

Université Mohammed V – Agdal
Faculté des Sciences
Département de Mathématiques et Informatique

A. EL GHAZI & S. EL HAJJI

Groupe Analyse Numérique et Optimisation

<http://perso.menara.ma/~elghazi/c.pdf>

<http://www.fsr.ac.ma/mia/data/c.pdf>

**DESA MIA
2005/2006**

Programmer en Langage C

Généralité sur le langage C

INTRODUCTION

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur un exemple commenté. Vous y découvrirez comment s'expriment les instructions de base. Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme.

Pour commencer voici un exemple de programme en langage C, accompagné d'un exemple d'exécution.

```
#include <stdio.h>
main()
{
printf("Premier programme\n ");
}
```

L'exécution donne :

Premier programme

Analyse du premier programme :

La directive # include

La ligne :

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

Est une directive du préprocesseur : Pour compiler correctement un fichier, le compilateur a besoin d'informations concernant les déclarations de structures de données et de variables externes ainsi que de l'aspect (on dira prototype) des fonctions prédéfinies. Toutes ces informations sont contenues dans des fichiers avec l'extension .h. Ces fichiers doivent être inclus dans le fichier que l'on veut compiler.

En générale le langage C offre la directive du préprocesseur

```
#include < nom de fichier >
```

Par exemple, pour utiliser la fonction printf, il faut inclure le fichier stdio.h (standard input output) .

La fonction main

La ligne :

```
main()
```

Est un "en-tête", elle précise que ce qui sera décrit à sa suite est le programme principal.

Un programme en C apparaît comme une fonction qui porte le nom `main()`, le programme doit être délimité par des accolades « `{` » pour le début, et « `}` » pour la fin.

On dit que les instructions situées entre ces accolades forment un "bloc".

La fonction printf

La ligne :

```
printf("Premier programme \n");
```

Est une instruction qui appelle une fonction "prédéfinie" nommée `printf.`, cette fonction reçoit un argument qui est :

"Premier programme \n"

Les guillemets servent à délimiter une "chaîne de caractères". La notation `\n` est conventionnelle: elle représente un caractère de fin de ligne, c'est-à-dire un caractère qui, lorsqu'il est envoyé à l'écran, provoque le passage à la ligne suivante.

Nous verrons que, de manière générale, le langage C prévoit une notation de ce type (`\` suivi d'un caractère) pour un certain nombre de caractères dits "de contrôle".

Quelques règles d'écriture

Ce paragraphe vous expose un certain nombre de règles d'écriture d'un programme en langage C. Nous y parlerons précisément de ce que l'on appelle les "identificateurs" et les "mots clés", du format libre dans lequel on écrit les instructions, de l'usage des séparateurs et des commentaires.

Les identificateurs

Les identificateurs servent à désigner les différents "objets" manipulés par le programme: variables, fonctions, etc. Ils sont formés d'une suite de caractères (lettres ou les chiffres), le premier d'entre eux étant nécessairement une lettre.

Noter que :

- le symbole `'_'` est considéré comme une lettre.
- C distingue les majuscules et les minuscules, ainsi: `'Nom__var'` est différent de `'nom_var'`
- La longueur des identificateurs n'est pas limitée, mais C distingue seulement les 31 premiers caractères.

Exemple :

Identificateurs corrects:

```
var1  
nbr_2  
_nom_3  
Nom_var  
deuxieme_var  
mot_francais
```

Identificateurs incorrects:

```
1var  
nbr-2  
-nom-3  
Nom var  
deuxième_var  
mot_français
```

Les mots clés

Langage C

Certains "mots clés" sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs :

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

Les séparateurs

Dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier doivent impérativement être séparés soit par un espace, soit par une fin de ligne. Par contre, dès que la syntaxe impose un séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme). Ainsi, vous devrez impérativement écrire :

```
int x,y ;  
et non :
```

```
int x,y  
En revanche, vous pourrez écrire indifféremment :
```

```
int n,compte,total ;  
Ou plus lisiblement :  
int n, compte, total ;
```

Les commentaires

Un commentaire c'est un texte explicatif destiné aux lecteurs du programme et qui n'a aucune incidence sur sa compilation. Il est formé de caractères quelconques placés entre les symboles /* et */.
Voici quelques exemples de commentaires:

```
// Mon premier Programme  
Ou :  
/* commentaire s'étendant  
sur plusieurs lignes  
de programme source */.
```

Les types et les variables

INTRODUCTION

Nous allons, tout au long de ce chapitre, étudier l'ensemble des types que fournit le langage C ainsi que l'utilisation des variables.

La notion de type

La mémoire centrale est un ensemble de "positions binaires" nommées bits. Les bits sont regroupés en octets (8bits), et chaque octet est repéré par son adresse.

L'ordinateur, ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être codée sous cette forme. Dans ces conditions, il ne suffit pas de connaître le contenu d'un emplacement de la mémoire pour être en mesure de lui attribuer une signification.

Par exemple, si un octet contient la valeur binaire suivante :1000 1101 alors ça peut représenter un nombre entier positif ou négatif, comme elle peut représenter un nombre réel ou un caractère, ou même une instruction de programme, ou un graphique

Donc il n'est pas possible d'attribuer une signification à une information binaire tant que l'on ne connaît pas la manière dont elle a été codée et son type.

C manipule deux types de base: les entiers et les nombres flottants.

Les entiers

En C, on dispose de divers types d'entiers qui se distinguent par la place qu'ils occupent en mémoire :

- o sur 1 octet, les entiers signés et non signés (char) et (unsigned char).
- o sur 2 octets, les entiers signés et non signés (short) et (unsigned short).
- o sur 4 octets, les entiers signés et non signés (long) et (unsigned long).
- o le type int (unsigned int) est selon les machines synonymes de short (unsigned short) ou de long (unsigned long)

Le type char

Le type char désigne un entier signé codé sur 1 octet. Il en découle que toutes les opérations autorisées sur les entiers peuvent être utilisées sur les caractères. Aussi surprenant que cela puisse paraître, on peut ajouter ou soustraire deux caractères, ajouter ou soustraire un entier à un caractère.

Une utilisation classique de cette souplesse d'utilisation est la conversion d'un caractère c désignant un chiffre en sa valeur v correspondante:

$v = c - '0'$

Les types short, long ou int

Le type short représente un entier signé codé sur 2 octets (de -32768 à 32767) et le type unsigned short représente un entier non signé codé sur 2 octets (de 0 à 65535). Le type long (ou int pour nos machines)

représente un entier signé codé sur 4 octets (de -2147843648 à 2147843647) et le type unsigned long (ou unsigned int pour nos machines) représente un entier non signé codé sur 4 octets (de 0 à 4294967295).

Le type réel

Les nombres à virgule flottante (abusivement appelés réels) servent à coder de manière approchée les nombres réels. Un nombre à virgule flottante est composé d'un signe, d'une mantisse et d'un exposant. On dispose de trois types de nombres à virgule flottante, les types float, double et long double.

Les floats

Un float est codé sur 4 octets avec 1 bit de signe, 23 bits de mantisse et 8 bits d'exposant (valeurs comprises entre $3.4 * 10^{-38}$ et $3.4 * 10^{38}$).

Les doubles

Un double est codé sur 8 octets avec 1 bit de signe, 52 bits de mantisse et 11 bits d'exposant (valeurs comprises entre $1.7 * 10^{-308}$ et $1.7 * 10^{308}$).

Les long doubles

Un long double est codé sur 10 octets avec 1 bit de signe, 64 bits de mantisse et 15 bits d'exposant (valeurs comprises entre $3.4 * 10^{-4932}$ et $3.4 * 10^{4932}$).

LES CONSTANTES

Nous avons présenté les divers types de données élémentaires du langage C sans dire comment écrire une constante de l'un de ces types dans un programme C. Nous allons à présent donner la syntaxe utilisée dans le langage C pour désigner des constantes littérales.

Les constantes entières

Les constantes entières peuvent s'exprimer

- o en notation décimale: 123, -123, etc...
- o en notation octale avec un 0 en première position: 0123
- o en notation hexadécimale avec les caractères 0x ou 0X en première position : 0x1b 0X2c, 0X1B, 0X2C, etc...

Le type d'une constante entière est le << plus petit >> type dans lequel il peut être représenté :

- o notation décimale : int, sinon long, sinon unsigned long
- o notation octale ou décimale : int, sinon unsigned int, sinon unsigned long

Des suffixes permettent de changer cette classification :

- o U, u : constante de type unsigned
- o L, l : constante de type long

Exemple.

1L, 0x7FFU, 16UL, etc...

Les constantes flottantes

Une constante flottante se présente sous la forme d'une suite de chiffres (partie entière), un point qui joue le rôle de virgule, une suite de chiffres (partie fractionnaire), une des deux lettres e ou E, éventuellement le signe + ou - suivi d'une suite de chiffres (valeur absolue de l'exposant)

La partie entière ou la partie fractionnaire peut être omise (pas les deux); de même le point ou l'exposant peut être omis (pas les deux).

Une constante flottante est supposée être de type double. Le suffixe F indique qu'elle est de type float. Le suffixe LF indique qu'elle est de type long double.

Exemple.

.5e7, 5.e6, 5e6, 5000

Les constantes de type caractère

Les constantes de type caractère se note entre apostrophes: 'a' '2' ""

Le caractère ' se note \" et le caractère \ se note \"\.

On peut également représenter des caractères non imprimables à l'aide de séquences d'échappement.

Voici une liste non exhaustive de caractères non imprimable:

char	valeur
\n	nouvelle ligne
\t	tabulation horizontale
\v	tabulation verticale
\b	retour d'un caractère en arrière
\r	retour chariot
\f	saut de page
\a	beep
\'	apostrophe
\"	guillemet
\\	anti-slash
\ddd	code ASCII en notation octale
\xdd	code ASCII en notation hexadécimale

LES VARIABLES

Une variable est un emplacement en mémoire identifié par un identificateur, contenant une valeur d'un type donné et dont la valeur peut être modifiée durant l'exécution du programme.

Avant d'utiliser une variable on doit la déclarer c à d spécifier son type et lui associer un identificateur.

Les déclarations suivantes :

```
char Terme ; int nombre ; double nombr_2;
```

sont des déclarations de variables ainsi Terme est défini comme une variable de type char, nombre comme une variable de type int et nombr_2 comme une variable de type double.

Noter que toute déclaration de variable se termine par un point-virgule et qu'une déclaration peut se trouver n'importe où sur une ligne. Plusieurs variables de même type peuvent bien sûr être déclarées en une seule fois. Le type est alors suivi d'une liste d'identificateurs séparés par des virgules. Ainsi :

```
short a, b, c;
```

Déclare trois variables a, b, c de type short.

Langage C

Les variables peuvent être initialisées à la déclaration avec des constantes. On utilise pour cela le symbole « = ». Les exemples suivants sont des initialisations de variables correctes :
`int max = 0; float som = 0.5 ;`

Les opérateurs et les expressions

INTRODUCTION

Le langage C est certainement l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (arithmétiques, relationnels, logiques) ou moins classiques (manipulations de bits)

LES OPÉRATEURS

Les opérateurs arithmétiques :

- - : changement de signe
- * : Multiplication
- / : Division
- % : Modulo (reste de la division de deux entiers)
- + : Addition
- - : Soustraction

Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -. En cas de priorités identiques, les calculs s'effectuent de "gauche à droite". On dit que l'on a affaire à une "associativité de gauche à droite".

Remarque : Une valeur de type caractère peut être considérée de deux façons:

- comme le caractère concerné: a, Z, fin de ligne....
- comme le code ASCII de ce caractère par exemple le caractère A est représenté par 69 ainsi :c ='A' ; b=c+1 ; affecte 70 à b.

Les opérateurs relationnels :

Ces opérateurs binaires (vrai ou faux) permettent d'établir des conditions logiques en comparant leurs deux opérandes.

- == test si égal
- != test si différent
- < test si inférieur
- <= test si inférieur ou égal
- > test si supérieur
- >= test si supérieur ou égal

Les opérateurs logiques:

Ces opérateurs permettent les opérations booléennes classiques sur des conditions logiques.

- ! Négation logique d'une condition
- && ET logique de conditions
- || OU logique de conditions

Opérateur d'incrément

Incrément préfixée/postfixée.

