LE LANGAGE C

1. GÉNÉRALITÉS SUR LE LANGAGE C

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C.
Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont: l'édition, la compilation, l'édition de liens et l'exécution.

1.1 PRÉSENTATION PAR L'EXEMPLE DE QUELQUES INSTRUCTIONS DU LANGAGE C

1.1.1 Un exemple de programme en langage C

Voici un premier exemple très simple de programme en langage C.

```
#include <stdio.h>
int main(void)
{       int i ;
        double x ;
        i=0 ;
        x=3.1415926535
        printf(''Bonjour tout le monde \n'') ;
        printf(''i vaut : %d et x vaut :%f'',i,x) ;
    return 0 ;
}
```

1.1.2 Structure d'un programme en langage C

La ligne: int main(void) se nomme un "en-tête". Elle précise que ce qui sera décrit à sa suite est en fait le "programme principal" (int main). Lorsque nous aborderons l'écriture des fonctions en C, nous verrons que celles-ci possèdent également un tel en-tête; ainsi, en C, le programme principal apparaîtra en fait comme une fonction dont le nom (int main) est imposé.

Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades "{" et "}". On dit que les instructions situées entre ces accolades forment un "bloc". Ainsi peut-on dire que la fonction main est constituée d'un en-tête et d'un bloc; il en ira de même pour toute fonction C. Notez qu'un bloc (comme en Pascal) peut lui-même contenir d'autres blocs. En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions (ce qui est le cas du Pascal).

1.1.3 Déclarations

Les deux instructions : int i; double x; sont des "déclarations".

La première précise que la variable nommée i est de type int, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C il existe plusieurs types d'entiers.

L'autre déclaration précise que la variables x sont de type double c'est-à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C il existe plusieurs types flottants.

En C, comme en Pascal, les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme (on devrait plutôt dire: au début de la fonction main). Il en ira de même pour toutes les variables définies dans une fonction; on les appelle "variables locales" (en toute rigueur, les variables définies dans notre exemple sont des variables locales de la fonction main). Nous verrons également (dans le chapitre consacré aux fonctions) qu'on peut définir des variables en dehors de toute fonction; on parlera alors de variables globales.

1.1.4 Pour écrire des informations: la fonction printf

L'instruction : printf ("Bonjour\n") ; appelle en fait une fonction "prédéfinie" (fournie avec le langage, et donc que vous n'avez pas à écrire vous-même) nommée printf. Ici, cette fonction reçoit un argument qui est : "Bonjour\n"

Les guillemets servent à délimiter une "chaîne de caractères" (suite 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", c'est-à-dire ne possédant pas de "graphisme" particulier.

Notez que, apparemment, bien que printf soit une fonction, nous n'utilisons pas sa valeur. Nous aurons l'occasion de revenir sur ce point, propre au langage C. Pour l'instant, admettez que nous pouvons, en C, utiliser une fonction comme ce que d'autres langages nomment une "procédure" ou un "sous-programme".

1.1.5 La fonction return

On la reverra plus tard, notez simplement qu'elle est obligatoire.

1.1.6 Les directives à destination du préprocesseur

La première ligne de notre programme : #include <stdio.h>
est en fait un peu particulière. Il s'agit d'une "directive" qui est prise en compte avant la traduction (compilation) du programme. Ces directives, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne. Leur emplacement au sein du programme n'est soumis à aucune contrainte (mais une directive ne s'applique qu'à la partie du programme qui lui succède). D'une manière générale, il est préférable de les placer au début, comme nous l'avons fait ici.
La directive demande en fait d'introduire (avant compilation) des instructions (en langage C) situées dans le fichier stdio.h. Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. En général, il est indispensable d'incorporer stdio.h.

1.2 QUELQUES RÈGLES D'ÉCRITURE

Ce paragraphe vous expose un certain nombre de règles générales intervenant dans l'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.

1.2.1 Les identificateurs

Les identificateurs servent à désigner les différents "objets" manipulés par le programme: variables, fonctions, etc Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les lettres ou les chiffres, le premier d'entre eux étant nécessairement une lettre.

1.2.2 Les mots clés

Certains "mots clés" sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. En voici la liste, classée par ordre alphabétique.

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

1.2.3 Les séparateurs

Dans notre langue écrite, les différents mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne. Il en va quasiment de même en langage C dans lequel les règles vont donc paraître naturelles. Ainsi, 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 : intx,y
En revanche, vous pourrez écrire indifféremment : int n,compte,total ou plus lisiblement : int n, compte, total

1.2.4 Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.
Ils sont formés de caractères quelconques placés entre les symboles /* et */. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de commentaires:

```
//      programme de calcul de racines carrées

        /* commentaire s'étendant
        sur plusieurs lignes
        de programme source     */
```

1.3 CRÉATION D'UN PROGRAMME EN LANGAGE C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir: édition, compilation et édition de liens.

1.3.1 L'édition du programme

L'édition du programme (on dit aussi parfois "saisie") consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme: on parle alors de "programme source". En général, ce texte sera conservé dans un fichier que l'on nommera "fichier source".

1.3.2 La compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En langage C, compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes.Le résultat de compilation porte le nom de module objet.

1.3.3 L'édition de liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque, au moins, les différents modules objet correspondant aux fonctions prédéfinies (on dit aussi "fonctions standard") utilisées par votre programme (comme printf, scanf, sqrt).
C'est effectivement le rôle de l'éditeur de liens que d'aller rechercher dans la "bibliothèque standard" les modules objet nécessaires. Le résultat de l'édition de liens est ce que l'on nomme un "programme exécutable", c'est-à-dire un ensemble autonome d'instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu'il soit nécessaire de faire appel à un quelconque composant de l'environnement de programmation en C.

Exercice :
Ecrire un programme permettant d'afficher « bonjour » à l'écran.

2. LES TYPES DE BASES DU LANGAGE C

Les types char, int et double que nous avons déjà rencontrés sont souvent dits "scalaires" ou "simples", car, à un instant donné, une variable d'un tel type contient une seule valeur. Ils s'opposent aux types "structurés" (on dit aussi "agrégés") qui correspondent à des variables qui, à un instant donné, contiennent plusieurs valeurs (de même type ou non).

2.1 LA NOTION DE TYPE

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

L'ordinateur, compte tenu de sa technologie (actuelle !), 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, on voit qu'il ne suffit pas de connaître le contenu d'un emplacement de la mémoire (d'un ou de plusieurs octets) pour être en mesure de lui attribuer une signification. Par exemple, si "vous" savez qu'un octet contient le "motif binaire" suivant : 01001101 vous pouvez considérer que cela représente le nombre entier 77 (puisque le motif ci-dessus correspond à la représentation en base 2 de ce nombre). Mais pourquoi cela représenterait-il un nombre? En effet, toutes les informations (nombres entiers, nombres réels, nombres complexes, caractères, instructions de programme en langage machine, graphiques...) devront, au bout du compte, être codées en binaire. Dans ces conditions, les huit bits ci-dessus peuvent peut-être représenter un caractère; dans ce cas, si nous connaissons la convention employée sur la machine concernée pour représenter les caractères, nous pouvons lui faire correspondre un caractère donné (par exemple M, dans le cas du code ASCII). On comprend donc qu'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. Qui plus est, en général, il ne sera même pas possible de "traiter" cette information. Par exemple, pour additionner deux informations, il faudra savoir quel codage a été employé afin de pouvoir mettre en œuvre les "bonnes" instructions (en langage machine).

D'une manière générale, la notion de type, telle qu'elle existe dans les langages évolués, sert à régler (entre autres choses) les problèmes que nous venons d'évoquer.

Les types de base du langage C se répartissent en trois grandes catégories en fonction de la nature des informations qu'ils permettent de représenter:

- nombres entiers (mot clé int),
- nombres flottants (mot clé double),
- caractères (mot clé char); nous verrons qu'en fait char apparaît (en C) comme un cas particulier de int.

2.2 LES TYPES ENTIERS

2.2.1 Leur représentation en mémoire

Le mot clé int correspond à la représentation de nombres entiers relatifs. Pour ce faire: un bit est réservé pour représenter le signe du nombre (en général 0 correspond à un nombre positif); les autres bits servent à représenter la valeur absolue du nombre.

2.2.2 Les différents types d'entiers

Le C prévoit que, sur une machine donnée, on puisse trouver jusqu'à trois "tailles" différentes d'entiers, désignées par les mots clés suivants:

- short

- int (c'est celui que nous utiliserons systématiquement),

- long int .

