Cours de **P**rogrammation

(Première Année - Licence)
2021 - 2022

LANGAGE C

IAI-TOGO (Lomé TOGO)
BP : 12456 Lomé - TOGO
Tél: (+228) 22 22 12 07 / (+228) 22 21 27 06
Fax (+228) 22 22 12 07
Site web: www.iai-togo.com

Chargés de cours : Sina GNOFAME / Mélouziba DIZEWE

Emails: gnofsin@gmail.com / melouzibadize@gmail.com

SEQUENCE I : GENERALITES SUR LE LANGAGE C

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur deux exemples commentés. Vous y découvrirez (pour l'instant, de façon encore informelle) comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux des structures fondamentales (boucle avec compteur, choix).

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.**

I. PRESENTATION PAR UN EXEMPLE, DE QUELQUES INSTRUCTIONS DU C

I.1. Exemple de programme

Voici un exemple de programme en langage C, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonctionnement.

```
#include<stdio.h>
#include<math.h>
#define NFOIS 5

main()
{
int i;
float x;
float racx;

printf ("Bonjour\n");
printf ("Je vais vous calculer %d racines carrées \n", NFOIS);

for (i=0; i<NFOIS; i++)
 {
   printf("Donnez un nombre : ");
   scanf("%f", &x);

   if(x < 0.0)
      printf("Le nombre %f ne possède pas de racine carrée \n", x);
   else
   {
       racx = sqrt(x);
       printf("Le nombre %f a pour racine carrée : %f\n", x, racx);
   }
 }
```

```
printf ("Travail terminé - Au revoir");
}
```

```
Bonjour
Je vais vous calculer 5 racines carrées
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carrée : 2.000000
Donnez un nombre : 2
Le nombre 2.000000 a pour racine carrée : 1.414214
Donnez un nombre : -3
Le nombre -3.000000 ne possède pas de racine carrée
Donnez un nombre : 5.8
Le nombre 5.800000 a pour racine carrée : 2.408319
Donnez un nombre : 12.58
Le nombre 12.580000 a pour racine carrée : 3.546829
Travail terminé - Au revoir
```

Nous reviendrons un peu plus loin sur le rôle des *trois premières lignes*. Pour l'instant, admettez simplement que le symbole **NFOIS** est équivalent à la valeur **5**.

I.2. Structure d'un programme en Langage C

La ligne :
main()

se nomme un « **en-tête** ». Elle précise que ce qui sera décrit à sa suite est en fait le ***programme principal*** **(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 (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 peut lui-même contenir d'autres blocs (c'est le cas de notre exemple). **En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions.**

I.3. Déclarations

Les trois instructions :
int i;
float x;
float racx;

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.

- Les deux autres déclarations précisent que les variables **x** et **racx** sont de type **float**, 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 dans la plupart des langages actuels, **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.

Remarque :
Une déclaration peut figurer à n'importe quel emplacement, pour peu qu'elle apparaisse avant que la variable correspondante ne soit utilisée.

I.4. Directives à destination du préprocesseur

Les trois premières lignes de notre programme :
#include<stdio.h>
#include<math.h>
#define NFOIS 5

sont en fait un peu particulières. Il s'agit de directives qui seront prises 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.

Les deux premières directives demandent en fait d'introduire (avant compilation) des instructions (en langage C) situées dans les fichiers *stdio.h* et *math.h*. Leur rôle ne sera complètement compréhensible qu'ultérieurement. Pour l'instant, notez que, dès lors que vous faites appel à une fonction prédéfinie, il est nécessaire d'incorporer de tels fichiers, nommés « fichiers en-têtes », qui contiennent des déclarations appropriées concernant cette fonction :
stdio.h pour printf et scanf,
math.h pour sqrt.

Fréquemment, ces déclarations permettront au compilateur d'effectuer des contrôles sur le nombre et le type des arguments que vous mentionnerez dans l'appel de votre fonction.

Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. ***En général, il est indispensable d'incorporer stdio.h.***

La troisième directive demande simplement de remplacer systématiquement, dans toute la suite du programme, le symbole NFOIS par 5. Autrement dit, le programme qui sera réellement compilé comportera ces instructions :

```
printf ("Je vais vous calculer %d racines carrées\n", 5);
for (i=0; i<5; i++)
```

Notez toutefois que le programme proposé est plus facile à adapter lorsque l'on emploie une directive *define*.

II. QUELQUES REGLES D'ECRITURE

Ce paragraphe 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, ainsi que de l'usage des **séparateurs** et des **commentaires**.