- a++ est équivalent à a=a+1;
- b=a++; est équivalent à b=a; puis a=a+1;

Langage C

- `b=++a;` est équivalent à `a=a+1;` puis `b=a;`

On dit que `++` est :

-un opérateur de pré incrémentation lorsqu'il est placé à gauche de la "lvalue" sur laquelle il porte,

-un opérateur de post incrémentation lorsqu'il est placé à droite de la "lvalue" sur laquelle il porte

Les opérateurs d'affectation élargie

C dispose d'opérateurs encore plus puissants. Ainsi, vous pourrez remplacer:

- `i = i + k` par `i += k`

- `a = a*b` par `a *= b.`

D'une manière générale, C permet de condenser les affectations de la forme :

`lvalue = lvalue opérateur expression` en : `lvalue opérateur= expression`

Cette possibilité concerne tous les opérateurs binaires arithmétiques. Voici la liste complète de tous ces nouveaux opérateurs nommés "opérateurs d'affectation élargie": `+=` `-=` `*=` `/=` `%=` `|=` ...

Opérateur conditionnel

`a?b:c` Vaut la valeur de `b` si `a` est vrai, `c` sinon.

Exemples :

- `a = 2+3`
o valeur de `a`: 5
- `r = 3%2`
o valeur de `a`: 1
- `a = (3==3)`
o valeur de `a`: 1
- `a = (6==5)`
o valeur de `a`: 0
- `a = (2!=3)`
o valeur de `a`: 1
- `a = (6<=3)`
o valeur de `a`: 0
- `a = !1`
o valeur de `a`: 0
- `a = ((3==3) || (6<=3))`
o valeur de `a`: 1
- `a = ((3==3) && (6<=3))`
o valeur de `a`: 0
- `i=1; a=i++;`
o valeur de `a`: 1 et de `i`:2
- `i=1; a=++i;`
o valeur de `a`: 2 et de `i`:2

LES CONVERSIONS D'AJUSTEMENT DE TYPE

Dans les expressions mixtes (dans lesquelles interviennent des opérandes de types différents), le compilateur procède à une conversion de type telle que `int -> float`. Une telle conversion se fait suivant une "hiérarchie" qui permet de ne pas dénaturer la valeur initiale⁵, à savoir :

`int -> long -> float -> double -> long double`

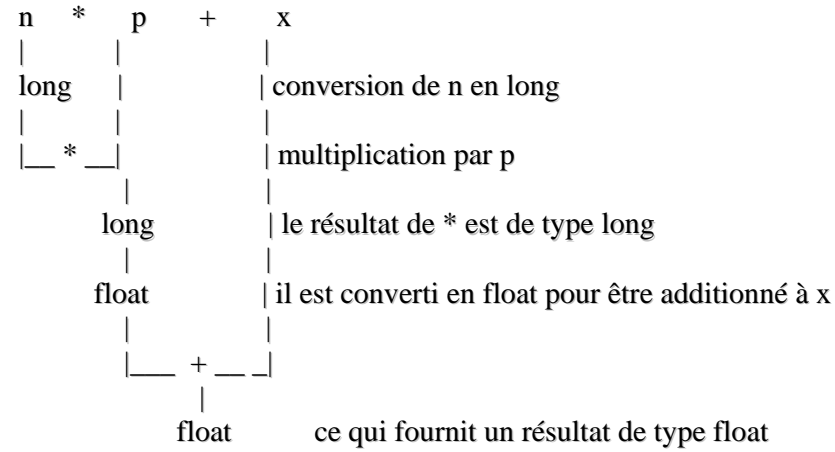
Langage C

On peut bien sûr convertir directement un int en double ; en revanche, on ne pourra pas convertir un double en float ou en int.

Exemple : si n est de type int, p de type long et x de type float, l'expression :

$n * p + x$

sera évaluée suivant ce schéma :



LES ENTRÉES-SORTIES CONVERSATIONELLES

INTRODUCTION

Jusqu'ici, nous avons utilisé de façon intuitive les fonctions printf pour afficher des informations à l'écran. Nous vous proposons maintenant d'étudier en détail les différentes possibilités de ces fonctions, ce qui nous permettra de répondre à des questions telles que :

- quelles sont les écritures "autorisées" pour des nombres fournis en données ? Que se passe-t-il lorsque l'utilisateur ne les respecte pas ?
- comment organiser les données lorsque l'on mélange les types numériques et les types caractères ?
- que se produit-il lorsque, en réponse à scanf, on fournit trop ou trop peu d'informations ?
- comment agir sur la présentation des informations à l'écran ?

LES POSSIBILITÉS DE LA FONCTION PRINTF

Nous avons déjà vu que le premier argument de printf est une chaîne de caractères qui spécifie à la fois des caractères à afficher tels quels, des "codes de format" repérés par %. Un "code de conversion" (tel que c, d ou f) y précise le type de l'information à afficher.

D'une manière générale, il existe d'autres caractères de conversion soit pour d'autres types de valeurs, soit pour agir sur la précision de l'information que l'on affiche. De plus, un code de format peut contenir des informations complémentaires agissant sur le "cadrage", le "gabarit" ou la "précision". Ici, nous nous limiterons aux possibilités les plus usitées de printf. Sachez toutefois que le paragraphe 1.2 de l'annexe A vous en fournit un panorama complet.

La syntaxe de printf

D'une manière générale, nous pouvons dire que l'appel à printf se présente ainsi :

```
printf ( format, liste_d'expressions )
```

format :

- constante chaîne (entre " "),
- pointeur sur une "chaîne de caractères" (cette notion sera étudiée ultérieurement).

liste_d'expressions : suite d'expressions séparées par des virgules d'un type en accord avec le code format correspondant.

Les principaux codes de conversion

- c char : caractère affiché (convient aussi à short ou à int compte tenu des conversions systématiques)
- d int (convient aussi à char ou à int, compte tenu des conversions systématiques)
- u unsigned int (convient aussi à unsigned char ou à unsigned short, compte tenu des conversions systématiques)
- ld long
- lu unsigned long

f double ou float (compte tenu des conversions systématiques float -> double) écrit en notation "décimale" avec six chiffres après le point (par exemple : 1.234500 ou 123.456789)

Action sur le gabarit d'affichage

Par défaut, les entiers sont affichés avec le nombre de caractères nécessaires (sans espaces avant ou après). Les flottants sont affichés avec six chiffres après le point (aussi bien pour le code e que f).

Un nombre placé après % dans le code de format précise un gabarit d'affichage, c'est-à-dire un nombre minimal de caractères à utiliser. Si le nombre peut s'écrire avec moins de caractères, printf le fera précéder d'un nombre suffisant d'espaces ; en revanche, si le nombre ne peut s'afficher convenablement dans le gabarit imparti, printf utilisera le nombre de caractères nécessaires.

Voici quelques exemples, dans lesquels nous fournissons, à la suite d'une instruction printf, à la fois des valeurs possibles des expressions à afficher et le résultat obtenu à l'écran. Notez que le symbole ^ représente un espace.

```
printf ("%3d", n) ; /* entier avec 3 caractères minimum */
```

- n = 20

```
^20
```

- n = 3 ^

```
^3
```

- n = 2358

```
2358
```

- n = -5200

```
-5200
```

```
printf ("%f", x) ; /* notation décimale gabarit par défaut (6 chiffres après point) */
```

- x = 1.2345

```
1.234500
```

- x = 12.3456789

```
12.345679
```

```
printf ("%10f", x) ; /* notation décimale gabarit mini 10 (toujours 6 chiffres après point) */
```

- x = 1.2345

```
^^1.234500
```

- x = 12.345

```
^12.345000
```

- x = 1.2345E5

```
123450.000000
```

Actions sur la précision

Pour les types flottants, on peut spécifier un nombre de chiffres (éventuellement inférieur à 6) après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle). Ce nombre doit apparaître, précédé d'un point, avant le code de format (et éventuellement après le gabarit). Voici quelques exemples :

```
printf ("%10.3f", x) ;
```

- x = 1.2345

```
^^^^1.235
```

- x = 1.2345E3

```
^^1234.500
```

- `x = 1.2345E7`
`12345000.000`

En cas d'erreur de programmation

Deux types d'erreur de programmation peuvent apparaître dans l'emploi de `printf`.

1. Code de format en désaccord avec le type de l'expression

Lorsque le code de format, bien qu'erroné, correspond à une information de même taille (c'est-à-dire occupant la même place en mémoire) que celle relative au type de l'expression, les conséquences de l'erreur se limitent à une "mauvaise interprétation de l'expression". C'est ce qui se passe, par exemple, lorsque l'on écrit une valeur de type `int` en `%u` ou une valeur de type `unsigned int` en `%d`.

En revanche, lorsque le code format correspond à une information de taille différente de celle relative au type de l'expression, les conséquences sont généralement plus désastreuses, du moins si d'autres valeurs doivent être affichées à la suite. En effet, tout se passe alors comme si, dans la "suite d'octets" (correspondant aux différentes valeurs à afficher) reçue par `printf`, le "repérage" des emplacements des valeurs suivantes se trouvait soumis à un "décalage".

2. Nombre de codes de format différent u nombre d'expressions de la liste

Dans ce cas, il faut savoir que C cherche toujours à satisfaire le contenu du format.

Ce qui signifie que, si des expressions de la liste n'ont pas de code format, elles ne seront pas affichées.

C'est le cas dans cette instruction où la valeur de `p` ne sera pas affichée : `printf ("%d", n, p) ;`

En revanche, si vous prévoyez trop de codes de format, les conséquences seront là encore assez désastreuses puisque `printf` cherchera à afficher... n'importe quoi. C'est le cas dans cette instruction où deux valeurs seront affichées, la seconde étant (relativement) aléatoire : `printf ("%d %d ", n) ;`

LA MACRO PUTCHAR

L'expression :

`putchar (c)`

joue le même rôle que :

`printf ("%c", c) ;`

LES POSSIBILITÉS DE LA FONCTION SCANF

La syntaxe de `scanf`

D'une manière générale, l'appel de `scanf` se présente ainsi :

`scanf (format, liste_d_adresses)`

format :

-constante chaîne (entre " "),

-pointeur sur une "chaîne de caractères" (cette notion sera étudiée ultérieurement)

liste_d_adresses : liste de "lvalue", séparées par des virgules, d'un type en accord avec le code de format correspondant.

Les principaux codes de conversion de `scanf`

Pour chaque code de conversion, nous précisons le type de la "lvalue" correspondante.

Langage C

c char
d int
u unsigned int
hd short int
hu unsigned short
ld long int
lu unsigned long
f ou e float écrit indifféremment dans l'une des deux notations : décimale ou exponentielle (avec la lettre e ou E)
lf ou le double avec la même présentation que ci-dessus
s chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

Remarque :

Contrairement à ce qui se passait pour printf, il ne peut plus y avoir ici de conversion automatique puisque l'argument transmis à scanf est l'adresse d'un emplacement mémoire. C'est ce qui justifie l'existence d'un code hd par exemple pour le type short ou encore celle des codes lf et le pour le type double.

Action sur le gabarit

Comme dans les codes de format de printf, on peut, dans un code de format de scanf, préciser un gabarit. Dans ce cas, le traitement d'un code de format s'interrompt soit à la rencontre d'un séparateur, soit lorsque le nombre de caractères indiqués a été atteint.

LA MACRO GETCHAR

L'expression : `c = getchar()` joue le même rôle que : `scanf ("%c", &c)` tout en étant plus rapide puisque ne faisant pas appel au mécanisme d'analyse d'un format. Notez bien que `getchar` utilise le même tampon (image d'une ligne) que `scanf`. En toute rigueur, `getchar` est une "macro" (comme `putchar`) dont les instructions figurent dans `stdio.h`. Là encore, l'omission d'une instruction `#include` appropriée conduit à une erreur à l'édition de liens.

Les instructions de contrôle

INTRODUCTION

A priori, dans un programme, les instructions sont exécutées séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent. Or la puissance et le "comportement intelligent" d'un programme proviennent essentiellement :

- de la possibilité d'effectuer des "choix", de se comporter différemment suivant les "circonstances"
- de la possibilité d'effectuer des "boucles", autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

L'INSTRUCTION IF

Blocs d'instructions

Un bloc est une suite d'instructions placées entre { et }. Les instructions figurant dans un bloc sont absolument quelconques. Il peut s'agir aussi bien d'instructions simples (terminées par un point-virgule) que d'instructions structurées (choix, boucles) lesquelles peuvent alors à leur tour renfermer d'autres blocs...

