usr/include* sous Unix ou *TURBOC**INCLUDE* sous MS-DOS avec Turbo-C, ou il est entouré de guillemets comme dans :

```
#include "mestypes.h"
```

quand le fichier est un fichier local de l'utilisateur. Le suffixe *.h* n'est pas obligatoire.

Un fichier qu'on inclut peut lui-même contenir des directives d'inclusion.

12.4 Identificateurs prédéfinis

Le pré-compilateur prédéfinit les identificateurs suivants dans les C nouvelle norme :

`__LINE__`	numéro courant de la ligne source
`__FILE__`	nom courant du fichier compilé
`__DATE__`	date de la compilation
`__TIME__`	heure de la compilation
`__STDC__`	vaut 1 si le compilateur est à la norme

Annexe A

SCHÉMAS DE TRADUCTION LD EN C

A.1 Constante

Elles sont définies au niveau du pré-compilateur ; elles seront remplacées textuellement avant la compilation.

```
const c = 3;                    # define c 3
```

A.2 Type

Deux types de base, **int** pour les entiers, les booléens et les caractères, **float** pour les réels, avec des options : un entier peut être court (**short**), caractère (**char**), long (**long**) ou sans signe (**unsigned**)...

On peut définir de nouveaux types par une déclaration de type :

```
type t = <définition          typedef <définition du
         du type>;                      type> t;
```

A.3 Variable

```
var v : untype;              untype v;
var v : constructeur de       <dépend du constructeur de
          type;                  type>
  Exemple :
var v :  tableau [1..5]        t v[4];
             de t;
```

A.4 Instructions

Les instructions ont ou n'ont pas de point-virgule les terminant : voir la syntaxe précise dans chaque cas. Dans la suite, une instruction ti est laissée sans point-virgule (ce n'est pas un séparateur d'instructions). Par exemple, si ti est une affectation, ti sera à faire suivre du point-virgule. Si ti est une suite d'instructions, alors on emploie une instruction composée, consistant à encadrer ti par des accolades {}.

A.4.1 Sélections

```
cas                              if (c1) t1
  c1 → t1                        else if ...
  ...                            else if (cn) tn
  cn → tn                        else tn+1
  aut → tn+1
fcas
```

Cas particulier où le cas porte toujours sur le même facteur :

```
 cas                             switch (exp)
   exp = v1 → t1                   { case v1 : t1 ; break ;
   ...                                        ...
   exp = vn → tn                     case vn : tn ; break ;
   aut → tn+1                        default : tn+1
  fcas                             }
```

A.4.2 Itérations

Boucle générale LD

```
jqa ev1, ev2, ev3 faire      { enum {boucle, ev1, ev2, ev3}
                                   etat = boucle ;
                               do /* jusqua etat != boucle */
  t1 ;                         { t1
   qd cond1 sortir par ev1 ;     if (cond1) {etat = ev1 ; break ;}
  t2 ;                           t2
   qd cond2 sortir par ev2 ;     if (cond1) {etat = ev2 ; break ;}
  ...                            ...
  tn ;                           tn
 refaire                          } while (etat == boucle) ;
                                switch (etat)
   sortie ev1 : s1              { case ev1 : s1 ; break ;
   sortie ev2 : s2                case ev2 : s2 ; break ;
   sortie ev3 : s3                case ev3 : s3 ; break ;
                                }
 fiter                        }
```

Boucle avec sortie en tête :

```
  jqa ev1 faire                  while ( ! cond1)
    qd cond1 sortir par ev1 ;      t1
   t1
  refaire
```

Boucle pour

```
pour i depuis d pas p jqa f faire
  t                                  for ( i=d ; i<=f ; i+=p ;)
refaire                                t
```

A.5 Sous-programmes

Seules les fonctions existent ; elles sont toutes déclarées au même niveau que le programme principal (fonction *main*). Un seul mode de transmission de paramètre : par valeur. Pour transmettre en résultat un objet qui n'est pas un tableau, on transmet son adresse.

```
procedure p fixe (i :t1) mod (var x :t2; var t :untableau);
debut
  ...
fin;
```

```
void p(t1 i, t2 *x, untableau t)
 {
      ...
 }
```

Annexe B

CORRIGÉS DES EXERCICES

B.1 Chapitre 5

```
#include <stdio.h>
#define VRAI 1
#define FAUX 0

 void main()
{  int a, b, r;     /*  2 opérandes et le résultat */
   int bon ;        /*  booléen  le calcul est-il faisable ? */
   char operateur; /* l'opérateur */
   while (1)
    { scanf("%d %c %d", &a, &operateur, &b);
      bon = VRAI;
      switch(operateur)
       {  case '+' : r = a + b;  break;
          case '-' : r = a - b;  break;
          case '*' : r = a * b;  break;
          case '%' : r = a % b;  break;
          case '/' :
            if (b == 0) bon = FAUX;
            else r = a / b;
            break;
          default : bon = FAUX; /* erreur de frappe */
       }
      if (bon) printf("%d\n", r);
      else printf("****\n");
    }
}
```