II.1. Identificateurs

Les identificateurs servent à désigner les différents « objets » manipulés par le programme : variables, fonctions, etc. (Nous rencontrerons ultérieurement les autres objets manipulés par le langage C : constantes, étiquettes de structure, d'union ou d'énumération, membres de structure ou d'union, types, étiquettes d'instruction GOTO, macros). 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**.

En ce qui concerne les lettres :

- le caractère souligné (_) est considéré comme une lettre. Il peut donc apparaître au début d'un identificateur.
  Voici quelques identificateurs corrects : lg_lig, valeur_5, _total, _89.
- les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Ainsi, en C, les identificateurs **ligne** et **Ligne** désignent deux objets différents.

En ce qui concerne la longueur des identificateurs, la norme ANSI prévoit qu'au moins les 31 premiers caractères soient « significatifs » (autrement dit, deux identificateurs qui diffèrent, par leurs 31 premières lettres désigneront deux objets différents).

II.2. 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 de quelques-uns :

int	if	continue	struct	volatile
short	else	return	switch	extern
char	for	default	while	register
float	do	sizeof	static	enum
double	case	void	typedef	

long	goto	signed	auto	
const	break	unsigned	union	

II.3. 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 (tel que : , =; * ( ) [ ] { }) 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, p ou plus lisiblement : **int n, compte, total, p**

II.4. Format libre

Le langage C autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment, le risque existe, si l'on n'y prend garde, d'aboutir à des programmes peu lisibles.

À titre d'exemple, voyez comment pourrait être (mal) présenté notre programme précédent :

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
main() { int i; float
x
; float racx; printf ("Bonjour\n"); printf
("Je vais vous calculer %d racines carrées\n", NFOIS); for (i=
0; i<NFOIS; i++) { printf ("Donnez un nombre : "); scanf ("%f"
, &x); if (x < 0.0)
printf ("Le nombre %f ne possède pas de racine carrée\n", x); else
{ racx = sqrt (x); printf ("Le nombre %f a pour racine carrée : %f\n",
x, racx); } } printf ("Travail terminé - Au revoir");}
```

II.5. 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, 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 fantaisiste &ç§{<>} ?%!!!!!! */

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

/* ========================================
  * commentaire quelque peu esthétique  *
  * et encadré, pouvant servir,          *
  * par exemple, d'en-tête de programme  *
======================================== */
```

Voici un exemple de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int i;              /* compteur de boucle */
float x;            /* nombre dont on veut la racine carrée */
float racx;         /* racine carrée du nombre */
```

Remarque :
Il est autorisé une seconde forme de commentaire, dit « de fin de ligne », que l'on retrouve également en C++. Un tel commentaire est introduit par **//**. Et tout ce qui suit ces deux caractères jusqu'à la fin de la ligne est considéré comme un commentaire. En voici un exemple :