Un bloc peut se réduire à une seule instruction, voire être "vide". Voici deux exemples de blocs corrects :

```
{ }  
{ i = 1 ; }
```

Syntaxe de l'instruction if

```
if (expression) if (expression) instruction_1 instruction_1 else instruction_2
```

Le mot else et l'instruction qu'il introduit sont facultatifs, de sorte que cette instruction if présente deux formes.

L'INSTRUCTION SWITCH

Syntaxe de l'instruction switch

```
switch (expression) {  
case constante_1 : [ suite_d'instructions_1 ]  
case constante_2 : [ suite_d'instructions_2 ]  
.....  
case constante_n : [ suite_d'instructions_n ]  
[ default : suite_d'instructions ]  
}
```

- expression : expression entière quelconque,

- constante : expression constante d'un type entier quelconque (char est accepté car il sera converti en int),
- suite_d'instructions : séquence d'instructions quelconques.
- les crochets ([et]) signifient que ce qu'ils renferment est facultatif.
- l'étiquette "default" Il est possible d'utiliser le mot clé "default" comme étiquette à laquelle le programme se "branchera" dans le cas où aucune valeur satisfaisante n'aura été rencontrée auparavant.

L'INSTRUCTION DO... WHILE

Abordons maintenant la première façon de réaliser une boucle en C, à savoir l'instruction do... while. Exemple d'introduction de l'instruction do... while

```
main()
{ int n ;
do
{ printf ("donnez un nb >0 : ") ;
scanf ("%d", &n) ;
}
while (n<=0)
printf ("vous avez fourni %d\n", n) ;
```

L'instruction :

do { } while (n<=0) ;
répète l'instruction qu'elle "contient" (ici un bloc) tant que la condition mentionnée (n<=0) est vraie (c'est-à-dire, en C, non nulle). Autrement dit, ici, elle demande un nombre à l'utilisateur tant qu'il ne fournit pas une valeur positive.

Syntaxe de l'instruction do... while

```
do instruction while (expression) ;
```

L'INSTRUCTION WHILE

Syntaxe de l'instruction while

```
while (expression) instruction
```

Exemple :

```
while ( printf ("donnez un nombre : ") , scanf ("%d", &n), som<=100)
som += n ;
```

L'INSTRUCTION FOR

Étudions maintenant la dernière instruction permettant de réaliser des boucles, à savoir l'instruction for.

Exemple d'introduction de l'instruction for

Considérez ce programme :

```
main()
{ int i ;
for ( i=1 ; i<=5 ; i++ )
{ printf ("bonjour " ) ;
printf ("%d fois\n", i) ;
}
}
```

La ligne :

```
for ( i=1 ; i<=5 ; i++ )
```

comporte en fait trois expressions. La première est évaluée (une seule fois) avant d'entrer dans la boucle. La deuxième conditionne la poursuite de la boucle. Elle est évaluée avant chaque parcours. La troisième, enfin, est évaluée à la fin de chaque parcours.

Le programme précédent est équivalent au suivant :

```
main()
{ int i ;
i = 1 ;
while (i<=5)
{ printf ("bonjour " ) ;
printf ("%d fois\n", i) ; i++ ;
}
}
```

Syntaxe de l'instruction for

```
for ( [ expression_1 ] ; [ expression_2 ] ; [ expression_3 ] ) instruction
```

les crochets ([et]) signifient que leur contenu est facultatif.

D'une manière générale, nous pouvons dire que :

```
for ( expression_1 ; expression_2 ; expression_3 ) instruction
```

est équivalent à :

```
expression_1 ; while (expression_2) { instruction expression_3 ;
```

Tableaux et Pointeurs

INTRODUCTION

Comme tous les langages, C permet d'utiliser des "tableaux". On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique; chaque élément est repéré par un "indice" précisant sa position au sein de l'ensemble.

Par ailleurs le langage C dispose de "pointeurs", c'est-à-dire de variables destinées à contenir des adresses d'autres "objets" (variables, fonctions. . .).

A priori, ces deux notions de tableaux et de pointeurs peuvent paraître fort éloignées l'une de l'autre. Toutefois, il se trouve qu'en C un lien indirect existe entre ces deux notions, à savoir qu'un identificateur de tableau est une "constante pointeur". Cela peut se répercuter dans le traitement des tableaux, notamment lorsque ceux-ci sont transmis en argument de l'appel d'une fonction.

C'est ce qui justifie que ces deux notions soient regroupées dans un seul chapitre.

LES TABLEAUX A UN INDICE

Exemple d'utilisation d'un tableau en C

Supposons que nous souhaitions déterminer, à partir de vingt notes d'élèves (fournies en données), combien d'entre elles sont supérieures à la moyenne de la classe.

S'il ne s'agissait que de calculer simplement la moyenne de ces notes, il nous suffirait d'en calculer la somme, en les cumulant dans une variable, au fur et à mesure de leur lecture. Mais, ici, il nous faut à nouveau pouvoir consulter les notes pour déterminer combien d'entre elles sont supérieures à la moyenne ainsi obtenue. Il est donc nécessaire de pouvoir "mémoriser" ces vingt notes.

Pour ce faire, il paraît peu raisonnable de prévoir vingt variables scalaires différentes (méthode qui, de toute manière, serait difficilement transposable à un nombre important de notes).

Le tableau va nous offrir une solution convenable à ce problème, comme le montre le programme suivant.

```
#include <stdio.h>
int main(void)
{ int i, som, nbm ;
  double moy ;
  int t[20] ;
  for (i=0 ; i<20 ; i++)
  { printf ("donnez la note numéro %d : ", i+1) ;
    scanf ("%d", &t[i]) ;
  }
  for(i=0, som=0 ; i<20 ; i++) som += t[i] ;
  moy = som / 20 ;
  printf ("\n\n moyenne de la classe : %f\n", moy) ;
  for (i=0, nbm=0 ; i<20 ; i++ )
  if (t[i] > moy) nbm++ ;
  printf ("%d élèves ont plus de cette moyenne", nbm) ;
  return 0 ;
}
```

La déclaration : `int t[20]` réserve l'emplacement pour 20 éléments de type `int`. Chaque élément est repéré par sa "position" dans le tableau, nommée "indice". Conventionnellement, en langage C, la première position porte le numéro 0. Ici, donc, nos indices vont de 0 à 19. Le premier élément du tableau sera désigné par `t[0]`, le troisième par `t[2]` le dernier par `t[19]`.

Plus généralement, une notation telle que $t[i]$ désigne un élément dont la position dans le tableau est fournie par la valeur de i . Elle joue le même rôle qu'une variable scalaire de type `int`.

QUELQUES REGLES

Les éléments de tableau

Un élément de tableau est une lvalue. Il peut donc apparaître à gauche d'un opérateur d'affectation comme dans: $t[2] = 5$

Il peut aussi apparaître comme opérande d'un opérateur d'incrément, comme dans:

$t[3]++$ $--t[i]$

En revanche, il n'est pas possible, si $t1$ et $t2$ sont des tableaux d'entiers, d'écrire $t1 = t2$;

Les indices

Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère, compte tenu des règles de conversion systématique).

La dimension d'un tableau

La dimension d'un tableau (son nombre d'éléments) ne peut être qu'une constante ou une expression constante.

Ainsi, cette construction :

```
#define N 50
```

....

```
int t[N] ; double h[2*N- 1 ];
```

est correcte. Elle ne le serait pas, par contre, si N était une constante symbolique définie par `const int N = 50` (les expressions N et $2*N-1$ n'étant alors plus calculables par le compilateur).

Débordement d'indice

Aucun contrôle de "débordement d'indice" n'est mis en place par la plupart des compilateurs. Pour en comprendre les conséquences, il faut savoir que, lorsque le compilateur rencontre une lvalue telle que $t[i]$, il en détermine l'adresse en ajoutant à l'adresse de début du tableau t , un "décalage" proportionnel à la valeur de i (et aussi proportionnel à la taille de chaque élément du tableau). De sorte qu'il est très facile (si l'on peut dire !) de désigner et, partant, de modifier, un emplacement situé avant ou après le tableau.

LES TABLEAUX À PLUSIEURS INDICES

Leur déclaration

Comme tous les langages, C autorise les tableaux à plusieurs indices (on dit aussi à plusieurs dimensions). Par exemple, la déclaration:

`int t[5][3]` réserve un tableau de 15 (5×3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations:

```
t[3][2] t[i][j] t[i-3][i +j]
```

Notez bien que, là encore, la notation désignant un élément d'un tel tableau est une lvalue.

POINTEURS

INTRODUCTION

Nous avons déjà été amené à utiliser l'opérateur & pour désigner l'adresse d'une lvalue. D'une manière générale, le langage C permet de manipuler des adresses par l'intermédiaire de variables nommées "pointeurs".

En guise d'introduction à cette nouvelle notion, considérons les instructions :

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

La première réserve une variable nommée ad comme étant un "pointeur" sur des entiers. Nous verrons que * est un opérateur qui désigne le contenu de l'adresse qui le suit. Ainsi, à titre "mnémonique", on peut dire que cette déclaration signifie que *ad, c'est-à-dire l'objet d'adresse ad, est de type int ; ce qui signifie bien que ad est l'adresse d'un entier.

L'instruction : ad = &n ; affecte à la variable ad la valeur de l'expression &n. L'opérateur & (que nous avons déjà utilisé avec scanf) est un opérateur unaire qui fournit comme résultat l'adresse de son opérande. Ainsi, cette instruction place dans la variable ad l'adresse de la variable n.

Après son exécution.

L'instruction suivante : *ad = 30 ; signifie : affecter à la lvalue *ad la valeur 30. Or *ad représente l'entier ayant pour adresse ad. Après exécution de cette instruction, la valeur de n devient 30. Bien entendu, ici, nous aurions obtenu le même résultat avec : n = 30 ;

QUELQUES EXEMPLES

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

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

Les variables ad1, ad2 et ad sont donc des pointeurs sur des entiers. Considérons maintenant ces instructions :

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

Les deux premières placent dans ad1 et ad2 les adresses de n et p. La troisième affecte à *ad1 la valeur de l'expression : * ad2 + 2.

Autrement dit, elle place à l'adresse désignée par ad1 la valeur (entière) d'adresse ad2, augmentée de 2.

Cette instruction joue donc ici le même rôle que : n = p + 2 ;

De manière comparable, l'expression : * ad1 += 3

jouerait le même rôle que : n = n + 3

et l'expression : (* ad1) ++

jouerait le même rôle que n++

Remarque :

1) Si ad est un pointeur, les expressions ad et *ad sont des lvalue ; autrement dit ad et *ad sont modifiables. En revanche, il n'en va pas de même de &ad. En effet, cette expression désigne, non plus une variable pointeur comme ad, mais l'adresse de la variable ad telle qu'elle a été définie par le compilateur. Cette adresse est nécessairement fixe et il ne saurait être question de la modifier (la même remarque

s'appliquerait à &n, où n serait une variable scalaire quelconque). D'une manière générale, des expressions telles que (&ad)++ ou (&p)++ seront rejetées à la compilation.

2) Une déclaration telle que : int * ad réserve un emplacement pour un pointeur sur un entier. Elle ne réserve pas en plus un emplacement pour un tel entier.

TABLEAUX ET POINTEUR

En langage C, l'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau.

TABLEAUX A UN INDICE

Supposons, par exemple, que l'on effectue la déclaration suivante :

```
int t[10]
```

La notation t est alors totalement équivalente à &t[0]. L'identificateur t est considéré comme étant de type pointeur sur le type correspondant aux éléments du tableau, c'est-à-dire, ici, int*.

Ainsi, voici quelques exemples de notations équivalentes :

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

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

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

Dans la seconde façon, nous avons dû recopier la "valeur" représentée par t dans un pointeur nommé p. En effet, il ne faut pas perdre de vue que le symbole t représente une adresse constante (t est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que t++ aurait été invalide, au même titre que, par exemple, 3++. Un nom de tableau est un pointeur constant ; ce n'est pas une lvalue. Remarque importante :

Nous venons de voir que la notation t[i] est équivalente à *(t+i) lorsque t est déclaré comme un tableau.