Chaque taille impose naturellement ses limites. Toutefois, ces dernières dépendent, non seulement du mot clé considéré, mais également de la machine utilisée: tous les int n'ont pas la même taille sur toutes les machines ! Fréquemment, deux des trois mots clés correspondent à une même taille (par exemple, sur PC, short et int correspondent à 16 bits, tandis que long correspond à 32 bits).

A titre indicatif, avec 16 bits, on représente des entiers s'étendant de -32 768 à 32 767; avec 32 bits, on peut couvrir les valeurs allant de -2 147 483 648 à 2 147 483 647.

Remarque: en toute rigueur, chacun des trois types (short, int et long) peut être nuancé par l'utilisation du "qualificatif" unsigned (non signé). Dans ce cas, il n'y a plus de bit réservé au signe et on ne représente plus que des nombres positifs. Son emploi est réservé à des situations particulières.

2.2.3 Notation des constantes entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire simplement sous forme décimale, avec ou sans signe, comme dans ces exemples: -533 ou 48...

2.3 LES TYPES FLOTTANTS

2.3.1 Les différents types et leur représentation en mémoire

Les types "flottants" permettent de représenter, de manière approchée, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation "scientifique" (ou "exponentielle") bien connue qui consiste à écrire un nombre sous la forme $1.5.10^{22}$ ou $0.472\ .10^{-8}$; dans une telle notation, on nomme "mantisses" les quantités telles que 1.5 ou 0.472 et "exposants" les quantités telles que 22 ou -8.

Le C prévoit trois types de flottants correspondant à des tailles différentes: float, double (c’est celui que l’on emploie systématiquement) et long double.

La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable. En revanche, il est important de noter que de telles représentations sont caractérisées par deux éléments:

- la précision: lors du codage d'un nombre décimal quelconque dans un type flottant, il est nécessaire de ne conserver qu'un nombre fini de bits. Or la plupart des nombres s'exprimant avec un nombre limité de décimales ne peuvent pas s'exprimer de façon exacte dans un tel codage. On est donc obligé de se limiter à une représentation approchée en faisant ce qu'on nomme une "erreur de troncature". Quelle que soit la machine utilisée, on est assuré que cette erreur (relative) ne dépassera pas 10^{-6} pour le type float et 10^{-10} pour le type long double.

- le domaine couvert, c'est-à-dire l'ensemble des nombres représentables à l'erreur de troncature près. Là encore, quelle que soit la machine utilisée, on est assuré qu'il s'étendra au moins de 10^{-37} à 10^{+37}.

2.3.2 Notation des constantes flottantes

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations: décimale et exponentielle.La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais, bien sûr, pas toutes les deux en même temps !). En voici quelques exemples corrects:

12.43 -0.38 -.38 4. .27

Par contre, la constante 47 serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant).

La notation exponentielle utilise la lettre e (ou E) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès que l'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents):

4. 25E4 4 . 25e+4 42. 5E3
54.27E-32 542.7E-33 5427e-34

Par défaut, toutes les constantes sont créées par le compilateur dans le type double.

2.4 LES TYPES CARACTÈRES

2.4.1 La notion de caractère en langage C

Le C permet, comme le Pascal, de manipuler des caractères codés en mémoire sur un octet. Bien entendu, le code employé, ainsi que l'ensemble des caractères représentables, dépend de l'environnement de programmation utilisé (c'est-à-dire à la fois de la machine concernée et du compilateur employé). Néanmoins, on est toujours certain de disposer des lettres (majuscules et minuscules), des chiffres, des signes de ponctuation et des différents séparateurs (en fait, tous ceux que l'on emploie pour écrire un programme !). En revanche, les caractères "nationaux" (caractères accentués ou ç) ou les caractères "semi-graphiques" ne se rencontrent pas dans tous les environnements.

Par ailleurs, la notion de caractère en C dépasse celle de caractère imprimable, c'est-à-dire auquel est obligatoirement associé un graphisme (et qu'on peut donc imprimer ou afficher sur un écran). C'est ainsi qu'il existe certains "caractères" de changement de ligne, de tabulation, d'activation d'une alarme sonore (cloche),... Nous avons d'ailleurs déjà utilisé le premier (sous la forme \n).

2.4.2 Notation des constantes caractères

Les constantes de type "caractère", lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quotes) le caractère voulu, comme dans ces exemples :'Y', ‘+', '$'
Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère " \caractères (\, ', " et ?) qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteur qui les empêche d'être notés de manière classique entre deux apostrophes. Voici la liste de ces caractères.

NOTATION EN C	CODE ASCII (hexadécimal)	ABREVIATION USUELLE	
\a	07	BEL	cloche ou bip (alert ou audible bell)
\b	08	BS	Retour arrière (Backspace)
\f	0C	FF	Saut de page (Form Feed)
\n	0A	LF	Saut de Ligne (Line Feed)
\r	0D	CR	Retour chariot (Carriage Return)
\t	09	HT	Tabulation horizontaLe (HorizontaL Tab)
\v	0B	VT	Tabutation verticale (VerticaL Tab)
\\	SC	\	
\’	2C	‘	
\ ’’	22	"	
\?	3F	?	

2.5 INITIALISATION ET CONSTANTES

La directive
#define n 100 permet de définir une constante s’appelant n et valant dans tout le programme 100.

3. LES OPÉRATEURS ET LES EXPRESSIONS EN LANGAGE C

3.1 L'ORIGINALITÉ DES NOTIONS D'OPÉRATEUR ET D'EXPRESSION EN LANGAGE C

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). Mais, de surcroît, le C dispose d'un important éventail d'opérateurs originaux d'affectation et d'incrémentation.

3.2 LES OPÉRATEURS ARITHMÉTIQUES EN C

3.2.1 Présentation des opérateurs

Comme tous les langages, C dispose d'opérateurs classiques "binaires" (c'est-à-dire portant sur deux "opérandes"), à savoir l'addition (+), la soustraction (-), la multiplication (*) et la division (/), ainsi que d'un opérateur "unaire" (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté - (comme dans -n ou -x+y).

Les opérateurs binaires ne sont a priori définis que pour deux opérandes ayant le même type parmi: int, long int, float, double, long double et ils fournissent un résultat de même type que leurs opérandes.

De plus, il existe un opérateur de "modulo" noté % qui ne peut porter que sur des entiers et qui fournit le reste de la division de son premier opérande par son second. Par exemple, 11%4 vaut 3, 23%6 vaut 5...

Notez bien qu'en C le quotient de deux entiers fournit un entier. Ainsi 5/2 vaut 2; en revanche, le quotient de deux flottants (noté, lui aussi, /) est bien un flottant (5.0/2.0 vaut bien approximativement 2.5).

Remarque: Il n'existe pas d'opérateur d'élévation à la puissance.

3.2.2 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En C, comme dans les autres langages, les règles sont "naturelles" et rejoignent celles de l'algèbre traditionnelle

3.2.3 Le cas du type char

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

- comme le caractère concerné: a, Z, fin de ligne....

- comme le code de ce caractère, c'est-à-dire un motif de 8 bits; or à ce dernier on peut toujours faire correspondre un nombre entier (le nombre qui, codé en binaire, fournit le motif en question); par exemple, dans le code ASCII, le caractère A est représenté par le motif binaire 01000101, auquel on peut faire correspondre le nombre 69.

3.3 LES OPÉRATEURS RELATIONNELS

Comme tout langage, C permet de "comparer" des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple : 2*a > b +5

Par contre, C se distingue de la plupart des autres langages sur deux points:

- le résultat de la comparaison est, non pas une valeur "booléenne" (on dit aussi "logique") prenant l'une des deux valeurs vrai ou faux, mais un entier valant:

0 si le résultat de la comparaison est faux,

1 si le résultat de la comparaison est vrai.

Ainsi, la comparaison ci-dessus devient en fait une expression de type entier. Cela signifie qu'elle pourra éventuellement intervenir dans des calculs arithmétiques;

Remarque importante: comparaisons de caractères

Compte tenu des règles de conversion, une comparaison peut porter sur deux caractères. Bien entendu, la comparaison d'égalité ne pose pas de problème particulier; par exemple (cl et c2 étant de type char):

cl == c2 sera vraie si cl et c2 ont la même valeur, c'est-à-dire si cl et c2 contiennent des caractères de même code, donc si cl et c2 contiennent le même caractère,

cl == 'e' sera vraie si le code de cl est égal au code de 'e', donc si cl contient le caractère e.