```
printf ("bonjour\n") ;          // formule de politesse
```

III. LA CREATION 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 du programme, compilation et édition de liens.

III.1. Edition 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 qu'on nomme « programme source ». En général, ce texte sera conservé dans un fichier que l'on nommera « fichier source».

Chaque système possède ses propres conventions de dénomination des fichiers. En général, un fichier peut, en plus de son nom, être caractérisé par un groupe de caractères (au moins 3) qu'on appelle une « extension » (ou, parfois un « type »); la plupart du temps, en langage C, les fichiers source porteront l'extension C.

III.2. 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 :

- **traitement par le préprocesseur** : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en langage C pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.
- **compilation proprement dite**, c'est-à-dire traduction en langage machine du texte en langage C fourni par le préprocesseur.

Le résultat de la compilation porte le nom de module objet.

III.3. Edition 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. Notez que cette bibliothèque est une collection de modules objet organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers.

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.

III.4. Fichiers en-tête

Nous avons vu que, grâce à la directive **#include**, vous pouviez demander au préprocesseur d'introduire des instructions (en langage C) provenant de ce que l'on appelle des fichiers « en-tête ». De tels fichiers comportent, entre autres choses :

- des déclarations relatives aux fonctions prédéfinies,
- des définitions de macros prédéfinies.

Lorsqu'on écrit un programme, on ne fait pas toujours la différence entre fonction et macro, puisque celles-ci s'utilisent de la même manière. Toutefois, les fonctions et les macros sont traitées de façon totalement différente par l'ensemble « préprocesseur + compilateur + éditeur de liens ».

En effet, les appels de macros sont remplacés (par du C) par le préprocesseur, du moins si vous avez incorporé le fichier en-tête correspondant. Si vous ne l'avez pas fait, aucun remplacement ne sera effectué, mais aucune erreur de compilation ne sera détectée : le compilateur croira simplement avoir affaire à un appel de fonction; ce n'est que l'éditeur de liens qui, ne la trouvant pas dans la bibliothèque standard, vous fournira un message.

Les fonctions, quant à elles, sont incorporées par l'éditeur de liens. Cela reste vrai, même si vous omettez la directive #include correspondante; dans ce cas, simplement, le compilateur n'aura pas disposé d'informations appropriées permettant d'effectuer des contrôles d'arguments (nombre et type) et de mettre en place d'éventuelles conversions. Aucune erreur ne sera signalée à la compilation ni à l'édition de liens. Les conséquences n'apparaîtront que lors de l'exécution : elles peuvent être invisibles dans le cas de fonctions comme **printf** ou, au contraire, conduire à des résultats erronés dans le cas de fonctions comme **sqrt**.

SEQUENCE II : TYPES DE BASE DU LANGAGE C

Les types **char**, **int** et **float** 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). Ici, nous étudierons en détail ce que l'on appelle les **types de base** du langage C; il s'agit des types scalaires à partir desquels pourront être construits tous les autres, dits « types dérivés », qu'il s'agisse :

- de types structurés comme les tableaux, les structures ou les unions,
- d'autres types simples comme les pointeurs ou les énumérations.

Cependant, nous vous proposons de faire un bref rappel concernant la manière dont l'information est représentée dans un ordinateur et la notion de type qui en découle.

I. 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 : **0100110**, 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). Ils peuvent également représenter une partie d'une instruction machine ou d'un nombre entier codé sur deux octets, ou d'un nombre réel codé sur 4 octets, ou…

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

II. DIFERRENTS TYPES

II.1. Types entiers

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.

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** (qu'on peut abréger en short),
- **int** (c'est celui que nous avons rencontré dans le chapitre précédent),
- **long int** (qu'on peut abréger en long).

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).

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.

TYPE	DESCRIPTION	TAILLE MEMOIRE
int	entier standard (signé)	4 octets: $-2^{31} \leq n \leq 22^{31-1}$
unsigned int	entier positif	4 octets: $0 \leq n \leq 2\ 2^{32}$
short	entier court signé	2 octets: $-2^{15} \leq n \leq 2^{15-1}$
unsigned short	entier court non signé	2 octets: $0 \leq n \leq 22^{16}$
char	caractère signé	1 octet : $-2^{7} \leq n \leq 2^{7-1}$
unsigned char	caractère non signé	1 octet : $0 \leq n \leq 2^{8}$

II.2. Types flottants

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.

Plus précisément, un nombre réel sera représenté en flottant en déterminant deux quantités **M (mantisse)** et **E (exposant)** telles que la valeur $\mathbf{M.B^E}$ représente une approximation de ce nombre. La base B est généralement unique pour une machine donnée (il s'agit souvent de 2 ou de 16) et elle ne figure pas explicitement dans la représentation machine du nombre.

Le C prévoit trois types de flottants correspondant à des tailles différentes : **float, double et long double**. La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable, sauf lorsque l'on doit faire une analyse fine des erreurs de calcul. Par contre, 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 que l'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}.

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale,
- 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, si ce n'est au niveau du temps d'exécution, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons plus tard).

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
48e13	48.e13	48.0E13

Par défaut, toutes les constantes sont créées par le compilateur dans le type double. Il est toutefois possible d'imposer à une constante flottante :

- d'être du type **float**, en faisant suivre son écriture de la lettre F (ou f) : cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision (le gain en place et la perte en précision dépendant de la machine concernée).
- d'être du type **long double**, en faisant suivre son écriture de la lettre L (ou l) : cela permet de gagner éventuellement en précision, en contrepartie d'une perte de place mémoire (le gain en précision et la perte en place dépendant de la machine concernée).

II.3. Types caractères

Comme la plupart des langages, le C permet de manipuler des caractères codés en mémoire sur un octet. Bien entendu, le code employé, ainsi que l'ensemble des caractères représentables, dépendent 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 figurent 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**).

Remarque :
De tels caractères sont souvent nommés « **caractères de contrôle** ». Dans le code ASCII (restreint ou non), ils ont des codes compris entre 0 et 31.

Les constantes de type « caractère », lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quottes) le caractère voulu, comme dans ces exemples :

'a' **'Y'** **'+'** **'$'**

Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère « \ », nommé « **antislash** » (en anglais, il se nomme « back-slash», en français, on le nomme aussi « barre inverse » ou « contre-slash »). Dans cette catégorie, on trouve également quelques 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)	ABRÉVIATION USUELLE	SIGNIFICATION
\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	Tabulation verticale (Vertical Tab)
\\	5C	\	
\'	2C	'	
\"	22	"	
\?	3F	?	

De plus, il est possible d'utiliser directement le code du caractère en l'exprimant, toujours à la suite du caractère « antislash » :

- soit sous forme **octale**,
- soit sous forme **hexadécimale** précédée de **x**.

Voici quelques exemples de notations équivalentes, dans le code ASCII :