En fait, cela reste vrai, quelle que soit la manière dont t a été déclaré. Ainsi, avec :

int * t ; les deux notations précédentes resteraient équivalentes. Autrement dit, on peut utiliser t[i] dans un programme où t est simplement déclaré comme un pointeur.

TABLEAUX A PLUSIEURS INDICES

Lorsque le compilateur rencontre une déclaration telle que :

```
int t[3] [4] ;
```

il considère en fait que t désigne un tableau de 3 éléments, chacun de ces éléments étant lui-même un tableau de 4 entiers.

Langage C

Autrement dit, si t représente bien l'adresse de début de notre tableau t , il n'est plus de type `int *` (comme c'était le cas pour un tableau à un indice) mais d'un type "pointeur sur des blocs de 4 entiers", type qui devrait se noter théoriquement :

```
int [4] *
```

Dans ces conditions, une expression telle que $t+1$ correspond à l'adresse de t , augmentée de 4 entiers (et non plus d'un seul !).

Exemple

Le tableau M à deux dimensions est défini comme suit:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe sur le tableau $M[0]$ qui a la valeur: `{0,1,2,3,4,5,6,7,8,9}`. L'expression $(M+1)$ est l'adresse du deuxième élément du tableau et pointe sur $M[1]$ qui a la valeur: `{10,11,12,13,14,15,16,17,18,19}`.

Explication

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le vecteur `{0,1,2,3,4,5,6,7,8,9}`, le deuxième élément est `{10,11,12,13,14,15,16,17,18,19}` et ainsi de suite.

L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que:

$M+I$ désigne l'adresse du tableau $M[I]$

Problème

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à.d.: aux éléments $M[0][0]$, $M[0][1]$, ..., $M[3][9]$?

Une solution consiste à convertir la valeur de M (qui est un pointeur sur un tableau du type `int`) en un pointeur sur `int`. On pourrait se contenter de procéder ainsi:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
int *P;
P = M; /* conversion automatique */
```

Cette dernière affectation entraîne une conversion automatique de l'adresse $\&M[0]$ dans l'adresse $\&M[0][0]$. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent: Généralement, on gagne en lisibilité en explicitant la conversion mise en oeuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

Solution

Voici finalement la version que nous utiliserons:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
               {10,11,12,13,14,15,16,17,18,19},
               {20,21,22,23,24,25,26,27,28,29},
               {30,31,32,33,34,35,36,37,38,39}};
```

```
int *P;
```

```
P = (int *)M; /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Exemple

Les instructions suivantes calculent la somme de tous les éléments du tableau M:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
               {10,11,12,13,14,15,16,17,18,19},
               {20,21,22,23,24,25,26,27,28,29},
               {30,31,32,33,34,35,36,37,38,39}};
```

```
int *P;
```

```
int I, SOM;
```

```
P = (int*)M;
```

```
SOM = 0;
```

```
for (I=0; I<40; I++)
```

```
    SOM += *(P+I);
```

Attention !

Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel il faut calculer avec le nombre de colonnes indiqué dans la déclaration du tableau.

Exemple

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes:

```
int A[3][4];
```

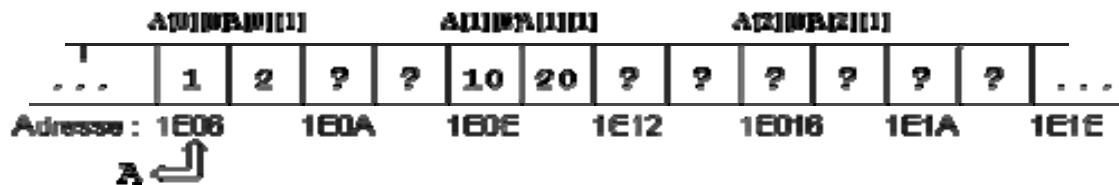
```
A[0][0]=1;
```

```
A[0][1]=2;
```

```
A[1][0]=10;
```

```
A[1][1]=20;
```

Dans la mémoire, ces composantes sont stockées comme suit :



L'adresse de l'élément A[I][J] se calcule alors par:

$A + I*4 + J$

Conclusion

Pour pouvoir travailler à l'aide de pointeurs dans un tableau à deux dimensions, nous avons besoin de quatre données:

- a) l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau
- b) la longueur d'une ligne réservée en mémoire
(- voir déclaration - ici: 4 colonnes)
- c) le nombre d'éléments effectivement utilisés dans une ligne
(- p.ex: lu au clavier - ici: 2 colonnes)
- d) le nombre de lignes effectivement utilisées
(- p.ex: lu au clavier - ici: 2 lignes)

LA COMPARAISON DE POINTEURS

Il ne faut pas oublier qu'en C un pointeur est défini à la fois par une adresse en mémoire et par un type. On ne pourra donc comparer que des pointeurs de même type. Par exemple, voici, en parallèle, deux suites d'instructions réalisant la même action :

mise à 1 des 10 éléments du tableau t :

```
int t[10] ;
```

```
int * p ;
```

```
for (p=t ; p<t+10 ; p++)
```

```
*p = 1 ;
```

```
int t[10] ;
```

```
int i ;
```

```
for (i=0 ; i<10 ; i++)
```

```
t[i] = 1 ;
```

LES FONCTIONS

INTRODUCTION

Le langage C permet de découper un programme en plusieurs parties nommées souvent "modules". Cette programmation dite "modulaire" se justifie pour de multiples raisons:

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le "programme principal" les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée à son tour en modules plus élémentaires; ce processus de décomposition pouvant être répété autant de fois que nécessaire, comme le préconisent les méthodes de "programmation structurée".
- La programmation modulaire permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus que la notion d'argument" permet de "paramétrer" certains modules.
- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. Cet aspect sera d'autant plus marqué que C autorise effectivement la compilation séparée de tels modules. En C, il n'existe qu'une seule sorte de module, nommé fonction.

LA FONCTION:

Une fonction peut recevoir des arguments et fournir un résultat scalaire qu'on utilisera dans une expression, comme, par exemple, dans :

```
y = sqrt(x)+3;
```

- La valeur d'une fonction pourra très bien ne pas être utilisée; c'est ce qui se passe fréquemment lorsque vous utilisez printf ou scanf. Bien entendu, cela n'a d'intérêt que parce que de telles fonctions réalisent une action (ce qui, dans d'autres langages, serait réservé aux sous-programmes ou procédures).

Une fonction pourra ne fournir aucune valeur.

- Une fonction pourra fournir un résultat non scalaire (nous n'en parlerons toutefois que dans le chapitre consacré aux structures.

- Une fonction pourra modifier les valeurs de certains de ses arguments (il vous faudra toutefois attendre d' avoir étudié les pointeurs pour voir par quel mécanisme elle y parviendra).

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

EXEMPLE DE DÉFINITION ET D'UTILISATION D'UNE FONCTION EN C

Exemple :

```
/***** le programme principal (fonction main) *****/  
int main(void)  
{  
double fexple (double m, int n, int p) ; /* déclaration de la fonction fexple */  
double x = 1.5 ;  
double y, z ;
```

Langage C

```
int n = 3, p = 5, q = 10 ;
/* appel de fexple avec les arguments x, n et p */
y = fexple (x, n, p) ;
printf ("valeur de y : %f\n", y) ;
Le langage C 18
/* appel de fexple avec les arguments x+0.5, q et n-1 */
z = fexple (x+0.5, q, n-1) ;
printf ("valeur de z : %f\n", z) ;
return 0 ;
}
/***** la fonction fexple *****/
double fexple (double x, int b, int c)
{
double val ; /* déclaration d'une variable "locale" à fexple*/
val = x * x + b * x + c ;
return val ;
}
```

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

Les noms des arguments n'ont d'importance qu'au sein du corps de la fonction. Ils servent à décrire le travail que devra effectuer la fonction quand on l'appellera en lui fournissant trois valeurs.

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration:

```
double val ;
```

Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type double nommée val. On dit que val est une "variable locale" à la fonction fexple, de même que les variables telles que n, p, y...sont des variables locales à la fonction main.

L'instruction suivante de notre fonction fexple est une affectation classique (faisant toutefois intervenir les valeurs des arguments x, n et p).

Enfin, l'instruction return val précise la valeur que fournira la fonction à la fin de son travail.

En définitive, on peut dire que fexple est une fonction telle que fexple (x, b, c) fournisse la valeur de l'expression

$$x^2 + bx + c.$$

Examinons maintenant la fonction main. Vous constatez qu'on y trouve une déclaration:

```
double fexple (double m, int n , int p) ;
```

Elle sert à prévenir le compilateur que fexple est une fonction et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Quant à l'utilisation de la fonction fexple au sein de la fonction main, elle est classique et comparable à celle d'une fonction prédéfinie telle que sqrt. Ici, nous nous sommes contentés d'appeler la fonction à deux reprises avec des arguments différents.

QUELQUES RÈGLES

Arguments muets et arguments effectifs

Les noms des arguments figurant dans l'en-tête de la fonction se nomment des "arguments muets" (ou encore "arguments formels" ou "paramètres formels"). Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

L'instruction return

Voici quelques règles générales concernant cette instruction.

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

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

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

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

Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un.

Cas des fonctions sans valeur de retour ou sans arguments

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

```
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction return.

Quand une fonction ne reçoit aucun argument, on place le mot clé void à la place de la liste d'arguments. Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type float

```
double tirage (void) ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son entête sera de la forme:

```
void message (void) et sa déclaration sera: void message (void);
```

EN C, LES ARGUMENTS SONT TRANSMIS PAR VALEUR

Exemple d'introduction:

```
#include <stdio.h>
int main(void)
{ void echange (int a, int b) ;
int n=10, p=20 ;
printf ("avant appel: %d %d\n", n, p) ;
echange (n, p) ;
printf ("après appel: %d %d", n, p) ;
return 0 ;
}
void echange (int a, int b)
{ int c ;
printf ("début echange : %d %d\n", a, b) ;
c = a ;
a = b ;
b = c ;
printf ("fin echange : %d %d\n", a, b) ;
```

```
}
```

avant appel : 10 20 début échange : 10 20 fin échange : 20 10 après appel : 10 20

La fonction échange reçoit deux valeurs correspondant à ses deux arguments muets a et b. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs n et p.

En effet, lors de l'appel de échange, il y a eu transmission de la valeur des expressions n et p. On peut dire que ces valeurs ont été copiées "localement" dans la fonction échange dans des emplacements nommés a et b. C'est effectivement sur ces copies qu'a travaillé la fonction échange, de sorte que les valeurs des variables n et p n'ont, quant à elles, pas été modifiées. C'est ce qui explique le résultat constaté.

Ce mode de transmission semble donc interdire a priori qu'une fonction produise une ou plusieurs valeurs "en retour", autres que celle de la fonction elle-même.

Or, il ne faut pas oublier qu'en C tous les "modules" doivent être écrits sous forme de fonction. Autrement dit, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction. La solution est de transmettre en argument la "valeur" de l'adresse" d'une variable. La fonction pourra éventuellement agir sur le "contenu" de cette adresse. C'est précisément ce que nous faisons lorsque nous utilisons la fonction scanf.

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

Les arguments effectifs de l'appel de échange sont, cette fois, les adresses des variables a et b. Voyez comme, dans échange, nous avons indiqué, comme arguments muets, deux variables pointeurs destinées à recevoir ces adresses.

LES VARIABLES GLOBALES

Nous avons vu comment échanger des informations entre différentes fonctions grâce à la transmission d'arguments et à la récupération d'une valeur de retour.

En fait, en C, plusieurs fonctions (dont, bien entendu le programme principal main) peuvent partager des variables communes qu'on qualifie alors de globales.

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration.

LES VARIABLES LOCALES

Les variables locales ne sont connues qu'à l'intérieur de la fonction où elles sont déclarées.

LES TABLEAUX TRANSMIS EN ARGUMENT

Lorsque l'on place le nom d'un tableau en argument effectif de l'appel d'une fonction, on transmet finalement l'adresse du tableau à la fonction, ce qui lui permet d'effectuer toutes les manipulations voulues sur ses éléments, qu'il s'agisse d'utiliser leur valeur ou de la modifier. Voyons quelques exemples pratiques.