L'ordre alphabétique est respecté pour les minuscules d'une part, pour les majuscules d'autre part; on a toujours 'a' < 'c', 'C' < 'S'...

- les chiffres sont rangés dans leur ordre naturel; on a toujours '2' < '5'...

3.4 LES OPÉRATEURS LOGIQUES

C dispose de trois opérateurs logiques classiques: et (noté &&), ou (noté | |) et non (noté !). Par exemple:
(a<b) && (c<d) prend la valeur 1 (vrai) si les deux expressions a<b et c<d sont toutes deux vraies (de valeur 1), la valeur 0 (faux) dans le cas contraire.

Il est important de constater que, ne disposant pas de type logique, C se contente de représenter vrai par 1 et faux par 0. C'est pourquoi ces opérateurs produisent un résultat numérique (de type int).

De plus, on pourrait s'attendre à ce que les opérandes de ces opérateurs ne puissent être que des expressions prenant soit la valeur 0, soit la valeur 1. En fait, ces opérateurs acceptent n'importe quel opérande numérique, y compris les types flottants, avec les règles de conversion implicite déjà rencontrées. Leur signification reste celle évoquée ci-dessus, à condition de considérer que:

- **0 correspond à faux,**

- toute valeur non nulle (et donc pas seulement la valeur 1) correspond à vrai.

3.5 L'OPÉRATEUR D'AFFECTATION ORDINAIRE

Nous avons déjà eu l'occasion de remarquer que : i = 5 était une expression qui réalisait une action : l'affectation de la valeur 5 à i. Cet opérateur d'affectation (=) peut faire intervenir d'autres expressions comme dans : c=b + 3
La faible priorité de cet opérateur = (elle est inférieure à celle de tous les opérateurs arithmétiques et de comparaison) fait qu'il y a d'abord évaluation de l'expression b + 3. La valeur ainsi obtenue est ensuite affectée à c.
En revanche, il n'est pas possible de faire apparaître une expression comme premier opérande de cet opérateur =. Ainsi, l'expression suivante n'aurait pas de sens:c +5 =x

3.6 LES OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

3.6.1 Leur rôle

Dans des programmes écrits dans un langage autre que C, on rencontre souvent des expressions (ou des instructions) telles que : i = i + 1, n = n – 1 qui "incrémentent" ou qui "décrémentent" de 1 la valeur d'une "variable" .

En C, ces actions peuvent être réalisées par des opérateurs "unaires" portant sur cette "lvalue". Ainsi, l'expression: ++i a pour effet d'incrémenter de 1 la valeur de i, et sa valeur est celle de i après incrémentation.

Là encore, comme pour l'affectation, nous avons affaire à une expression qui non seulement possède une valeur, mais qui, de surcroît, réalise une action (incrémentation de i).

Il est important de voir que la valeur de cette expression est celle de i après incrémentation. Ainsi, si la valeur de i est 5, l'expression : n = ++i-5 affectera à i la valeur 6 et à n la valeur 1.
En revanche, lorsque cet opérateur est placé après la "lvalue" sur laquelle il porte, la valeur de l'expression correspondante est celle de la variable avant incrémentation. Ainsi, si i vaut 5, l'expression: n = i++ - 5 affectera à i la valeur 6 et à n la valeur 0 (car ici la valeur de l'expression i++ est 5).
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.

Bien entendu, lorsque seul importe l'effet d'incrémentation d'une "lvalue", cet opérateur peut être indifféremment placé avant ou après. Ainsi, ces deux instructions (ici, il s'agit bien d'instructions car les expressions sont terminées par un point-virgule—leur valeur se trouve donc inutilisée) sont équivalentes:i++; ++i;

3.6.2 Leurs priorités

Les priorités élevées de ces opérateurs unaires (voir tableau en fin de chapitre) permettent d'écrire des expressions assez compliquées sans qu'il soit nécessaire d'employer des parenthèses pour isoler la "Ivalue" sur laquelle ils portent. Ainsi, l'expression suivante a un sens:

3 * i++ *j-- + k++
(si * avait été plus prioritaire que la post incrémentation, ce dernier aurait été appliqué à l'expression 3*i qui n'est pas une "lvalue"; l'expression n'aurait alors pas eu de sens).

Remarque:

Il est toujours possible (mais non obligatoire) de placer un ou plusieurs espaces entre un opérateur et les opérandes sur lesquels il porte. Nous utilisons souvent cette latitude pour accroître la lisibilité de nos instructions. Cependant, dans le cas des opérateurs d'incrémentation, nous avons plutôt tendance à ne pas le faire, cela pour mieux "rapprocher" l'opérateur de la "lvalue" sur laquelle il porte.

3.6.3 Leur intérêt

Ces opérateurs allègent l'écriture de certaines expressions et offrent surtout le grand avantage d'éviter la "redondance" qui est de mise dans la plupart des autres langages. En effet, dans une notation telle que:i++ on ne "cite" qu'une seule fois la "lvalue" concernée alors qu'on est amené à le faire deux fois dans la notation:i=i+ 1.

Les risques d'erreurs de programmation s'en trouvent ainsi quelque peu limités. Bien entendu, cet aspect prendra d'autant plus d'importance que la "lvalue" correspondante sera d'autant plus complexe.

D'une manière générale, nous utiliserons fréquemment ces opérateurs dans la manipulation de tableaux ou de chaînes de caractères. Ainsi, anticipant sur les chapitres suivants, nous pouvons indiquer qu'il sera possible de lire l'ensemble des valeurs d'un tableau nommé t en répétant la seule instruction:

t [i++] = getchar();

Celle-ci réalisera à la fois:
- la lecture d'un caractère au clavier,

- l'affectation de ce caractère à l'élément de rang i du tableau t,

- I'incrémentation de 1 de la valeur de i (qui sera ainsi préparée pour la lecture du prochain élément).

3.7 LES OPÉRATEURS D'AFFECTATION ÉLARGIE

Nous venons de voir comment les opérateurs d'incrémentation permettaient de simplifier l'écriture de certaines affectations. Par exemple:i++ remplaçait avantageusement:i=i+ 1

Mais C dispose d'opérateurs encore plus puissants. Ainsi, vous pourrez remplacer: i = i + k par : i += k ou, mieux encore : 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 et de manipulation de bits. Voici la liste complète de tous ces nouveaux opérateurs nommés "opérateurs d'affectation élargie": += -= *= /= %= |= ...
Ces opérateurs, comme ceux d'incrémentation, permettent de condenser l'écriture de certaines instructions et contribuent à éviter la redondance introduite fréquemment par l'opérateur d'affectation classique.

4. LES ENTRÉES-SORTIES CONVERSATIONELLES

4.1 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.

4.1.1 Les principaux codes de conversion

c char: caractère affiché "en clair" (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 ennotation "décimale" avec six chiffres après le point (par exemple: 1.234500 ou 123.456789)

e double ou float (compte tenu des conversions systématiques float - > double) écrit en notation ‘’exponentielle’’ (mantisse entre 1 inclus et 10 exclu) avec six chiffres après le point décimal, sous la forme x.xxxxxxe+yyy ou x.xxxxxxe-yyy pour les nombres positifs et -x.xxxxxxe+yyy ou -x.xxxxxxe-yyy pour les nombres négatifs

s chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

4.1.2 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.

4.1.3 La macro putchar

L' expression putchar() joue le même rôle que: printf (" %c" , c)

Son exécution est toutefois plus rapide, dans la mesure où elle ne fait pas appel au mécanisme d'analyse de format. Notez qu'en toute rigueur putchar n'est pas une vraie fonction mais une "macro". Ses instructions (écrites en C) seront incorporées à votre programme par la directive:

#include <stdio.h>

Alors que cette directive était facultative pour printf (qui est une fonction), elle devient absolument nécessaire pour putchar. En son absence, l'éditeur de liens serait amené à rechercher une fonction putchar en bibliothèque et, ne la trouvant pas, il vous gratifierait d'un message d'erreur.

4.2 LES POSSIBILITÉS DE LA FONCTION SCANF

Nous avons déjà rencontré quelques exemples d'appels de scanf. Nous y avons notamment vu la nécessité de recourir à l'opérateur & pour désigner l'adresse de la variable (plus généralement de la "lvalue") pour laquelle on souhaite lire une valeur. Vous avez pu remarquer que cette fonction possédait une certaine ressemblance avec printf et qu'en particulier elle faisait, elle aussi, appel à des "codes de format".