```
'A'     '\x41'     '\101'
'\r'    '\x0d'     '\15'    '\015'
'\a'    '\x07'     '\x7'    '\07'    '\007'
```

Remarque :

- En fait, il existe plusieurs versions de code ASCII, mais toutes ont en commun la première moitié des codes (correspondant aux caractères qu'on trouve dans toutes les implémentations); les exemples cités ici appartiennent bien à cette partie commune.
- Le caractère **\,** suivi d'un caractère autre que ceux du tableau ci-dessus ou d'un chiffre de 0 à 7 est simplement ignoré. Ainsi, dans le cas où l'on a affaireau code ASCII, \9 correspond au caractère 9 (de code ASCII 57), tandis que \7 correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».
- En fait, la norme prévoit deux types : **signed char** et **unsigned char** (char correspondant soit à l'un, soit à l'autre, suivant le compilateur utilisé). Là encore, nous y reviendrons. Pour l'instant, sachez que cet attribut de signe n'agit pas sur la représentation d'un caractère en mémoire. Il pourra par contre, avoir un rôle dans le cas où l'on s'intéresse à la valeur numérique associée à un caractère.

II.4. Autres types

Outre les nouveaux types entiers dont nous avons parlé, la norme C99 introduit :

- le type booléen, sous le nom bool; une variable de ce type ne peut prendre que l'une des deux valeurs : vrai (noté true) et faux (noté false);
  #include <stdbool.h>
- des types complexes, sous les noms float complex, double complex et long double complex.

SEQUENCE III : OPERATEURS ET EXPRESSIONS

I. ORIGINALITE DES NOTIONS D'OPERATEUR ET D'EXPRESSION

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*.

En effet, dans la plupart des langages, on trouve, comme en C :

- d'une part, des **expressions** formées (entre autres) à l'aide d'opérateurs,
- d'autre part, des **instructions** pouvant éventuellement faire intervenir des expressions, comme, par exemple, l'instruction d'affectation : **y = a * x +b**; ou encore l'instruction d'affichage : **printf ("Valeur %d", n + 2*p)**; dans laquelle apparaît l'expression **n + 2*p**.

On a à faire à deux notions parfaitement disjointes. En langage C, il en va différemment puisque :

- d'une part, les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possédera une valeur), mais également agir sur le contenu de variables. Ainsi, l'expression (car, comme nous le verrons, il s'agit bien d'une expression en C) : ++i réalisera une action, à savoir : augmenter la valeur i de 1; en même temps, elle aura une valeur, à savoir celle de i après incrémentation.
- d'autre part, une affectation apparemment classique telle que : **i = 5** pourra, à son tour, être considérée comme une expression (ici, de valeur 5). D'ailleurs, en C, l'affectation (=) est un opérateur. Par exemple, la notation suivante : **k = i = 5** représente une expression en C (ce n'est pas encore une instruction - nous y reviendrons). Elle sera interprétée comme : **k = (i = 5)**. Autrement dit, elle affectera à i la valeur 5 puis elle affectera à k la valeur de l'expression **i = 5**, c'est-à-dire 5.

En fait, en C, les notions d'expression et d'instruction sont étroitement liées puisque **la principale instruction** de ce langage est **une expression terminée par un point-virgule**. On la nomme souvent « instruction expression ». Voici des exemples de telles **instructions** qui reprennent les **expressions** évoquées ci-dessus :

```
++i;
i = 5;
k = i = 5;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans les deux cas, il y a évaluation d'une expression (++i ou i=5) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression i=5, c'est-à-dire 5, est à son tour affectée à k; par contre, la valeur finale de l'expression complète est, là encore, inutilisée.

Ce chapitre vous présente la plupart des opérateurs du C ainsi que les règles de priorité et de conversion de type qui interviennent dans les évaluations des expressions. Les (quelques) autres opérateurs concernent essentiellement les pointeurs, l'accès aux tableaux et aux structures et les manipulations de bits. Ils seront exposés dans la suite de ce support.

II. OPERATEURS ARITHMETIQUES

II.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 dans -x+y).

Les opérateurs **binaires** ne sont à 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.

Remarque :
En machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.