Cas des tableaux à un indice

a) Premier exemple: tableau de taille fixe

Voici un exemple de fonction qui met la valeur 1 dans tous les éléments d'un tableau de 10 éléments, l'adresse de ce tableau étant transmise en argument.

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

Voici deux exemples d'appels possibles de cette fonction :

```
int t1[10], t2[10] ;
```

```
.....
fct(t1) ;
```

```
....
fct(t2) ;
```

L'en-tête de f peut être indifféremment écrit de l'une des manières suivantes:

```
void fct (int t[10]) void fct (int * t) void fct (int t[])
```

Second exemple: tableau dont le nombre d'éléments est variable

Comme nous venons de le voir, lorsqu'un tableau à un seul indice apparaît en argument d'une fonction, le compilateur n'a pas besoin d'en connaître la taille exacte. Il est ainsi facile de réaliser une fonction capable de travailler avec un tableau de dimension quelconque, à condition de lui en transmettre la taille en argument. Voici, par exemple, une fonction qui calcule la somme des éléments d'un tableau d'entiers de taille quelconque:

```
int som (int t[],int nb)
{ int s = 0, i ;
  for (i=0 ; i<nb ; i++)
  s += t[i] ;
  return (s) ;
}
```

Voici quelques exemples d'appels de cette fonction:

```
int main(void)
{ int t1[30], t2[15], t3[10] ;
  int s1, s2, s3 ;
  .....
  s1 = som(t1, 30) ;
  s2 = som(t2, 15) + som(t3, 10) ;
  .....
}
```

Cas des tableaux à plusieurs indices

a) Premier exemple: tableau de taille fixe

Voici un exemple d'une fonction qui place la valeur 1 dans chacun des éléments d'un tableau de dimensions 10 et 15

```
void raun (int t[10][15])
```

Langage C

```
{ int i, j ;  
for (i=0 ; i<10 ; i++)  
for (j=0 ; j<15 ; j++) t[i][j] = 1 ;  
}
```

b) Second exemple: tableau de dimensions variables

Supposons que nous cherchions à écrire une fonction qui place la valeur 0 dans chacun des éléments de la diagonale d'un tableau carré de taille quelconque. Une façon de résoudre ce problème consiste à "adresser" les éléments voulus par des pointeurs en effectuant le "calcul d'adresse approprié".

```
void diag (int * p, int n)
```

```
{ int i ;  
for (i=0; i<n; i++)  
{  
* p = 0 ;  
p += n+1;  
}  
}
```

Notre fonction reçoit donc, en premier argument, l'adresse du premier élément du tableau, sous forme d'un pointeur de type int *. Ici, nous avons tenu compte de ce que deux éléments consécutifs de la diagonale sont

séparés par n éléments. Si, donc, un pointeur désigne un élément de la diagonale, pour "pointer" sur le suivant il

suffit d'incrémenter ce pointeur de n+1 unités (l'unité étant ici la taille d'un entier).

Remarques:

1) Un appel de notre fonction diag se présentera ainsi:

```
int t [30] [30] ;  
diag (t, 30)
```

Or l'argument effectif t est, certes, l'adresse de t, mais d'un type pointeur sur des blocs de 10 entiers et non pointeur sur des entiers. En fait, la présence d'un prototype pour diag fera qu'il sera converti en un int *.

2) Cette fonction pourrait également s'écrire en y déclarant un tableau à une seule dimension dont la taille (n*n)

devrait alors être fournie en argument (en plus de n). Le même mécanisme d'incrémentation de n+1 s'appliquerait

alors, non plus à un pointeur, mais à la valeur d'un indice.

LES FONCTIONS RECURSIVES

Le langage C autorise la récursivité des appels de fonctions.

Exemple : long fac(n)

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

LES CHAINES DE CARACTÈRES

REPRÉSENTATION DES CHAINES

La convention adoptée

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

Cas des chaînes constantes

C'est cette convention qu'utilise le compilateur pour représenter les "constantes chaîne" (sous-entendu que vous introduisez dans vos programmes), sous des notations de la forme: " bonjour "

De plus, une telle notation sera traduite par le compilateur en un pointeur (sur des éléments de type char) sur la zone mémoire correspondante.

Voici un programme illustrant ces deux particularités:

```
#include <stdio.h>
int main(void)
{ char * adr ;
  adr = "bonjour" ; bonjour
  while (*adr)
  {printf ("%c",* adr) ;
   adr++ ;
  }
  return 0 ;
}
```

La déclaration :

```
char * adr;
```

réserve simplement l'emplacement pour un pointeur sur un caractère (ou une suite de caractères). En ce qui concerne la constante : "bonjour"

le compilateur a créé en mémoire la suite d'octets correspondants mais, dans l'affectation :

```
adr = "bonjour"
```

la notation "bonjour" a comme valeur, non pas la valeur de la chaîne elle-même, mais son adresse.

Initialisation de tableaux de caractères

Comme nous l'avons dit, vous serez souvent amené, en C, à placer des chaînes dans des tableaux de caractères. Mais, si vous déclarez, par exemple : `char ch[20]` ; vous ne pourrez pas pour autant transférer une chaîne constante dans `ch`, en écrivant une affectation du genre : `ch = "bonjour"`;

En effet, `ch` est une constante pointeur qui correspond à l'adresse que le compilateur a attribuée au tableau `ch`; ce n'est pas une lvalue; il n'est donc pas question de lui attribuer une autre valeur (ici, il s'agirait de l'adresse attribuée par le compilateur à la constante chaîne "bonjour") .

Par contre, C vous autorise à initialiser votre tableau de caractères à l'aide d'une chaîne constante. Ainsi, vous pourrez écrire : `char ch[20] = "bonjour"`

Cela sera parfaitement équivalent à une initialisation de `ch` réalisée par une énumération de caractères (en n'omettant pas le code zéro - noté `\0`) :

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

N'oubliez pas que, dans ce dernier cas, les 12 caractères non initialisés explicitement seront .

- soit initialisés à zéro (pour un tableau de classe statique) : on voit que, dans ce cas, l'omission du caractère `\0` ne serait (ici) pas grave,

- soit "aléatoires" (pour un tableau de classe automatique): dans ce cas, l'omission du caractère '\0' serait nettement plus gênante. De plus, comme le langage C autorise l'omission de la dimension d'un tableau lors de sa déclaration, lorsqu'elle est accompagnée d'une initialisation, il est possible d'écrire une instruction telle que :

```
char message[] = "bonjour";
```

Celle-ci réserve un tableau, nommé message, de 8 caractères (compte tenu du caractère'\0')

Initialisation de tableaux de pointeurs sur des chaînes

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

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

Cette déclaration réalise donc à la fois la création des 7 chaînes constantes correspondant aux 7 jours de la semaine et l'initialisation du tableau jour avec les 7 adresses de ces 7 chaînes. Voici un exemple employant cette déclaration (nous y avons fait appel, pour l'affichage d'une chaîne, au code de format %s, dont nous reparlerons un peu plus loin) :

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

```
int main(void)
```

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

```
int i ;
```

```
printf ("donnez un entier entre 1 et 7 : ") ;
```

```
scanf ("%d", &i) ;
```

```
printf ("Le jour numéro %d de la semaine est %s", i, jour[i-1] ) ;
```

```
return 0 ;
```

```
}
```

donnez un entier entre 1 et 7 : 3

le jour numéro 3 de la semaine est mercredi

Remarque: la situation présentée ne doit pas être confondue avec la précédente. Ici, nous avons affaire à un tableau de sept pointeurs, chacun d'entre eux désignant une chaîne constante.

POUR LIRE ET ÉCRIRE DES CHAINES

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

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

Voyez cet exemple de programme:

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

```
int main(void)
```

```
{ char nom[20], prenom[20], ville[25] ;
```

```
printf ("quelle est votre ville : ") ;
```

```
gets (ville) ;
```

```
printf ("donnez votre nom et votre prénom : ") ;
```

```
scanf ("%s %s", nom, prenom) ;
```

```
printf ("bonjour Mr %s %s qui habitez à ", prenom, nom) ;
```

```
puts (ville) ;
```

```
return 0 ;
```

```
}
```

quelle est votre ville : Rabat

donnez votre nom et votre prénom : AHMED Mohamed

bonjour Mr AHMED Mohamed qui habitez à Rabat

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

De plus, la délimitation de la chaîne lue ne s'effectue pas de la même façon avec `scanf` et `gets`. Plus précisément :

- avec le code `%s` de `scanf`, on utilise les délimiteurs "habituels" (l'espace ou la fin de ligne). Cela interdit donc la lecture d'une chaîne contenant des espaces. De plus, le caractère délimiteur n'est pas "consommé" : il reste disponible pour une prochaine lecture ;

- avec `gets`, seule la fin de ligne sert de délimiteur. De plus, contrairement à ce qui se produit avec `scanf`, ce caractère est effectivement "consommé": il ne risque pas d'être pris en compte lors d'une nouvelle lecture. Dans tous les cas, vous remarquerez que la lecture de `n` caractères implique le stockage en mémoire de `n + 1` caractères, car le caractère de fin de chaîne (`\0`) est généré automatiquement par toutes les fonctions de lecture (notez toutefois que le caractère séparateur—fin de ligne ou autre—n'est pas recopié en mémoire). Ainsi, dans notre précédent programme, il n'est pas possible (du moins pas souhaitable !) que le nom fourni en donnée contienne plus de 19 caractères.

Remarques:

1) Dans les appels des fonctions `scanf` et `puts`, les identificateurs de tableau comme nom, prénom ou ville n'ont pas besoin d'être précédés de l'opérateur `&` puisqu'ils représentent déjà des adresses.

2) La fonction `gets` fournit en résultat soit un pointeur sur la chaîne lue (c'est donc en fait la valeur de son argument), soit le pointeur nul en cas d'anomalie.

3) La fonction `puts` réalise un changement de ligne à la fin de l'affichage de la chaîne, ce qui n'est pas le cas de `printf` avec le code de format `%s`.

4) Nous nous sommes limité ici aux entrées-sorties "conversationnelles". Les autres possibilités seront examinées dans le chapitre consacré aux fichiers.

5) Si, dans notre précédent programme, l'utilisateur introduit une fin de ligne entre le nom et le prénom, la chaîne affectée à `prenom` n'est rien d'autre que... la chaîne vide ! Ceci provient de ce que la fin de ligne servant de délimiteur pour le premier `%s` n'est pas consommée et se trouve donc reprise par le `%s` suivant...

6) Compte tenu de ce que `gets` consomme la fin de ligne servant de délimiteur, alors que le code `%s` de `scanf` ne le fait pas, il n'est guère possible, dans le programme précédent, d'inverser les utilisations de `scanf` et de `gets` (en lisant la ville par `scanf` puis le nom et le prénom par `scanf`): dans ce cas, la fin de ligne non consommée par `scanf` amènerait `gets` à introduire une chaîne vide comme nom. D'une manière générale, d'ailleurs, il est préférable, autant que possible, de faire appel à `gets` plutôt qu'au code `%s` pour lire des chaînes.

FONCTIONS PORTANT SUR DES CHAINES

C dispose de nombreuses fonctions de manipulation de chaînes. Avant d'en voir les principales, voyons quelques principes généraux.

La fonction `strlen` fournit en résultat la longueur d'une chaîne dont on lui a transmis l'adresse en valeur.

La fonction `strcat(ch1,ch2) <string.h>` recopie la seconde chaîne `ch2` à la suite de la première `ch1`.

La fonction `strncat(ch1, ch2, lgmax) <string.h>` travaille de la même façon que `strcat` en offrant un contrôle sur le nombre de caractères qui seront concaténés à la chaîne `ch2`.

La fonction `strcmp(ch1, ch2) <string.h>` compare deux chaînes et fournit une valeur entière positive si `ch1 > ch2`, nulle si `ch1 = ch2` et négative si `ch1 < ch2`.

La fonction `strncmp(ch1,ch2,lgmax) <string.h>` travaille comme `strcmp` mais elle limite la comparaison au nombre `lgmax` de caractères.

Les fonctions `stricmp(ch1, ch2)` et `strnicmp(ch1, ch2, lgmax) <string.h>` travaillent comme `strcmp` et `strncmp` mais sans tenir compte de la différence entre majuscules et minuscules.