Cependant, ces ressemblances masquent également des différences assez importantes au niveau :

- de la signification des codes de format. Certains codes correspondront à des types différents, suivant qu'ils sont employés avec printf ou avec scanf;

- de l'interprétation des caractères du format qui ne font pas partie d'un code de format.

En revanche, compte tenu de la complexité de scanf, nous vous en exposerons les différentes possibilités de façon progressive, à l'aide d'exemples. Notamment, ce n'est qu'à la fin de ce chapitre que vous serez en mesure de connaître toutes les conséquences de données incorrectes.

4.2.1 Les principaux codes de conversion de scanf

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

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 (éventuellement sans point, c'est-à-dire comme un entier) 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.

4.2.2 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.

4.3 Exercice :

Utiliser les instructions printf et scanf pour afficher des nombres quelconques entiers ou réels entrés par l'utilisateur.

5. LES INSTRUCTIONS DE CONTROLES

5.1 L'INSTRUCTION IF

Nous avons déjà rencontré des exemples d'instruction if et nous avons vu que cette dernière pouvait éventuellement faire intervenir un "bloc". Précisons donc tout d'abord ce qu'est un bloc d'une manière générale.

5.1.1 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...

Rappelons qu'il y a en C , comme en Pascal , une sorte de récursivité de la notion d'instruction. Dans la description de la syntaxe des différentes instructions, nous serons souvent amené à mentionner ce terme d'instruction. Comme nous l'avons déjà noté, celui-ci désignera toujours n'importe quelle instruction C : simple, structurée ou un bloc.

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

{} {i= 1 ;}

Le second bloc ne présente aucun intérêt en pratique puisqu'il pourra toujours être remplacé par l'instruction simple qu'il contient.

En revanche, nous verrons que le premier bloc (lequel pourrait a priori être remplacé par... rien) apportera une meilleure lisibilité dans le cas de boucles ayant un "corps" vide.
Notez encore que {; } est un bloc constitué d'une seule instruction "vide", ce qui est "syntaxiquement" correct.

Remarque importante:N'oubliez pas que toute instruction simple est toujours terminée par un point-virgule. Ainsi, ce bloc :{i =5;k=3}est incorrect car il manque un point-virgule à la fin de la seconde instruction qu'il contient.
D'autre part, un bloc joue le même rôle syntaxique qu'une instruction simple (point virgule compris). Evitez donc d'ajouter des points-virgules intempestifs à la suite d'un bloc.

5.1.2 Syntaxe de l'instruction if

Le mot else et l'instruction qu'il introduit sont facultatifs.

Remarque:Notez bien que la syntaxe de cette instruction n'impose en soi aucun point-virgule, si ce n'est ceux qui terminent naturellement les instructions simples qui y figurent.

Un else se rapporte toujours au dernier if rencontré auquel un else n'a pas encore été attribué.

5.2 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.

N.B.: les crochets ([et]) signifient que ce qu'ils renferment est facultatif.

5.3 L’instruction do... while

do   instruction

while (expression);

1) Notez bien, d'une part la présence de parenthèses autour de l'expression qui régit la poursuite de la boucle, d'autre part la présence d'un point-virgule à la fin de cette instruction.

5.4 L’instruction while

while   (expression)

instruction

Commentaires

1) Là encore, notez bien la présence de parenthèses pour délimiter la condition de poursuite. Remarquez que, par contre, la syntaxe n'impose aucun point-virgule de fin (il s'en trouvera naturellement un à la fin de l'instruction qui suit si celle-ci est simple).
2) L'expression utilisée comme condition de poursuite est évaluée avant le premier tour de boucle. Il est donc nécessaire que sa valeur soit définie à ce moment.

3) Lorsque la condition de poursuite est une expression qui fait appel à l'opérateur séquentiel, n'oubliez pas qu'alors toutes les expressions qui la constituent seront évaluées avant le "test de poursuite" de la boucle.

5.5 L'INSTRUCTION FOR

Etudions maintenant la dernière instruction permettant de réaliser des boucles, à savoir I' instruction for.

5.5.1 Exemple d'introduction de l'instruction for

Considérez ce programme:

```
#include <stdio.h>
int main(void)
{       int i   ;
        for (i=1 ; i<=5 ; i++ )
        {       printf ("bonjour ") ;
                printf ("°/.d fois\n", i);
        }
return 0 ;
}
```

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.

5.5.2 Syntaxe de l'instruction FOR

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

5.6 Exercices :

1) Soit le programme suivant :

```
#include <stdio.h>
int main(void)
{       int i,n,som;
        som = 0;
        for(i=0;i<4;i++)
        {       printf(''donnez un entier '');
                scanf('' %d '',&n);
                som+=n;
        }
printf(''somme : %d\n'',som);
return 0 ;
}
```

Ecrire un programme réalisant exactement la même chose, en employant :

a) une instruction while;
b) une instruction do ... while.

2) Calculer la moyenne de notes fournies au clavier. Le nombre de notes n'est pas connu à priori et l'utilisateur peut en fournir autant qu'il le désire. Pour signaler qu'il a terminé on convient de fournir une note négative non comptée dans la moyenne.

3) Afficher un triangle rempli d'étoiles, s'étendant sur un nombre de lignes fourni en donnée et se présentant ainsi :

```
*
**
***
****
```

4) Déterminer si un nombre entier fourni en donnée est premier ou non.

5) Ecrire un programme qui détermine la nième valeur de la suite de Fibonacci définie comme suit :
$u_1 = 1$; $u_2 = 1$; $u_n = u_{n-1} + u_{n-2}$ pour $n > 2$.

6) Ecrire un programme qui affiche la table de multiplication des nombres de 1 à 10, sous la forme suivante :

```
        I       1       2       3       4       5       6       7       8       9       10
-------------------------------------------------------------------------------------------
1       I       1       2       3       4       5       6       7       8       9       10
2       I       2       4       6       8       10      12      14      16      18      20
...
10      I       10      20      30      40      50      60      70      80      90      100
```

6. LA PROGRAMMATION MODULAIRE ET LES FONCTIONS

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

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

6.1 LA FONCTION: LA SEULE SORTE DE MODULE EXISTANT EN C

En C, il n'existe qu'une seule sorte de module, nommé fonction. Ce terme, quelque peu abusif, pourrait laisser croire que les modules du C sont moins généraux que ceux des autres langages. Or il n'en est rien, bien au contraire ! Certes, la fonction pourra y être utilisée comme dans d'autres langages, c'est-à-dire recevoir des arguments et fournir un résultat scalaire qu'on utilisera dans une expression, comme, par exemple, dans :

y = sqrt(x)+3;

Mais, en C, la fonction pourra prendre des aspects différents, pouvant complètement dénaturer l'idée qu'on se fait d'une fonction. Par exemple:

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

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

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

```
/***** 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 ;
        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) ;
```

```
        /* 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 (fdouble 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 me 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"...):

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

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

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

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 (mais comme jusqu'ici nous avions affaire à un programme constitué d'une seule fonction, cette distinction n'était pas utile). Un peu plus loin, nous examinerons plus en détail cette notion de variable locale et celle de "portée" qui s'y attache.

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

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

En définitive, on peut dire que fexple est une fonction telle que fexple (x, b, c) fournisse la valeur de l'expression $x^2 + bx + c$.
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. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration.

Quant à l'utilisation de notre fonction fexple au sein de la fonction main, elle est classique et comparable à celle d'une fonction prédéfinie telle que scanf ou sqrt. Ici, nous nous sommes contenté d'appeler notre fonction à deux reprises avec des arguments différents.

6.3 QUELQUES RÈGLES

6.3.1 Arguments muets et arguments effectifs

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

6.3.2 L'instruction return

Voici quelques règles générales concernant cette instruction.

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

```
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)
}
```

Notez bien que non seulement l'instruction return définit la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée. Nous verrons qu'une fonction peut ne fournir aucune valeur: aucune instruction return ne figurera alors dans sa définition. Dans ce cas (absence d'instruction return), le retour est mis en place automatiquement par le compilateur à la "fin" de la fonction.
- Si le type de l'expression figurant dans return est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion.

Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. C'est d'ailleurs ce que nous avons fait fréquemment avec printf ou scanf. Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de calculer un résultat. En revanche, il est interdit d'utiliser la valeur d'une fonction ne fournissant pas de résultat (si certains compilateurs l'acceptent, vous obtiendrez, lors de l'exécution, une valeur aléatoire!).

6.3.3 Cas des fonctions sans valeur de retour ou sans arguments

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

```
void sansval (int n)
```

et voici quelle serait sa déclaration:

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

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

Quand une fonction ne reçoit aucun argument, on place le mot clé void (le même que précédemment, mais avec une signification différente !) à la place de la liste d'arguments. Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type float (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !):

```
double tirage (void)
```

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

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

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

```
void message (void)
```

et sa déclaration sera:

```
void message (void);
```

6.4 EN C, LES ARGUMENTS SONT TRANSMIS PAR VALEUR

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

```
#include <stdio.h>
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 echange : 10 20 fin echange : 20 10 après appel : 10 20

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

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

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

Or, il ne faut pas oublier qu'en C tous les "modules" doivent être écrits sous forme de fonction. Autrement dit, ce simple problème d'échange des valeurs de deux variables doit pouvoir se résoudre à l'aide d'une fonction.