B.2 Chapitre 6

```
#include <stdio.h>
#define TAILLE 10
```

```
 void main()
{  int tab[TAILLE];  /* tableau nécessaire pour le tri */
   int i, j;         /* indices dans tab */
   int n;            /* dernier entier lu */
   int nb;           /* nombre d'entiers dans tab */
   enum {on_lit, trop, fini} etat;  /* boucle de lecture */
  nb = 0; i = 0;   /* tab est vide au départ */
  etat = on_lit;
   while (etat == on_lit)
    {  if ((scanf("%d", &n) == EOF)) etat = fini;
       else  /* il faut ajouter n dans tab */
         if (nb == TAILLE) etat = trop;
         else /* c'est bon, il faut insérer n à sa place */
          { i = nb;
             while (n < tab[i - 1] && i > 0)
              { tab[i] = tab[i - 1];
                i--;
              }
            tab[i] = n; nb = nb++;
          }
     }
   if (etat == trop) printf("trop d'entiers à trier!\n");
   else
    {  for (i = 0; i < nb; i++)
        printf("%d  ", tab[i]);
      printf("\n");
    }
}
```

B.3 Chapitre 7

1.

```
#include <stdio.h>
 void main()
  {  int x = 4;
     int *p;
     int **pp;
    pp = &p;
    p = &x;
    **pp = 0; /* mise à 0 de x par double indirection */
    printf("%d\n", i);
  }
```

2.

```
#include <stdio.h>
#define TAILLE 10
```

```
int tab[TAILLE] = {1, 0, 3, 4, 0, 6, 0, 0, 9, 10};

void main()
  {  int *deb = &tab[0];           /* repère le 1er entier */
     int *fin = &tab[TAILLE - 1]; /* dernier */
     int *p;                      /* parcours */
     for ( p = deb; p <= fin; p++ )
       if (*p == 0)
         printf("%d  ", p - deb); /* valeur relative */
    printf("\n");
  }
```

B.4 Chapitre 8

1.

```
  /* Module de gestion de l'heure */

#include <stdio.h>

int heures, minutes, secondes;      /* l'heure, globale au module */

void Afficher_heure( void)
  { printf("Il est %d heure%c %d minute%c %d seconde%c\n",
           heures, (heures > 1)? 's' : ' ',
           minutes, (minutes > 1)? 's' : ' ',
           secondes, (secondes > 1)? 's' : ' ');
  }

void Etablir_heure( int h,  int m,  int s)
  { heures = h; minutes = m; secondes = s;
  }

void tic( void)
  { secondes += 1;
     if (secondes >= 60)
       { secondes = 0;
         minutes += 1;
          if (minutes >= 60)
            { minutes = 0;
              heures += 1;
               if (heures > 24) heures = 0;
            }
       }
  }

void main( void)
  { Etablir_heure(1, 59, 23); Afficher_heure(); tic(); Afficher_heure();
    Etablir_heure(23, 59, 59); Afficher_heure(); tic(); Afficher_heure();
  }
```

2.

```
#include <stdio.h>

 char texte[] = "On veut crypter un texte selon le principe du\n\
décalage des lettres.";

  /* Crypter un caractère */
 void Crypter_car( int cle,  char *c)
  {  enum {majuscule, minuscule} sorte; /* sorte de lettre */
     if (*c >= 'a' && *c <= 'z') sorte = minuscule;
     else  if (*c >= 'A' && *c <= 'Z') sorte = majuscule;
     else  return;
    *c = *c + cle % 26;  /* codage : rétablir une lettre si on déborde */
     if (sorte == minuscule && *c > 'z' || sorte == majuscule && *c > 'Z')
       *c = *c - 26;
  }

  /* Crypter un texte */
 void Crypter_texte( int cle,  char *t)
  {  while (*t)
       Crypter_car(cle, t++);
  }

 void main( void)
  {  int i;
    printf("Texte avant cryptage :\n  ");
    i = 0;
     while (texte[i]) printf("%c", texte[i++]);
    Crypter_texte(2, texte);
    printf("\nTexte après cryptage :\n  ");
    i = 0;
     while (texte[i]) printf("%c", texte[i++]);
    printf("\n");
  }
```

B.5 Chapitre 9

Le module de gestion de la pile est le suivant :

```
  /* module de gestion d'une pile de "chaînes" */

 typedef  char chaine[20];          /* type des éléments de la pile */
 static  int s = 0;                /* sommet de la pile */
 static chaine pile[50];           /* "la" pile */

 void empiler(chaine c)
  { strcpy(pile[s++], c);
  }

 void depiler(chaine c)
```

```
  { strcpy(c, pile[s--]);
  }

chaine* sommet( void)
  {  return (chaine*)pile[s];
  }

 int pilevide( void)
  {  return (s < 0);
  }
```