La fonction `strcpy(destin, source) <string.h>` recopie la chaîne source dans l'emplacement d'adresse destin.

La fonction `strncpy(destin, source, lgmax) <string.h>` limite la copie au nombre de caractères `lgmax`.

La fonction `strchr(ch,caractère) <string.h>` recherche dans `ch`, la première position où apparaît le caractère mentionné.

La fonction `strrchr(ch,caractère) <string.h>` opère de même mais en partant de la fin de `ch`;

Langage C

La fonction `strstr(ch,ssch)` <string.h> recherche dans `ch` la première occurrence de la sous chaîne `ssch`.

LES STRUCTURES

INTRODUCTION

Nous avons déjà vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.

La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nommé champ) se fera, cette fois, non plus par une indication de position, mais par son nom au sein de la structure.

DÉCLARATION D'UNE STRUCTURE

Exemple :

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

Celle-ci définit un modèle de structure mais ne réserve pas de variables correspondant à cette structure. Ce modèle s'appelle ici enreg et il précise le nom et le type de chacun des "champs" constituant la structure (numero, qte et prix).

Une fois un tel modèle défini, nous pouvons déclarer des "variables" du type correspondant (souvent, nous parlerons de structure pour désigner une variable dont le type est un modèle de structure).

Par exemple : struct enreg art1 ; réserve un emplacement nommé art1 "de type enreg" destiné à contenir deux entiers et un double. De manière semblable : struct enreg art1, art2 ; réserverait deux emplacements art1 et art2 du type enreg.

UTILISATION D'UNE STRUCTURE

En C il est possible d'utiliser une structure de deux manières :

- en travaillant individuellement sur chacun de ses champs,
- en travaillant de manière "globale" sur l'ensemble de la structure.

UTILISATION DES CHAMPS D'UNE STRUCTURE

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur "point" (.) suivi du nom de champ tel qu'il a été défini dans le modèle (le nom de modèle lui-même n'intervenant d'ailleurs pas).

Voici quelques exemples utilisant le modèle enreg et les variables art1 et art2 déclarées de ce type.

art1.numero = 15 ; affecte la valeur 15 au champ numero de la structure art1.

printf ("%f", art1.prix) ; la valeur du champ prix de la structure art1.

scanf ("%f", &art2.prix) ; lit une valeur qui sera affectée au champ prix de la structure

art2. Notez bien la présence de l'opérateur &.

art1.numero++ incrémente de 1 la valeur du champ numero de la structure art1.

UTILISATION GLOBALE D'UNE STRUCTURE

Il est possible d'affecter à une structure le contenu d'une structure définie à partir du même modèle. Par exemple, si les structures art1 et art2 ont été déclarées suivant le modèle enreg défini précédemment, nous pourrions écrire :

```
art1 = art2;
```

Une telle affectation globale remplace avantageusement :

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

Notez bien qu'une affectation globale n'est possible que si les structures ont été définies avec le même nom de modèle; en particulier, elle sera impossible avec des variables ayant une structure analogue mais définies sous deux noms différents.

Remarque:

L'affectation globale n'est pas possible entre tableaux. Elle l'est, par contre, entre structures. Aussi est-il possible, en créant artificiellement une structure contenant un seul champ qui est un tableau, de réaliser une affectation globale entre tableaux.

INITIALISATIONS DE STRUCTURES

Voici un exemple d'initialisation de la structure art1, au moment de sa déclaration:

```
struct enreg art1 = { 100, 285, 2000 };
```

Vous voyez que la description des différents champs se présente sous la forme d'une liste de valeurs séparées par des virgules, chaque valeur étant une constante ayant le type du champ correspondant. Là encore, il est possible d'omettre certaines valeurs.

SIMPLIFIER LA DÉCLARATION AVEC TYPEDEF

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

Exemples :

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

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

APPLICATION AUX STRUCTURES

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

```
struct enreg
{ int numero ;
  int qte ;
  double prix ;
}
typedef struct enreg s_enreg ;
s_enreg art1, art2
ou encore, plus simplement:
typedef struct
{ int numero ;
  int qte ;
  float prix ;
} s_enreg ;
```

s_enreg art1, art2 ;

IMBRICATION DE STRUCTURES

Dans nos exemples d'introduction des structures, nous nous sommes limité à une structure simple ne comportant que trois champs d'un type de base. Mais chacun des champs d'une structure peut être d'un type absolument quelconque: pointeur, tableau, structure,... De même, un tableau peut être constitué d'éléments qui sont eux-mêmes des structures. Voyons ici quelques situations classiques.

STRUCTURE COMPORTANT DES TABLEAUX

Soit la déclaration suivante:

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

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

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

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

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

La notation:

employe.heures[4] désigne le cinquième élément du tableau heures de la structure employe.

Voici un exemple d'initialisation d'une structure de type personne lors de sa déclaration:

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

TABLEAUX DE STRUCTURES

Voyez ces déclarations :

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

La structure point pourrait, par exemple, servir à représenter un point d'un plan, point qui serait défini par son nom (caractère) et ses deux coordonnées.

La structure courbe, quant à elle, pourrait servir à représenter un ensemble de 50 points du type ainsi défini. Notez bien que point est un nom de modèle de structure, tandis que courbe représente effectivement un "objet" de type "tableau de 50 éléments du type point". Si i est un entier, la notation : courbe[i].nom représente le nom du point de rang i du tableau courbe. Il s'agit donc d'une valeur de type char. Notez bien que la notation : courbe.nom[i] n'a pas de sens. De même, la notation : courbe[i].x représente la valeur du champ x de l'élément de rang i du tableau courbe.

TRANSMISSION D'UNE STRUCTURE EN ARGUMENT D'UNE FONCTION

Aucun problème particulier ne se pose. Il s'agit simplement d'appliquer ce que nous connaissons déjà.

Voici un

exemple simple:

```
#include <stdio.h>
struct enreg { int a;
double b;
}
int main(void)
{
struct enreg x;
void fct (struct enreg y);
x.a = 1; x.b = 12.5;
printf ("\navant appel fct: %d %e",x.a,x.b);
fct (x);
printf ("\n au retour dans main: %d %e", x.a, x.b);
}
void fct (struct enreg s)
{
s.a = 0;
s.b=1;
printf ("\ndans fct: %d %e", s.a, s.b);
}
avant appel fct : 1 1.25000e+01
dans fct : 0 1.00000e+00
au retour dans main : 1 1.25000e+01
```

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

TRANSMISSION DE L'ADRESSE D'UNE STRUCTURE: L'OPERATEUR ->

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

```
fct (&x) ;
```

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

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

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

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

Voici ce que pourrait devenir notre précédent exemple en employant l'opérateur -> :

```
#include <stdio.h>
struct enreg { int a;
float b;
}
int main(void)
{ struct enreg x;
```

Langage C

```
void fct (struct enreg *);
x.a = 1; x.b = 12.5;
printf ("\navant appel fct: %d %e",x.a,x.b);
fct (&x);
printf ("\n au retour dans main : %d %e", x.a, x.b);
return 0 ;
}
void fct (struct enreg * ads)
{
ads->a = 0;
ads->b = 1;
printf ("\ndans fct: %d %e", ads->a, ads->b);
}
avant appel fct: 1 1.25000e+01
dans fct: 0 1.00000e+00
au retour dans main: 0 1.00000e+00
```

EXERCICES

1) Ecrire un programme qui:

a) lit au clavier des informations dans un tableau de structures du type point défini comme suit:

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

Le nombre d'éléments du tableau sera fixé par une instruction #define.

b) affiche à l'écran l'ensemble des informations précédentes.

2) Réaliser la même chose que dans l'exercice précédent, mais en prévoyant, cette fois, une fonction pour la lecture des informations et une fonction pour l'affichage.

SOLUTIONS

```
#include <stdio.h>
#define NPOINTS 5
main()
{ struct point { int num ;
float x ;
float y ;
} ;
struct point courbe[NPOINTS] ;
int i ;
for (i=0 ; i<NPOINTS ; i++)
{ printf ("numéro : ") ; scanf ("%d", &courbe[i].num) ;
printf ("x : ") ; scanf ("%f", &courbe[i].x) ;
printf ("y : ") ; scanf ("%f", &courbe[i].y) ;
}
printf (" **** structure fournie ****\n") ;
for (i=0 ; i<NPOINTS ; i++)
printf ("numéro : %d x : %f y : %f\n",
courbe[i].num, courbe[i].x, courbe[i].y) ;
}
```

```
#include <stdio.h>
#define NPOINTS 5
struct point { int num ;
float x ;
float y ;
} ;
void lit (struct point []); /* ou void lit (struct point *) */
void affiche (struct point []); /* ou void lit (struct point *) */
main()
{
struct point courbe[NPOINTS] ;
lit (courbe) ;
affiche (courbe) ;
}
void lit (struct point courbe []) /* ou void lit (struct point * courbe) */
{
int i ;
for (i=0 ; i<NPOINTS ; i++)
{ printf ("numéro : ") ; scanf ("%d", &courbe[i].num) ;
printf ("x : ") ; scanf ("%f", &courbe[i].x) ;
printf ("y : ") ; scanf ("%f", &courbe[i].y) ;
}
}
void affiche (struct point courbe []) /* ou void affiche (struct point * courbe) */
{
int i ;
printf ("**** structure fournie ****\n") ;
for (i=0 ; i<NPOINTS ; i++)
printf ("%d %f %f\n", courbe[i].num, courbe[i].x, courbe[i].y)
```

LES FICHIERS

INTRODUCTION

Le terme fichier désigne un ensemble d'informations situé sur une "mémoire de masse" telle que le disque ou la disquette. En C, comme d'ailleurs dans d'autres langages, tous les périphériques, qu'ils soient d'archivage (disque, disquette, . . .) ou de communication (clavier, écran, imprimante, . . .), peuvent être considérés comme des fichiers. On distingue traditionnellement deux techniques de gestion de fichiers :

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

En fait, pour des fichiers disque (ou disquette), la distinction entre accès séquentiel et accès direct n'a plus véritablement de raison d'être. D'ailleurs, comme vous le verrez, en langage C, vous utiliserez les mêmes fonctions dans les deux cas (exception faite d'une fonction de déplacement de pointeur de fichier). Qui plus est, rien ne vous empêchera de mélanger les deux modes d'accès pour un même fichier.

Cependant, pour assurer une certaine progressivité à notre propos, nous avons préféré commencer par vous montrer comment travailler de manière séquentielle.

CRÉATION SÉQUENTIELLE D'UN FICHIER

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

qu'on lui fournit au clavier.

```
#include <stdio.h>
int main(void)
{ char nomfich[21] ;
  int n ;
  FILE * sortie ;
  printf ("nom du fichier à créer : ") ;
  scanf ("%20s", nomfich) ;
  sortie = fopen (nomfich, "w") ;
  do
  { printf ("donnez un entier : ") ;
    scanf ("%d", &n) ;
    if (n) fwrite(&n, sizeof(int), 1, sortie) ;
  }
  while (n) ;
  fclose (sortie) ;
  return 0 ;
}
```

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

La déclaration : FILE * sortie; signifie que sortie est un pointeur sur un objet de type FILE. Ce nom désigne en fait un modèle de structure défini dans le fichier stdio.h (par une instruction typedef, ce qui explique l'absence du mot struct).

N'oubliez pas que cette déclaration ne réserve qu'un emplacement pour un pointeur. C'est la fonction fopen qui créera effectivement une telle structure et qui en fournira l'adresse en résultat.

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

- le nom du fichier concerné, fourni sous forme d'une chaîne de caractères; ici, nous avons prévu que ce nom ne dépassera pas 20 caractères (le chiffre 21 tenant compte du caractère `\0`); notez qu'en général ce nom pourra comporter une information (chemin, répertoire,...) permettant de préciser l'endroit où se trouve le fichier.

- une indication, fournie elle aussi sous forme d'une chaîne, précisant ce que l'on souhaite faire avec ce fichier.

Ici, on trouve `w` (abréviation de `write`) qui permet de réaliser une "ouverture en écriture". Plus précisément, si le fichier cité n'existe pas, il sera créé par `fopen`. S'il existe déjà, son ancien contenu deviendra inaccessible.