Nous verrons que ce problème possède plusieurs solutions, à savoir:

- Transmettre en argument la "valeur" de l'adresse" d'une variable. La fonction pourra éventuellement agir sur le "contenu" de cette adresse. C'est précisément ce que nous faisons lorsque nous utilisons la fonction scanf. Nous examinerons cette technique en détail dans le chapitre consacré aux "pointeurs".

- Utiliser des "variables globales", comme nous le verrons dans le prochain paragraphe; cette deuxième solution devra toutefois être réservée à des cas exceptionnels, compte tenu des risques qu'elle présente ("effets de bords").

Remarque:

C'est bien parce que la transmission des arguments se fait "par valeur" que les arguments effectifs peuvent prendre la forme d'une expression quelconque. Dans les langages où le seul mode de transmission est celui "par adresse", les arguments effectifs ne peuvent être que l'équivalent d'une lvalue.

6.5 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 comme en Pascal, 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.

6.6 LES VARIABLES LOCALES

Les variables locales ne sont connues qu'à l'intérieur de la fonction où elles sont déclarées. Il est toutefois possible de demander d'attribuer un emplacement permanent à une varaible locale et qu'ainsi sa valuer se conserve d'un appel au suivant. Il suffit pour cela de la déclarer à l'aide du mot clé STATIC.

6.7 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);
}
```

6.8 Exercices

1) Ecrire:

- une fonction, nommée f1, se contentant d'afficher "bonjour" (elle ne possédera aucun argument ni valeur de retour),

- une fonction, nommée f2 qui affiche "bonjour" un nombre de fois égal à la valeur reçue en argument (int) et qui ne renvoie aucune valeur,

- une fonction, nommée f3 qui fait la même chose que f2 mais qui, de plus, renvoie la valeur (int) 0.

Ecrire un petit programme appelant successivement chacune de ces trois fonctions, après les avoir convenablement déclarées sous forme d'un prototype.

2) Qu'affiche le programme suivant :
```
int n=5;
int main(void)
{
```

```
        void fct (int p);
        int n=3;
        fct(n);
        return 0 ;
}
void fct(int p)
{
        printf('' %d %d '',n,p);
}
```

3) Ecrire une fonction qui se contente de comptabiliser le nombre de fois où elle a été appelée en affichant seulement un message "de temps en temps", à savoir:

- au premier appel: *** appel 1 fois ***
- au dixième appel: *** appel 10 fois ***
- au centième appel: *** appel 100 fois ***
- et ainsi de suite pour le millième, le dix millième appel...

On supposera que le nombre maximal d'appels ne peut dépasser la capacité d'un long.

4) Ecrire une fonction premier permettant de déterminer si un nombre entier donné est premier.

5) Ecrire un programme donnant la liste des nombres premiers de 1 à N.

7. LES TABLEAUX ET LES POINTEURS

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, comme certains langages tels que Pascal, 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.

7.1 LES TABLEAUX A UN INDICE

7.1.1 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[l9].

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.

7.1.2 Quelques règles

a) 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'incrémentation, comme dans:

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

En revanche, il n'est pas possible, si tl et t2 sont des tableaux d'entiers, d'écrire tl = t2; en fait, le langage C n'offre aucune possibilité d'affectations globales de tableaux, comme c'était le cas, par exemple, en Pascal.

b) 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).

c) 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).

d) 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.

7.1.3 LES TABLEAUX À PLUSIEURS INDICES

7.1.4 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 x 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations:

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

Notez bien que, 1à encore, la notation désignant un élément d'un tel tableau est une lvalue. Il n'en ira toutefois pas de même de notations telles que t[3] ou t[j] bien que, comme nous le verrons un peu plus tard, de telles notations aient un sens en C.

Aucune limitation ne pèse sur le nombre d'indices que peut comporter un tableau. Seules les limitations de taille mémoire liées à un environnement donné risquent de se faire sentir.

7.2 INITIALISATION DES TABLEAUX

Il est possible, comme on le fait pour une variable scalaire, d'initialiser (partiellement ou totalement) un tableau lors de sa déclaration. Cette fois, cependant, les valeurs fournies devront obligatoirement être des expressions constantes, et cela quelle que soit la classe d'allocation du tableau concerné (alors que les variables scalaires automatiques pouvaient être initialisées avec des expressions quelconques).

Voici quelques exemples vous montrant comment initialiser un tableau.

7.2.1 NOTION DE POINTEUR - LES OPÉRATEURS * ET &

7.2.2 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:

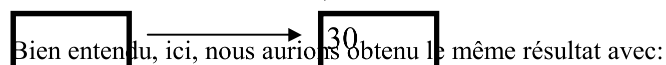
```
ad = &n;
```

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.

L'instruction suivante:

```
*ad = 30;
```

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



Bien entendu, ici, nous aurions obtenu le même résultat avec:

```
n = 30;
```

7.2.3 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++

(nous verrons plus loin que, sans les parenthèses, cette expression aurait une signification différente).

Remarques :

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).

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. Cette remarque prendra encore plus d'acuité lorsque les "objets pointés" seront des chaînes ou des tableaux.

7.2.4 Incrémentation de pointeurs

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

En effet, ad est censée contenir l'adresse d'un entier et, pour C, l'expression ci-dessus représente l'adresse de l'entier suivant. Certes, dans notre exemple, cela n'a guère d'intérêt car nous ne savons pas avec certitude ce qui se trouve à cet endroit. Mais nous verrons que cela s'avérera fort utile dans le traitement de tableaux ou de chaînes.

Notez bien qu'il ne faut pas confondre un pointeur avec un nombre entier. En effet, l'expression ci-dessus ne représente pas l'adresse de ad augmentée de un (octet). Plus précisément, la différence entre ad+l et ad est ici de sizeof(int) octets. Si ad avait été déclarée par : double * ad ; cette différence serait de sizeof(double) octets.
De manière comparable, l'expression :
ad++
incrémente l'adresse contenue dans ad de manière qu'elle désigne l'objet suivant.

Notez bien que des expressions telles que ad+1 ou ad++ sont, en général, valides, quelle que soit l'information se trouvant réellement à l'emplacement correspondant. D'autre part, il est possible d'incrémenter ou de décrémenter un pointeur de n'importe quelle quantité entière.

Remarque:

Il existera une exception à ces possibilités, à savoir le cas des pointeurs sur des fonctions, dont nous parlerons plus loin (vous pouvez dès maintenant comprendre qu'incrémenter un pointeur d'une quantité correspondant à la taille d'une fonction n'a pas de sens en soi !).

7.3 COMMENT SIMULER UNE TRANSMISSION PAR ADRESSE AVEC UN POINTEUR

Nous avons vu que le mode de transmission par valeur semblait interdire à une fonction de modifier la valeur de ses arguments effectifs et nous avions mentionné que les pointeurs fourniraient une solution à ce problème.

Nous sommes maintenant en mesure d'écrire une fonction effectuant la permutation des valeurs de deux variables. Voici un programme qui réalise cette opération avec des valeurs entières :

```
#include <stdio.h>
int main(void)