Son interface, définissant ce que son client doit connaître (et donc ce qui est accessible), est déclaré dans un fichier d'en-tête, appelé ici `pile.h` :

```
  /* Interface du module "pile de chaînes" */

 typedef  char chaine[20];
 extern  void depiler();
 extern  void empiler();
 extern chaine* sommet();
 extern  int pilevide();
```

Le programme client inclut le fichier d'en-tête :

```
  /* Lecture d'un texte, et affichage à l'envers */
#include <stdio.h>
#include "pile.h"

 void main( void )
  { chaine x;
     while ((scanf("%s",x)) != EOF)
      empiler(x);
     do { depiler(x);
          printf("%s ", x);
       }  while ( !pilevide());
    printf("\n");
  }
```

Le programme de lecture et affichage ne fait pas appel à l'accesseur `sommet`. Pour afficher la chaîne en sommet de pile, il écrit ;

```
printf("%s", sommet());
```

B.6 Chapitre 10

Le module de gestion du type abstrait pile est le suivant :

```
  /* module de gestion du type abstrait pile de "chaînes" */

 typedef  char chaine[20];          /* type des éléments de la pile */
```

```
typedef struct                    /* type abstrait PILE, composé de */
  { int s;                       /* son sommet */
    chaine pile[50];             /* et de "la" pile */
  } PILE;

void empiler(PILE *p, chaine c)
  { strcpy(p->pile[p->s++], c);
  }

void depiler(PILE *p, chaine c)
  { strcpy(c, p->pile[p->s--]);
  }

chaine* sommet(PILE *p)
  { return (chaine*)p->pile[p->s];
  }

int pilevide(PILE *p)
  { return (p->s < 0);
  }

void init(PILE *p)
  { p->s = 0;
  }
```

Tous les accès à la structure se font via le repère `p` qui pointe sur un article. L'interface connu des utilisateurs est dans le fichiers `pileab.h` :

```
  /* Interface du module "pile de chaînes" */

typedef char chaine[20];
typedef struct
  { int s;                  /* sommet de la pile */
    chaine pile[50];            /* "la" pile */
  } PILE;
extern void depiler();
extern void empiler();
extern chaine* sommet();
extern int pilevide();
extern void init();
```

et le programme utilisateur devient le suivant :

```
  /* Lecture d'un texte, et affichage à l'envers */
  /*    via le type abstrait PILE */
#include <stdio.h>
#include "pileab.h"

void main( void )
  { chaine x;
    PILE ma_pile;    /* la pile nécessaire au traitement */
```

```
    init(&ma_pile);
     while ((scanf("%s",x)) != EOF)
      empiler(&ma_pile, x);
     do { depiler(&ma_pile, x);
          printf("%s ", x);
        } while ( !pilevide(&ma_pile));
    printf("\n");
  }
```

La seule chose à ne pas oublier est de transmettre la pile `ma_pile` par référence au module de pile.

B.7 Chapitre 11

Le module de gestion du type abstrait pile subit beaucoup plus de modifications que ses clients :

```
 /* module de gestion du type abstrait pile de "chaînes" */
 /*    la taille de la pile est dynamique */

typedef char chaine[20];          /* type des éléments de la pile */
typedef struct                    /* type abstrait PILE, composé de */
 { int s;                        /* son sommet */
   chaine *pile;                 /* et de "la" pile, sans taille */
 } PILE;

void empiler(PILE *p, chaine c)
 { strcpy(*(p->pile), c);
   p->pile++; p->s++;
 }

void depiler(PILE *p, chaine c)
 { p->s--; p->pile--;
   strcpy(c, *(p->pile));
 }

chaine *sommet(PILE *p)
 { chaine *ss = p->pile;
   ss--;
    return (chaine*) *ss;
 }

int pilevide(PILE *p)
 { return (p->s < 0);
 }

void init(PILE *p, int taille)   /* initialise à vide une pile de taille "taille" */
 { p->s = 0;
   p->pile = (chaine*) malloc(taille * sizeof(chaine));
 }
```

Seules les spécifications de la procédure d'initialisation ont changé dans le fichier d'en-tête (qu'on ne recopie pas ici ; il s'appelle `pileabv.h`). Cette spécification nouvelle entraîne une seule modification chez le client : il faut donner la taille de la pile lors de son initialisation.

```
  /* Lecture d'un texte, et affichage à l'envers */
  /*    via le type abstrait PILE de taille dynamique */
#include <stdio.h>
#include "pileabv.h"

 void main( void )
  { chaine x;
    PILE ma_pile;                    /* la pile nécessaire au traitement */
    init(&ma_pile, 15);              /* allocation et initialisation de ma_pile */
     while ((scanf("%s", &x)) != EOF)
      empiler(&ma_pile, x);
     do { depiler(&ma_pile, x);
          printf("%s ", x);
        } while (!pilevide(&ma_pile));
    printf("\n");
  }
```

Index