Autrement dit, après l'appel de cette fonction, on se retrouve dans tous les cas en présence d'un fichier "vide".

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

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

La fonction `fwrite` possède quatre arguments précisant :

- l'adresse d'un bloc d'informations (ici `&n`),

- la taille d'un bloc, en octets : ici `sizeof(int)`; notez l'emploi de l'opérateur `sizeof` qui assure la portabilité du programme,

- le nombre de blocs de cette taille que l'on souhaite transférer dans le fichier (ici `1`),

- l'adresse de la structure décrivant le fichier (`sortie`).

Notez que, d'une manière générale, `fwrite` permet de transférer plusieurs blocs consécutifs de même taille à partir d'une adresse donnée.

Enfin, la fonction `fclose` réalise ce que l'on nomme une "fermeture" de fichier. Elle force l'écriture sur disque du tampon associé au fichier

Remarque:

1. `fopen` fournit un pointeur nul en cas d'impossibilité d'ouverture du fichier. Ce sera le cas, par exemple, si l'on cherche à ouvrir en lecture un fichier inexistant ou encore si l'on cherche à créer un fichier sur une disquette saturée.
2. `fwrite` fournit le nombre de blocs effectivement écrits. Si cette valeur est inférieure au nombre prévu, cela signifie qu'une erreur est survenue en cours d'écriture. Cela peut être, par exemple, une disquette pleine, mais cela peut se produire également lorsque l'ouverture du fichier s'est mal déroulée (et que l'on n'a pas pris soin d'examiner le code de retour de `fopen`).

LISTE SÉQUENTIELLE D'UN FICHIER

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

```
#include <stdio.h>
int main(void)
{ char nomfich[21] ;
  int n ;
  Le langage C 43
  FILE * entree ;
  printf ("nom du fichier à lister : ") ;
  scanf ("%20s", nomfich) ;
  entree = fopen (nomfich, "r") ;
  while ( fread (&n, sizeof(int), 1, entree), ! feof(entree) )
  { printf ("\n%d", n) ;
  }
  fclose (entree) ;
```

```
return 0 ;  
}
```

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

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

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

dont les arguments sont comparables à ceux de `fwrite`. Mais, cette fois, la condition d'arrêt de la boucle est: `feof (entree)`

Celle-ci prend la valeur vrai (c'est-à-dire 1) lorsque la fin du fichier a été rencontrée. Notez bien qu'il n'est pas suffisant d'avoir lu le dernier octet du fichier pour que cette condition prenne la valeur vrai. Il est nécessaire d'avoir tenté de lire au-delà; c'est ce qui explique que nous ayons examiné cette condition après l'appel de `fread` et non avant.

Remarques:

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

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

2) N'oubliez pas que le premier argument des fonctions `fwrite` et `fread` est une adresse. Ainsi, lorsque vous aurez affaire à un tableau, il faudra utiliser simplement son nom (sans le faire précéder de `&`), tandis qu'avec une structure il faudra effectivement utiliser l'opérateur `&` pour en obtenir l'adresse. Dans ce dernier cas, même si l'on ne cherche pas à rendre son programme portable, il sera préférable d'utiliser l'opérateur `sizeof` pour déterminer avec certitude la taille des blocs correspondants.

3) `fread` fournit le nombre de blocs effectivement lus (et non pas le nombre d'octets lus). Ce résultat peut être inférieur au nombre de blocs demandés soit lorsque l'on a rencontré une fin de fichier, soit lorsqu'une erreur de lecture est apparue. Dans notre précédent exemple d'exécution, `fread` fournit toujours 1, sauf la dernière fois où elle fournit 0.

L'ACCÈS DIRECT

Les fonctions `fread` et `fwrite` lisent ou écrivent un certain nombre d'octets dans un fichier, à partir d'une "position courante". Cette dernière n'est rien d'autre qu'un "pointeur" dans le fichier, c'est-à-dire un nombre précisant le rang du prochain octet à lire ou à écrire. Après chaque opération de lecture ou d'écriture, ce pointeur se trouve incrémenté du nombre d'octets transférés. C'est ainsi que l'on réalise un accès séquentiel au fichier. Mais il est également possible d'agir directement sur ce pointeur de fichier à l'aide de la fonction `fseek`. Cela permet ainsi de réaliser des lectures ou des écritures en n'importe quel point du fichier, sans avoir besoin de parcourir toutes les informations qui précèdent. On peut ainsi réaliser ce que l'on nomme généralement un "accès direct".

ACCES DIRECT EN LECTURE SUR UN FICHER EXISTANT

Voici un programme qui permet d'accéder à n'importe quel entier d'un fichier du type de ceux que pouvait créer notre programme du paragraphe précédent

```
#include <stdio.h>  
int main(void)  
{ char nomfich[21];  
int n ;  
long num ;
```

```
FILE * entree ;
printf ("nom du fichier à consulter : ");
scanf ("%20s", nomfich) ;
entree = fopen (nomfich, "r") ;
while ( printf (" numéro de l'entier recherché : "),scanf ("%ld", &num), num )
{ fseek (entree, sizeof(int)*(num-1), SEEK_SET)
fread (&n, sizeof(int), 1, entree) ;
printf (" valeur : %d \n", n) ;
}
fclose (entree) ;
return 0 ;
}
```

La principale nouveauté réside essentiellement dans l'appel de la fonction `fseek`:

```
fseek ( entree, sizeof(int)*(num-1), SEEK_SET);
```

Cette dernière possède trois arguments:

- le fichier concerné (désigné par le pointeur sur une structure de type `FILE`, tel qu'il a été fourni par `fopen`),
- un entier de type long spécifiant la valeur que l'on souhaite donner au pointeur de fichier. Il faut noter que l'on dispose de trois manières d'agir effectivement sur le pointeur, le choix entre les trois étant fait par l'argument suivant,
- le choix du mode d'action sur le pointeur de fichier: il est défini par une constante entière. Les valeurs suivantes sont prédéfinies dans `<stdio.h>`:

* `SEEK_SET` (en général 0): le second argument désigne un déplacement (en octets) depuis le début du fichier.

* `SEEK_CUR` (en général 1) : le second argument désigne un déplacement exprimé à partir de la position courante; il s'agit donc en quelque sorte d'un déplacement relatif dont la valeur peut, le cas échéant, être négative.

* `SEEK_END` (en général 2): le second argument désigne un déplacement depuis la fin du fichier.

Ici, il s'agit de donner au pointeur de fichier une valeur correspondant à l'emplacement d'un entier (`sizeof(int)`)

octets) dont l'utilisateur fournit le rang. Il est donc naturel de donner au troisième argument la valeur 0. Notez, au passage, la "formule" : `sizeof(int) * (num-1)` qui se justifie par le fait que nous nous sommes convenu que, pour l'utilisateur, le premier entier du fichier porterait le rang 1 et non 0.

LES POSSIBILITES DE L'ACCES DIRECT

Outre les possibilités de "consultation immédiate" qu'il procure, l'accès direct facilite et accélère les opérations de mise à jour d'un fichier. Mais, de surcroît, l'accès direct permet de remplir un fichier de façon quelconque.

Ainsi, nous pourrions constituer notre fichier d'entiers en laissant l'utilisateur les fournir dans l'ordre de son choix, comme dans cet exemple de programme:

```
#include <stdio.h>
int main(void)
{ char nomfich[21] ;
FILE * sortie ;
long num ;
int n ;
printf ("nom fichier : ") ;
scanf ("%20s",nomfich) ;
Le langage C 45
sortie = fopen (nomfich, "w") ;
while (printf("\nrang de l'entier : "), scanf("%ld",&num), num)
```

```
{printf ("valeur de l'entier : ");  
scanf ("%d", &n) ;  
fseek (sortie, sizeof(int)*(num-1), SEEK_SET);  
fwrite (&n, sizeof(int), 1, sortie) ;  
}  
fclose(sortie) ;  
return 0 ;  
}
```

Or il faut savoir qu'avec beaucoup de systèmes, dès que vous écrivez le *énième* octet d'un fichier, il y a automatiquement réservation de la place de tous les octets précédents; leur contenu, par contre, doit être considéré comme étant aléatoire.

Dans ces conditions, vous voyez que, à partir du moment où rien n'impose à l'utilisateur de ne pas "laisser de trous" lors de la création du fichier, il faudra être en mesure de repérer ces trous lors d'éventuelles consultations ultérieures du fichier. Plusieurs techniques existent à cet effet:

- on peut, par exemple, avant d'exécuter le programme précédent, commencer par "initialiser" tous les emplacements du fichier à une valeur spéciale, dont on sait qu'elle ne pourra pas apparaître comme valeur effective,

- on peut aussi "gérer une table des emplacements inexistants", cette table devant alors être conservée (de préférence) dans le fichier lui-même.

D'autre part, il faut bien voir que l'accès direct n'a d'intérêt que lorsque l'on est en mesure de fournir le rang de l'emplacement concerné. Ce n'est pas toujours possible. Ainsi, si l'on considère ne serait ce qu'un simple fichier de type répertoire téléphonique, on voit qu'en général on cherchera à accéder à une personne par son nom plutôt que par son numéro d'ordre dans le fichier. Cette contrainte qui semble imposer une recherche séquentielle peut être contournée par la création de ce que l'on nomme un "index", c'est-à-dire une table de correspondance entre un nom d'individu et sa position dans le fichier.

Nous n'en dirons pas plus sur ces méthodes spécifiques de gestion de fichiers qui sortent du cadre de ce polycopié.

EN CAS D'ERREUR

a) Erreur de pointage

Il faut bien voir que le positionnement dans le fichier se fait sur un octet de rang donné, et non, comme on pourrait le préférer, sur un "bloc" (ou "enregistrement") de rang donné. D'ailleurs, n'oubliez pas qu'en général cette notion d'enregistrement n'est pas exprimée de manière intrinsèque au sein du fichier. Ainsi, dans notre programme précédent, vous pourriez, par mégarde, utiliser la formule suivante:

```
sizeof(int) * num - 1
```

laquelle vous positionnerait systématiquement "à cheval" entre le dernier octet d'un entier et le premier du suivant. Bien entendu, les résultats obtenus seraient quelque peu fantaisistes.

Cette remarque prend encore plus d'acuité lorsque vous créez un fichier à partir de structures. Dans ce cas, nous ne saurions trop vous conseiller d'avoir systématiquement recours à l'opérateur `sizeof` pour déterminer la taille réelle de ces structures.

b) Tentative de positionnement hors fichier

Lorsque l'on accède ainsi directement à l'information d'un fichier, le risque existe de tenter de se positionner... en dehors du fichier. En principe, la fonction `fseek` fournit:

- la valeur 0 lorsque le positionnement s'est déroulé correctement,
- une valeur quelconque dans le cas contraire.

Toutefois, beaucoup d'implémentation ne respectent pas la norme à ce sujet. Dans ces conditions, il nous paraît plus raisonnable de programmer une protection efficace en déterminant, en début de programme, la taille effective du fichier à consulter. Pour cela, il suffit de vous positionner en fin de fichier avec `fseek` puis de faire appel à la fonction `ftell` qui vous restitue la position courante du pointeur de fichier. Ainsi, dans notre précédent programme, nous pourrions introduire les instructions:

```
long taille;
```

```
.....
```

```
fseek ( entree, 0, SEEK_END );
```

```
taille = ftell (entree);
```

Il suffit alors de vérifier que la position de l'enregistrement demandé (ici: `sizeof(int)*(num-1)`) est bien inférieure à la valeur de `taille` pour éviter tout problème.

LES ENTRÉES-SORTIES FORMATÉES ET LES FICHIERS DE TEXTE

Nous venons de voir que les fonctions `fread` et `fwrite` réalisent un transfert d'information (entre mémoire et fichier) que l'on pourrait qualifier de "brut", dans le sens où il se fait sans aucune transformation de l'information. Les octets qui figurent dans le fichier sont des "copies conformes" de ceux qui apparaissent en mémoire. Mais, en langage C, il est également possible d'accompagner ces transferts d'information d'opérations de "formatage" analogues à celles que réalisent `printf` ou `scanf`.