{       void echange (int * ad1, int * ad2) ;
        int a=10, b=20 ;
        printf ("avant appel %d %d\n", a, b) ;
        echange (&a, &b) ;
        printf ("après appel %d %d", a, b) ;         avant appel 10 20
}                                                    après appel 20 10
void echange (int * ad1, int * ad2)
{
        int x ;
        x = * ad1 ;
        * ad1 = * ad2 ;
        * ad2 = x ;
}
```

Les arguments effectifs de l'appel de echange sont, cette fois, les adresses des variables n et p (et non plus leurs valeurs). Notez bien que la transmission se fait toujours par valeur, à savoir que l'on transmet à la fonction echange les valeurs des expressions &n et &p.

Voyez comme, dans echange, nous avons indiqué, comme arguments muets, deux variables pointeurs destinées à recevoir ces adresses. D'autre part, remarquez bien qu'il n'aurait pas. fallu se contenter d'échanger simplement les valeurs de ces arguments en écrivant (par analogie avec la fonction echange du chapitre précédent):

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

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

7.4 UN NOM DE TABLEAU EST UN POINTEUR CONSTANT

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

7.4.1 Cas des tableaux à un indice

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

t + 1   &t[1]
t + i   &t[i]

t[i]   * (t+i)

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

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

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

Dans la seconde façon, nous avons dû recopier la "valeur" représentée par t dans un pointeur nommé p. En effet, il ne faut pas perdre de vue que le symbole t représente une adresse constante (t est une constante de type pointeur sur des entiers). Autrement dit, une expression telle que t++ aurait été invalide, au même titre que, 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 (encore faudra-t-il, toutefois, avoir alloué l'"espace mémoire" nécessaire).

7.4.2 Cas des tableaux à plusieurs indices

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

A simple titre indicatif, nous vous présentons ici les règles employées par C, en nous limitant au cas de tableaux à deux indices.

Lorsque le compilateur rencontre une déclaration telle que: int t[3] [4];

il considère en fait que t désigne un tableau de 3 éléments, chacun de ces éléments étant lui même un tableau de 4 entiers. Autrement dit, si t représente bien l'adresse de début de notre tableau t, il n'est plus de type int * (comme c'était le cas pour un tableau à un indice) mais d'un type "pointeur sur des blocs de 4 entiers", type qui devrait se noter théoriquement4:

int [4] *

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

D'autre part, les notations telles que t[0], t[1] ou t[i] ont un sens. Par exemple, t[0] représente l'adresse de début du premier bloc (de 4 entiers) de t, t[l], celle du second bloc... Cette fois, il s'agit bien de pointeurs de type int *.

Autrement, dit les notations suivantes sont totalement équivalentes (elles correspondent à la même adresse et elles sont de même type):

t[0]   &t[0][0]
t[l]   &t[11[0]

7.5 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.

7.5.1 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[])

b) 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) ;
        .....
}
```

7.5.2 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])
{ 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.

7.6 Exercices

1) Ecrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit:

a) en utilisant uniquement le "formalisme tableau",
b) en utilisant le "formalisme pointeur", chaque fois que cela est possible.

2) Ecrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. Il faudra donc prévoir 4 arguments: le tableau, sa dimension, le maximum et le minimum.

Ecrire un petit programme d'essai.

3) Ecrire une fonction permettant de trier par ordre croissant les valeurs entières d'un tableau de taille quelconque (transmise en argument). Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.

4) Ecrire une fonction calculant la somme de deux matrices dont les éléments sont de type double. Les adresses des trois matrices et leurs dimensions (communes) seront transmises en argument.

8. LES CHAINES DE CARACTÈRES

8.1 REPRÉSENTATiON DES CHAINES

8.1.1 La convention adoptée

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

8.1.2 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; on retrouve là le même phénomène que pour les tableaux.

8.1.3 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')

8.1.4 Initialisation de tableaux de pointeurs sur des chaînes

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

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

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

8.2 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 cher %s %s qui habitez à ", prenom, nom) ;
        puts (ville) ;
        return 0 ;
}
```

quelle est votre ville : Paris
donnez votre nom et votre prénom : Dupont Yves
bonjour cher Yves Dupont qui habitez à Paris

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, prenom 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.

8.3 GÉNÉRALITÉS SUR LES FONCTIONS PORTANT SUR DES CHAINES

C dispose de nombreuses fonctions de manipulation de chaînes. Avant d'en voir les principales (les autres étant, de toute façon, présentées dans l'annexe A), 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 chaine 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 apparait le caractère mentionné.

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

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

8.4 Exercices

1) Ecrire un programme déterminant le nombre de lettres e (minuscules) présentes dans un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères) fourni au clavier.

2) Ecrire un programme qui supprime toutes les lettres e (minuscules) d'un texte de moins d'une ligne (supposée ne pas dépasser 132 caractères) fourni au clavier. Le texte ainsi modifié sera créé, en mémoire, à la place de l'ancien.

3) Ecrire un programme qui lit au clavier un mot (d'au plus 30 caractères) et qui l'affiche "à l'envers " .

4) Ecrire un programme qui lit un verbe du premier groupe et qui en affiche la conjugaison au présent de l'indicatif, sous la forme :

je chante
tu chantes
il chante
nous chantons
vous chantez
ils chantent

Le programme devra vérifier que le mot fourni se termine bien par "er". On supposera qu'il ne peut comporter plus de 26 lettres et qu'il s'agit d'un verbe régulier. Autrement dit, on admettra que l'utilisateur ne fournira pas un verbe tel que "manger" (le programme afficherait alors : "nous mangons" !).

9. LES STRUCTURES

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.

9.1 DÉCLARATION D'UNE STRUCTURE

Voyez tout d'abord cette déclaration :

```
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 artl "de type enreg" destiné à contenir deux entiers et un double.

De manière semblable : struct enreg art1, art2 ;
réserverait deux emplacements artl et art2 du type enreg.

9.2 UTILISATION D'UNE STRUCTURE

En C, comme en Pascal, 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.

9.2.1 Utilisation des champs d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur "point" (.) suivi du nom de champ 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 artl et art2 déclarées de ce type.

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

printf ("%e", artl.prix) ; affiche, suivant le code format %e, la valeur du champ prix de la structure artl.

scanf ("%e", &art2.prix) ;lit, suivant le code format %e, une valeur qui sera affectée au champ prix de la structure
art2. Notez bien la présence de l'opérateur &.

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

Remarque : la priorité de l'opérateur "." est très élevée, de sorte qu'aucune des expressions ci-dessus ne nécessite de parenthèses.

9.2.2 Utilisation globale d'une structure

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

Une telle affectation globale remplace avantageusement :

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

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

L'opérateur d'affectation et, comme nous le verrons un peu plus loin, l'opérateur d'adresse & sont les seuls opérateurs s'appliquant à une structure (de manière globale).

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.

9.2.3 Initialisations de structures

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

- En l'absence d'initialisation explicite, les structures de classe "statique" sont, par défaut, initialisées à zéro; celles possédant la classe "automatique" ne sont pas initialisées par défaut (elles contiendront donc des valeurs aléatoires).

-Il est possible d'initialiser explicitement une structure lors de sa déclaration. On ne peut toutefois employer que des constantes ou des expressions constantes et cela aussi bien pour les structures statiques que pour les structures automatiques (alors que, pour les variables scalaires automatiques, il était possible d'employer une expression quelconque):

Voici un exemple d'initialisation de notre structure artl, 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.

9.3 POUR SIMPLIFIER LA DÉCLARATION DE TYPES DÉFINIR DES SYNONYMES AVEC TYPEDEF

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

9.3.1 Exemples d'utilisation de typedef

La déclaration : `typedef int entier;`

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

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

9.3.2 Application aux structures

En faisant usage de typedef, les déclarations des structures artl 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 ;
```

Par la suite, nous ne ferons appel qu'occasionnellement à typedef, afin de ne pas vous enfermer dans un style de notations que vous ne retrouverez pas nécessairement dans les programmes que vous serez amené à utiliser.

9.4 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.

9.4.1 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. Il s'agit d'un élément de type float. Notez que, malgré les priorités identiques des opérateurs . et [], leur associativité de gauche à droite évite l'emploi de parenthèses.
De même:
employe.nom[0]
représente le premier caractère du champ nom de la structure employe.

Par ailleurs:

&courant.heures[4]

représente l'adresse du cinquième élément du tableau heures de la structure courant. Notez que, la priorité de l'opérateur & étant inférieure à celle des deux autres, les parenthèses ne sont, 1à encore, pas nécessaires.

Enfin:

courant.nom

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

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

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

9.4.2 Tableaux de structures

Voyez ces déclarations :

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

attention : rajouter extern _turboFloat après les include et (void)_turboFloat n'importe ou dans le main() (Bug Borland C++)

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] 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.

9.5 À PROPOS DE LA PORTÉE DU MODÈLE DE STRUCTURE

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

- si elle se situe au sein d'une fonction (y compris, la "fonction main"), elle n'est accessible que depuis cette fonction,

- si elle se situe en dehors d'une fonction, elle est accessible de toute la partie du fichier source qui suit sa déclaration; elle peut ainsi être utilisée par plusieurs fonctions.

Il est néanmoins toujours possible de placer un certain nombre de déclarations de modèles de structures dans un fichier séparé que l'on incorpore par #include à tous les fichiers source où l'on en a besoin. Cette méthode évite la duplication des déclarations identiques avec les risques d'erreurs qui lui sont inhérents.

9.6 TRANSMISSION D'UNE STRUCTURE EN ARGUMENT D'UNE FONCTION

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

9.6.1 Transmission de la valeur d'une structure

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 appet 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.

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

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

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

9.7 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.

10. LES FICHIERS

Nous avons déjà eu l'occasion d'étudier les "entrées-sorties conversationnelles ", c'est-à-dire les fonctions permettant d'échanger des informations entre le programme et l'utilisateur. Nous vous proposons ici d'étudier les fonctions permettant au programme d'échanger des informations avec des "fichiers". A priori, ce terme de fichier désigne plutôt un ensemble d'informations situé sur une "mémoire de masse" telle que le disque ou la disquette. Nous verrons toutefois qu'en C, comme d'ailleurs dans d'autres langages, tous les périphériques, qu ' ils soient d'archivage (disque, disquette , . . .) ou de communication (clavier, écran, imprimante,...), peuvent être considérés comme des fichiers. Ainsi, en définitive, les entrées-sorties conversationnelles apparaîtront comme un cas particulier de la gestion de fichiers.

Rappelons que l'on distingue traditionnellement deux techniques de gestion de fichiers :

- 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.

10.1 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) ;
        retunr 0 ;
}
```

Nous avons déclaré un tableau de caractères nomfich destiné à contenir, sous forme d'une chaine, 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 l),

- 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

Remarques:

1) On emploie souvent le terme flux (en anglais stream) pour désigner un pointeur sur une structure de type FILE. Ici, par exemple, sortie est un flux que la fonction fopen aura associé à un certain fichier. D'une manière générale, par souci de simplification, lorsque aucune ambiguïté ne sera possible, nous utiliserons souvent le mot fichier à la place de flux .

2) 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.

3) 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).

10.2 LISTE SÉQUENTIELLE D'UN FICHIER

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

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

```
        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.

10.3 L'ACCÈS DIRECT

Les fonctions fread et fwrite lisent ou écrivent un certain nombre d'octets dans un fichier, à partir d'une "position courante". Cette dernière n'est rien d'autre qu'un "pointeur" dans le fichier, c'est-à-dire un nombre précisant le rang du prochain octet à lire ou à écrire. 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".

10.3.1 Accès direct en lecture sur un fichier 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 2.1.

```
#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.

10.3.2 Les possibilités de l'accès 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) ;
```

```
        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.

10.3.3 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 parait 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.

10.4 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.

10.5 LES DIFFÉRENTES POSSIBILITÉS D'OUVERTURE D'UN FICHIER

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

r : lecture seulement; le fichier doit exister.

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

a: écriture enfin 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" .

10.6 LES FICHIERS PRÉDÉFINIS

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

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

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

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

On trouve parfois également:

stdaux: unité auxiliaire

stdprt: imprimante

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

TRUC < DONNEES > RESULTATS

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

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

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

Dans le paragraphe 2.3 du chapitre consacré aux chaînes de caractères, nous vous avons montré comment régler les problèmes posés par scanf, en faisant appel à l'association des deux fonctions gets et sscanf. Pour ce faire, nous avions dû toutefois supposer que les "lignes" lues par gets ne dépasseraient pas une certaine longueur. Cette hypothèse est déjà restrictive dans le cas d'informations provenant du clavier: même si cela peut paraître naturel à la plupart des utilisateurs de ne pas dépasser, par exemple, une largeur d'écran, le risque existe d'en voir certains entrer une ligne trop longue qui "plantera" le programme. Cette même hypothèse devient franchement intolérable dans le cas de lecture dans un fichier (sur lequel peut avoir été redirigée l'entrée standard !).

En fait, il est très simple de régler définitivement ce problème. Il suffit d'employer, à la place de (revoyez l'exemple du paragraphe 2.3 du chapitre consacré aux chaînes):

gets (ligne);

une instruction telle que (LG désignant le nombre maximal de caractères acceptés):

fgets (ligne, LG, stdin)

10.7 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).

5) Ecrire un programme permettent de trier par ordre alphabétique du nom, les informations du fichier crée à l’exercice 2.

6) Ecrire un programme regroupant tous ces « sous programme » avec possibilité de menu de choix pour l’utilisateur.
Pour tous ces exercices utiliser une structure.

11. LA GESTION DYNAMIQUE DE LA MEMOIRE

11.1 LES OUTILS DE BASE DE LA GESTION DYNAMIQUE: MALLOC ET FREE

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

11.1.1 La fonction malloc

a) Premier exemple

Considérez ces intructions:

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

L'appel : malloc (50) alloue un emplacement de 50 octets dans le ‘’tas’’ et en fournit l'adresse en retour. Ici, cette dernière est placée dans le pointeur adr.

L'instruction for qui vient à la suite n'est donnée qu'à titre d'exemple d'utilisation de la zone ainsi créée.

b) Second exemple

```
long * adr;
.....
adr = malloc (100 * sizeof(long));

for (i=0; i<100; i++) *(adr+i) = 1;
```

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

adr + i

correspond à l'adresse contenue dans adr, augmentée de sizeof(long) fois la valeur de i (puisque adr pointe sur des objets de longueur sizeof(long)).

c) D'une manière générale

Le prototype de malloc (qui figure à la fois dans stdlib.h et dans alloc.h) est précisément:

void * malloc (size t taille)   (sldlib.h) (alloc.h)

Il montre tout d'abord que le résultat fourni par malloc est un "pointeur générique" (revoyez éventuellement le paragraphe correspondant du chapitre relatif aux pointeurs). Il pourra donc être converti implicitement en un pointeur de n' importe quel type; ainsi, dans nos précédents exemples, il a pu être converti en char * ou en long *. Une telle conversion peut apparaître relativement fictive, dans la mesure où l'adresse correspondante n'est pas modifiée par la conversion. Elle n'en a pas moins une grande importance puisque, comme nous l'avons déjà mentionné à diverses reprises, la connaissance du type d'un pointeur intervient dans les calculs arithmétiques portant sur ce pointeur.

D'autre part, nous constatons que l'unique argument de malloc est d'un type a priori inattendu (nous aurions pu penser à int ou long). En fait, size_t est un nom de type prédéfini (par typedef). Le type exact lui correspondant

dépend de l'implémentation. Cela signifie donc que la taille maximale des objets que l'on peut s'allouer par malloc dépend de l'implémentation .

Enfin, il faut savoir que malloc fournit un pointeur nul (NULL) dans le cas où l'allocation mémoire a échoué. Bien entendu, dans un programme "opérationnel", il sera nécessaire de s'assurer qu'aucun problème de cette sorte n'apparaît.

11.1.2 La fonction free

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

Voici un exemple de programme, exécuté ici dans un environnement DOS. Il vous montre comment malloc peut profiter d'un espace préalablement libéré sur le tas.

```
#include <stdio.h>
#include <alloc.h>
int main(void)
{
        char * adr1, * adr2, * adr3 ;

        adr1 = malloc (100) ;
        printf ("allocation de 100 octets en %p\n", adr1) ;
        adr2 = malloc (50) ;
        printf ("allocation de 50 octets en %p\n", adr2) ;
        free (adr1) ;
        printf ("libération de 100 octets en %p\n", adr1) ;
        adr3 = malloc (40) ;
        printf ("allocation de 40 octets en %p\n", adr3) ;
        return 0 ;
}
```

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

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

Remarques:

1) Dans cet exemple, vous pouvez constater que l'allocation d'une zone de 100 octets nécessite en fait un peu plus de place mémoire (exactement 104 octets). La différence (4 octets) correspond à des "octets de service" dans lesquels le système place les informations nécessaires à sa gestion dynamique de la mémoire.

2) La norme prévoit que malloc alloue convenablement de l'espace, en tenant compte d'éventuelles contraintes d'alignement de l'objet concerné. Or, en toute rigueur, malloc n'a pas d'information précise sur le type de l'objet (elle n'en a que la longueur !). Dans ces conditions, le respect de la norme peut prendre des allures différentes suivant l'implémentation :

- alignement basé sur la taille demandée,

- alignement systématique tenant compte de la contrainte d'alignement la plus forte.