Les fichiers concernés par ces opérations de formatage sont alors ce que l'on a coutume d'appeler des "fichiers de type texte" ou encore des "fichiers de texte". Ce sont des fichiers que vous pouvez manipuler avec un éditeur quelconque, un traitement de texte quelconque ou, plus simplement, lister par les commandes appropriées du système d'exploitation (`TYPE` ou `PRINT` sous DOS).

Dans de tels fichiers, chaque octet représente un caractère. Généralement, on y trouve des caractères de fin de ligne (`\n`), de sorte qu'ils apparaissent comme une suite de lignes. Les fonctions permettant de travailler avec des fichiers de texte ne sont rien d'autre qu'une généralisation aux fichiers de celles que nous avons déjà rencontrées pour les entrées-sorties conversationnelles. Nous nous contenterons donc d'en fournir une brève liste:

```
fscanf ( fichier, format, liste_d'adresses )
```

```
fprintf ( fichier, format, liste_d'expressions )
```

```
fgetc ( fichier )
```

```
fputc ( entier, fichier )
```

```
fgets ( chaîne, lgmax, fichier )
```

```
fputs ( chaîne, fichier )
```

La signification de leurs arguments est la même que pour les fonctions conversationnelles correspondantes. Seule `fgets` comporte un argument entier (`lgmax`) de contrôle de longueur. Il précise le nombre maximal de caractères (y compris le `\0` de fin) qui seront placés dans la chaîne.

Leur valeur de retour est la même que pour les fonctions conversationnelles. Cependant, il nous faut apporter quelques indications supplémentaires qui ne se justifiaient pas pour des entrées-sorties conversationnelles, à savoir que la valeur de retour fournie par `fgetc` est du type `int` (et non, comme on pourrait le croire, de type `char`).

Lorsque la fin de fichier est atteinte, cette fonction fournit la valeur `EOF` (constante prédéfinie dans `<stdio.h>` - en général `-1`). La fin de fichier n'est détectée que lorsque l'on cherche à lire un caractère alors qu'il n'y en a plus de disponible, et non pas, dès que l'on a lu le dernier caractère. D'autre part, notez bien que cette convention fait, en quelque sorte, double emploi avec la fonction `feof`.

D'une manière générale, toutes les fonctions présentées ci-dessus fournissent une valeur de retour bien définie en cas de fin de fichier ou d'erreur.

Remarque importante:

A priori, on peut toujours dire que n'importe quel fichier, quelle que soit la manière dont l'information y a été représentée, peut être considéré comme une suite de caractères. Bien entendu, si l'on cherche à "lister", par exemple, le contenu d'un fichier tel que celui créé dans le paragraphe 2.1 (suite d'entiers), le résultat risque d'être sans signification (on obtiendra une suite de caractères apparemment quelconques, sans rapport aucun avec les nombres enregistrés).

Mais, sans aller jusqu'à le lister, on peut se demander s'il ne serait pas possible de le recopier, à l'aide d'une répétition de `fgetc` et de `fputc`. Or cela semble effectivement possible puisque ces fonctions se contentent de prélever un caractère (donc un octet) et de le recopier tel quel. Ainsi, quel que soit le contenu de l'octet lu, on le retrouvera dans le fichier de sortie.

En réalité, cela n'est que partiellement vrai car certains systèmes (DOS en particulier) distinguent les fichiers de

texte des autres (qu'ils appellent parfois "fichiers binaires"); plus précisément, lors de l'ouverture du fichier, on peut spécifier si l'on souhaite ou non considérer le contenu du fichier comme du texte. Cette distinction est en fait motivée par le fait que le caractère de fin de ligne (\n) possède, sur ces systèmes, une représentation particulière obtenue par la succession de deux caractères (retour chariot \r, suivi de fin de ligne \n). Or, dans ce cas, pour qu'un programme C puisse ne "voir" qu'un seul caractère de fin de ligne et qu'il s'agisse bien de \n, il faut opérer un traitement particulier consistant à :

- remplacer chaque occurrence de ce couple de caractères par \n, dans le cas d'une lecture,
- remplacer chaque demande d'écriture de \n par l'écriture de ce couple de caractères.

Bien entendu, de telles substitutions ne doivent pas être réalisées sur de "vrais fichiers binaires". Il faut donc bien pouvoir opérer une distinction au sein du programme. Cette distinction se fait au moment de l'ouverture du fichier, en ajoutant l'une des lettres t (pour "texte") ou b (pour "binaire") au mode d'ouverture. En général, dans les implémentations où l'on distingue les fichiers de texte des autres, les fonctions d'entrées-sorties formatées refusent de travailler avec un fichier qui n'a pas été spécifié de ce type lors de son ouverture.

LES DIFFÉRENTES POSSIBILITÉS D'OUVERTURE D'UN FICHIER

Dans nos précédents exemples, nous n'avons utilisé que les modes w et r. Nous vous fournissons ici la liste des différentes possibilités offertes par fopen.

r : lecture seulement; le fichier doit exister.

w: écriture seulement. Si le fichier n'existe pas, il est créé. S'il existe, son (ancien) contenu est perdu.

a: écriture en fin de fichier (append). Si le fichier existe déjà, il sera "étendu". S'il n'existe pas, il sera créé - on se ramène alors au mode w.

r+: mise à jour (lecture et écriture). Le fichier doit exister. Notez qu'alors il n'est pas possible de réaliser une lecture à la suite d'une écriture ou une écriture à la suite d'une lecture, sans positionner le pointeur de fichier par fseek. Il est toutefois possible d'enchaîner plusieurs lectures ou écritures consécutives (de façon séquentielle).

w+: création pour mise à jour. Si le fichier existe, son (ancien) contenu sera détruit. S'il n'existe pas, il sera créé.

Notez que l'on obtiendrait un mode comparable à w+ en ouvrant un fichier vide (mais existant) en mode r+.

a+: extension et mise à jour. Si le fichier n'existe pas, il sera créé. S'il existe, le pointeur sera positionné en fin de fichier.

t ou b: lorsque l'implémentation distingue les fichiers de texte des autres, il est possible d'ajouter l'une de ces deux lettres à chacun des 6 modes précédents. La lettre t précise que l'on a affaire à un fichier de texte; la lettre b précise que l'on a affaire à un fichier "binaire" .

LES FICHIERS PRÉDÉFINIS

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

stdin: unité d'entrée (par défaut, le clavier)

stdout: unité de sortie (par défaut, l'écran)

stderr: unité d'affichage des messages d'erreurs (par défaut, l'écran)

On trouve parfois également:

stdaux: unité auxiliaire

stdprt: imprimante

Les deux premiers fichiers correspondent aux unités standard d'entrée et de sortie d'un programme.

Lorsque vous

exécutez un programme depuis le système, vous pouvez éventuellement "rediriger" ces fichiers. Par exemple, la

commande système suivante (valable à la fois sous UNIX et sous DOS)

TRUC < DONNEES > RESULTATS

exécute le programme TRUC, en utilisant comme unité d'entrée le fichier DONNEES et comme unité de sortie le fichier RESULTATS.

Dans ces conditions, une instruction telle que, par exemple, fgetchar deviendrait équivalente à fgetc(fich) où fich serait un flux obtenu par appel à fopen. De même, scanf(...) deviendrait équivalent à fscanf(fich, . . .), etc.

Notez bien qu'au sein du programme même il n'est pas possible de savoir si un fichier prédéfini a été redirigé au moment du lancement du programme; autrement dit, lorsqu'une fonction comme fgetchar ou scanf lit des informations, elle ne peut absolument pas savoir si ces dernières proviennent du clavier ou d'un fichier.

EXERCICES

1) Ecrire un programme permettant d'afficher le contenu d'un fichier texte en numérotant les lignes. Les lignes seront supposées ne jamais comporter plus de 80 caractères.

2) Ecrire un programme permettant de créer séquentiellement un fichier "répertoire" comportant pour chaque personne :

- nom (20 caractères maximum),
- prénom (15 caractères maximum),
- âge (entier),
- numéro de téléphone (11 caractères maximum).

Les informations relatives aux différentes personnes seront lues au clavier.

3) Ecrire un programme permettant, à partir du fichier créé par l'exercice précédent, de retrouver les informations correspondant à une personne de nom donné.

4) Ecrire un programme permettant, à partir du fichier créé par le programme de l'exercice 2, de retrouver les informations relatives à une personne de "rang" donné (par accès direct).

SOLUTION

```
1)
#include <stdio.h>
#define LGMAX 81
main()
{ char nomfich[21] ; /* nom de fichier */
  FILE * entree ;
  int num = 1 ; /* numéro de ligne */
  char ligne [LGMAX] ;
  printf ("donnez le nom du fichier à lister : ");
  scanf ("%20s", nomfich) ;
  entree = fopen (nomfich, "r")1 ;
  printf (" **** liste du fichier %s ****\n", nomfich) ;
  while ( fgets (ligne, LGMAX, entree) )
  { printf ("%5d ", num++) ;
    printf ("%s", ligne) ;
  }
}
```

2)

Langage C

```
#include <stdio.h>
#define LGNOM 20
#define LGPRENOM 15
#define LGTEL 11
main()
{
char nomfich[21] ; /* nom de fichier */
FILE * sortie ;
struct { char nom [LGNOM+1] ;
char prenom [LGPRENOM+1] ;
int age ;
char tel [LGTEL+1] ;
} bloc ;
printf ("donnez le nom du fichier à créer : ");
gets (nomfich) ;
sortie = fopen (nomfich, "w") ;
printf (" --- pour finir la saisie, donnez un nom 'vide' ---\n") ;
while ( printf ("nom : "), gets (bloc.nom), strlen(bloc.nom) )
{ printf ("prénom : ") ;
gets (bloc.prenom) ;
printf ("age : ") ;
scanf ("%d", &bloc.age) ; getchar() ;
printf ("téléphone : ") ;
gets (bloc.tel) ;
fwrite (&bloc, sizeof(bloc), 1, sortie) ;
}
fclose (sortie) ;
}
3)
```

```
#include <string.h>
#define LGNOM 20
#define LGPRENOM 15
#define LGTEL 11
main()
{
char nomfich[21] ; /* nom de fichier */
FILE * entree ;
struct { char nom [LGNOM+1] ;
char prenom [LGPRENOM+1] ;
int age ;
char tel [LGTEL+1] ;
} bloc ;
char nomcher [LGNOM+1] ; /* nom recherché */
int trouve ; /* indicateur nom trouvé */
printf ("donnez le nom du fichier à consulter : ");
gets (nomfich) ;
entree = fopen (nomfich, "r") ;
printf (" quel nom recherchez vous : ") ;
gets (nomcher) ;
trouve = 0 ;
```

Langage C

```
do
{ fread (&bloc, sizeof(bloc), 1, entree) ;
if ( strcmp (nomcher, bloc.nom)==0 ) trouve = 1 ;
}
while ( (!trouve) && (!feof(entree)) ) ;
if (trouve)
{ printf ("prénom : %s\n", bloc.prenom) ;
printf ("age : %d\n", bloc.age) ;
printf ("téléphone : %s\n", bloc.tel) ;
}
else printf ("-- ce nom ne figure pas au fichier --") ;
}
```

4)

```
#define LGNOM 20
#define LGPRENOM 15
#define LGTEL 11
main()
{
char nomfich[21] ; /* nom de fichier */
FILE * entree ;
struct { char nom [LGNOM+1] ;
char prenom [LGPRENOM+1] ;
int age ;
char tel [LGTEL+1] ;
} bloc ;
int num ; /* numéro de bloc recherché */
long taille, /* taille du fichier en octets */
pos ; /* position dans le fichier */
printf ("donnez le nom du fichier à consulter : ");
gets (nomfich) ;
entree = fopen (nomfich, "r") ;
fseek (entree, 0, SEEK_END) ; taille = ftell(entree) ;
printf (" quel numéro recherchez vous : ") ;
scanf ("%d",&num) ;
pos = num * sizeof(bloc) ;
if ( num<0 || pos >= taille )
printf ("-- ce numéro ne figure pas dans le fichier ") ;
else
{ fseek (entree, pos, 0 ) ;
fread (&bloc, sizeof(bloc), 1, entree) ;
printf ("nom : %s\n", bloc.nom) ;
printf ("prénom : %s\n", bloc.prenom) ;
printf ("age : %d\n", bloc.age) ;
printf ("téléphone : %s\n", bloc.tel) ;
}
}
.
```