Dans tous les cas, bien sûr, aucun risque n'existe. Simplement, la taille mémoire réellement utilisée pourra, pour un type donné, différer d'une implémention à une autre (notez qu'en toute rigueur c'est déjà ce qui se produit avec les "octets de service" dont le nombre peut varier avec l'implémentation).

11.2 Exercice : utiliser ces nouveautés pour compléter votre programme sur les fichiers.

1. GÉNÉRALITÉS SUR LE LANGAGE C 1

1.1 PRÉSENTATION PAR L'EXEMPLE DE QUELQUES INSTRUCTIONS DU LANGAGE C 1
1.1.1 Un exemple de programme en langage C 1
1.1.2 Structure d'un programme en langage C 1
1.1.3 Déclarations 1
1.1.4 Pour écrire des informations: la fonction printf 1
1.1.5 La fonction return 2
1.1.6 Les directives à destination du préprocesseur 2

1.2 QUELQUES RÈGLES D'ÉCRITURE 2
1.2.1 Les identificateurs 2
1.2.2 Les mots clés 2
1.2.3 Les séparateurs 2
1.2.4 Les commentaires 3

1.3 CRÉATION D'UN PROGRAMME EN LANGAGE C 3
1.3.1 L'édition du programme 3
1.3.2 La compilation 3
1.3.3 L'édition de liens 3

2. LES TYPES DE BASES DU LANGAGE C 4

2.1 LA NOTION DE TYPE 4

2.2 LES TYPES ENTIERS 4
2.2.1 Leur représentation en mémoire 4
2.2.2 Les différents types d'entiers 4
2.2.3 Notation des constantes entières 5

2.3 LES TYPES FLOTTANTS 5
2.3.1 Les différents types et leur représentation en mémoire 5
2.3.2 Notation des constantes flottantes 5

2.4 LES TYPES CARACTÈRES 6
2.4.1 La notion de caractère en langage C 6
2.4.2 Notation des constantes caractères 6

2.5 INITIALISATION ET CONSTANTES 6

3. LES OPÉRATEURS ET LES EXPRESSIONS EN LANGAGE C 7

3.1 L'ORIGINALITÉ DES NOTIONS D'OPÉRATEUR ET D'EXPRESSION EN LANGAGE C 7

3.2 LES OPÉRATEURS ARITHMÉTIQUES EN C 7
3.2.1 Présentation des opérateurs 7
3.2.2 Les priorités relatives des opérateurs 7
3.2.3 Le cas du type char 7

3.3 LES OPÉRATEURS RELATIONNELS 7

3.4 LES OPÉRATEURS LOGIQUES 8

3.5 L'OPÉRATEUR D'AFFECTATION ORDINAIRE 8

3.6 LES OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION 8
3.6.1 Leur rôle 8
3.6.2 Leurs priorités 9
3.6.3 Leur intérêt 9

3.7 LES OPÉRATEURS D'AFFECTATION ÉLARGIE 10

4. LES ENTRÉES-SORTIES CONVERSATIONELLES 11

4.1 LES POSSIBILITÉS DE LA FONCTION PRINTF 11
4.1.1 Les principaux codes de conversion 11
4.1.2 Action sur le gabarit d'affichage 11
4.1.3 La macro putchar 11

4.2 LES POSSIBILITÉS DE LA FONCTION SCANF 12
4.2.1 Les principaux codes de conversion de scanf 12
4.2.2 La macro getchar 12

4.3 Exercice : 13

5. LES INSTRUCTIONS DE CONTROLES 14

5.1 L'INSTRUCTION IF 14
5.1.1 Blocs d'instructions 14
5.1.2 Syntaxe de l'instruction if 14

5.2 L'instruction switch 14

5.3 L’instruction do... while 15

5.4 L’instruction while 15

5.5 L'INSTRUCTION FOR 15
5.5.1 Exemple d'introduction de l'instruction for 15
5.5.2 Syntaxe de l’instruction FOR 16

5.6 Exercices : 16

6. LA PROGRAMMATION MODULAIRE ET LES FONCTIONS 17

6.1 LA FONCTION: LA SEULE SORTE DE MODULE EXISTANT EN C 17

6.2 EXEMPLE DE DÉFINITION ET D'UTILISATION D'UNE FONCTION EN C 17

6.3 QUELQUES RÈGLES 19
6.3.1 Arguments muets et arguments effectifs 19
6.3.2 L'instruction return 19
6.3.3 Cas des fonctions sans valeur de retour ou sans arguments 19

6.4 EN C, LES ARGUMENTS SONT TRANSMIS PAR VALEUR 20

6.5 LES VARIABLES GLOBALES 21

6.6 LES VARIABLES LOCALES 21

6.7 LES FONCTIONS RECURSIVES 21

6.8 Exercices **21**

7. LES TABLEAUX ET LES POINTEURS 23

7.1 LES TABLEAUX A UN INDICE **23**
7.1.1 Exemple d'utilisation d'un tableau en C 23
7.1.2 Quelques règles 24
7.1.3 LES TABLEAUX À PLUSIEURS INDICES 24
7.1.4 Leur déclaration 24

7.2 INITIALISATION DES TABLEAUX **25**
7.2.1 NOTION DE POINTEUR - LES OPÉRATEURS * ET & 25
7.2.2 Introduction 25
7.2.3 Quelques exemples 25
7.2.4 Incrémentation de pointeurs 26

7.3 COMMENT SIMULER UNE TRANSMISSION PAR ADRESSE AVEC UN POINTEUR **27**

7.4 UN NOM DE TABLEAU EST UN POINTEUR CONSTANT **27**
7.4.1 Cas des tableaux à un indice 27
7.4.2 Cas des tableaux à plusieurs indices 28

7.5 LES TABLEAUX TRANSMIS EN ARGUMENT **29**
7.5.1 Cas des tableaux à un indice 29
7.5.2 Cas des tableaux à plusieurs indices 29

7.6 Exercices **30**

8. LES CHAINES DE CARACTÈRES 31

8.1 REPRÉSENTATiON DES CHAINES **31**
8.1.1 La convention adoptée 31
8.1.2 Cas des chaînes constantes 31
8.1.3 Initialisation de tableaux de caractères 31
8.1.4 Initialisation de tableaux de pointeurs sur des chaînes 32

8.2 POUR LIRE ET ÉCRIRE DES CHAINES **32**

8.3 GÉNÉRALITÉS SUR LES FONCTIONS PORTANT SUR DES CHAINES **33**

8.4 Exercices **34**

9. LES STRUCTURES 35

9.1 DÉCLARATION D'UNE STRUCTURE **35**

9.2 UTILISATION D'UNE STRUCTURE **35**
9.2.1 Utilisation des champs d'une structure 35
9.2.2 Utilisation globale d'une structure 36
9.2.3 Initialisations de structures 36

9.3 POUR SIMPLIFIER LA DÉCLARATION DE TYPES DÉFINIR DES SYNONYMES AVEC TYPEDEF **36**
9.3.1 Exemples d'utilisation de typedef 36
9.3.2 Application aux structures 37

9.4 IMBRICATION DE STRUCTURES **37**

9.4.1 Structure comportant des tableaux 37
9.4.2 Tableaux de structures 38

9.5 À PROPOS DE LA PORTÉE DU MODÈLE DE STRUCTURE 38

9.6 TRANSMISSION D'UNE STRUCTURE EN ARGUMENT D'UNE FONCTION 39
9.6.1 Transmission de la valeur d'une structure 39
9.6.2 Transmission de l'adresse d'une structure: l'opérateur -> 39

9.7 Exercices 40

10. LES FICHIERS 41

10.1 CRÉATION SÉQUENTIELLE D'UN FICHIER 41

10.2 LISTE SÉQUENTIELLE D'UN FICHIER 42

10.3 L'ACCÈS DIRECT 43
10.3.1 Accès direct en lecture sur un fichier existant 44
10.3.2 Les possibilités de l'accès direct 44
10.3.3 En cas d'erreur 45

10.4 LES ENTRÉES-SORTIES FORMATÉES ET LES FICHIERS DE TEXTE 46

10.5 LES DIFFÉRENTES POSSIBILITÉS D'OUVERTURE D'UN FICHIER 47

10.6 LES FICHIERS PRÉDÉFINIS 48

10.7 Exercices 48

11. LA GESTION DYNAMIQUE DE LA MEMOIRE 50

11.1 LES OUTILS DE BASE DE LA GESTION DYNAMIQUE: MALLOC ET FREE 50
11.1.1 La fonction malloc 50
11.1.2 La fonction free 51

11.2 Exercice : utiliser ces nouveautés pour compléter votre programme sur les fichiers. 51