Mais nous verrons que par le jeu des conversions implicites, le compilateur saura leur donner une signification :

- soit lorsqu'ils porteront sur des opérateurs de type différent,
- soit lorsqu'ils porteront sur des opérandes de type char ou short.

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 exemples, 11%4 vaut 3; 23%6 vaut 5.

La norme ANSI ne définit les opérateurs % et / que pour des valeurs positives de leurs deux opérandes. Dans les autres cas, le résultat dépend de l'implémentation.

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. Il est nécessaire de faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x3 comme x*x*x), soit à la fonction power de la bibliothèque standard (voyez éventuellement l'annexe).

II.2. 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 (du moins, en ce qui concerne les opérateurs arithmétiques dont nous parlons ici).

Les opérateurs unaires **+** et **-** ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs ***, / et %.** Enfin, sur un dernier niveau, apparaissent les opérateurs binaires **+** et **-**.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a affaireà une ***associativité de gauche à droite*** (nous verrons que quelques opérateurs, autres qu'arithmétiques, utilisent une associativité de droite à gauche).

Enfin, des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez que ces parenthèses peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

a + b * c	a + (b * c)
a * b + c % d	(a * b) + (c % d)
- c % d	(- c) % d
- a + c % d	(- a) + (c % d)
- a / - b + c	((- a) / (- b)) + c
- a / - (b + c)	(- a) / (- (b + c))

Remarque :

- Les règles de priorité interviennent pour définir la signification exacte d'une expression. Néanmoins, lorsque deux opérateurs sont théoriquement commutatifs, on ne peut être certain de l'ordre dans lequel ils seront finalement exécutés. Par exemple, une expression telle que a+b+c pourra aussi bien être calculée en ajoutant c à la somme de a et b, qu'en ajoutant a à la somme de b et c. Même l'emploi de parenthèses dans ce cas ne suffit pas à « forcer » l'ordre des calculs. Notez bien qu'une telle remarque n'a d'importance que lorsque l'on cherche à maîtriser parfaitement les erreurs de calcul.
- Il est tout à fait possible qu'une opération portant sur deux valeurs entières conduise à un résultat non représentable dans le type concerné, parce que en dehors des limites permises, lesquelles, rappelons-le, dépendent de la machine employée. Dans ce cas, la plupart du temps, on obtient un résultat aberrant : les bits excédentaires sont ignorés, le résultat est analogue à celui obtenu lorsque la somme de deux nombres à 3 chiffres est un nombre à quatre chiffres dont on élimine le chiffre de gauche; l'exécution du programme se poursuit sans que l'utilisateur ait été informé d'une quelconque anomalie. Notez bien à

ce propos qu'un opérateur appliqué par exemple à deux opérandes de type int fournit toujours un résultat de type int, même s'il n'est plus représentable dans ce type et qu'un type long aurait pu convenir.

- De la même manière, il se peut qu'à un moment donné vous cherchiez à diviser un entier par zéro. Cette fois, la plupart du temps, cette anomalie est effectivement détectée : un message d'erreur est fourni à l'utilisateur, et l'exécution du programme est interrompue.
- Comme les opérations entières, les opérations sur les flottants peuvent conduire à des résultats non représentables dans le type concerné (de valeur absolue trop grande ou trop petite). Dans ce cas, le comportement dépend des environnements de programmation utilisés; en particulier, il peut y avoir arrêt de l'exécution du programme. Là encore, il faut bien noter qu'un opérateur appliqué par exemple à deux opérandes de type float fournit toujours un résultat de type float, même s'il n'est plus représentable dans ce type et qu'un type double aurait pu convenir.
- En ce qui concerne la division par zéro des flottants, elle conduit toujours à un message et à l'arrêt du programme.

II.3. Conversions implicites pouvant intervenir dans un calcul d'expression

II.3.1. Notion d'expression mixte

Comme nous l'avons dit, les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais vous pouvez écrire ce que l'on nomme des « expressions mixtes » dans lesquelles interviennent des opérandes de types différents.
Voici un exemple d'expression autorisée, dans laquelle n et p sont supposés de type int, tandis que x est supposé de type float : **n * x + p**

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit **n*x**. Pour que cela soit possible, il va mettre en place des instructions de conversion de la valeur de **n** dans le type **float** (car on considère que ce type float permet de représenter à peu près convenablement une valeur entière ; l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type float et elle fournira un résultat de type float.

Pour l'addition, on se retrouve à nouveau en présence de deux opérandes de types différents (float et int). Le même mécanisme sera mis en place, et le résultat final sera de type float.

Remarque :
Attention, le compilateur ne peut que prévoir les instructions de conversion (qui seront donc exécutées en même temps que les autres instructions du programme).

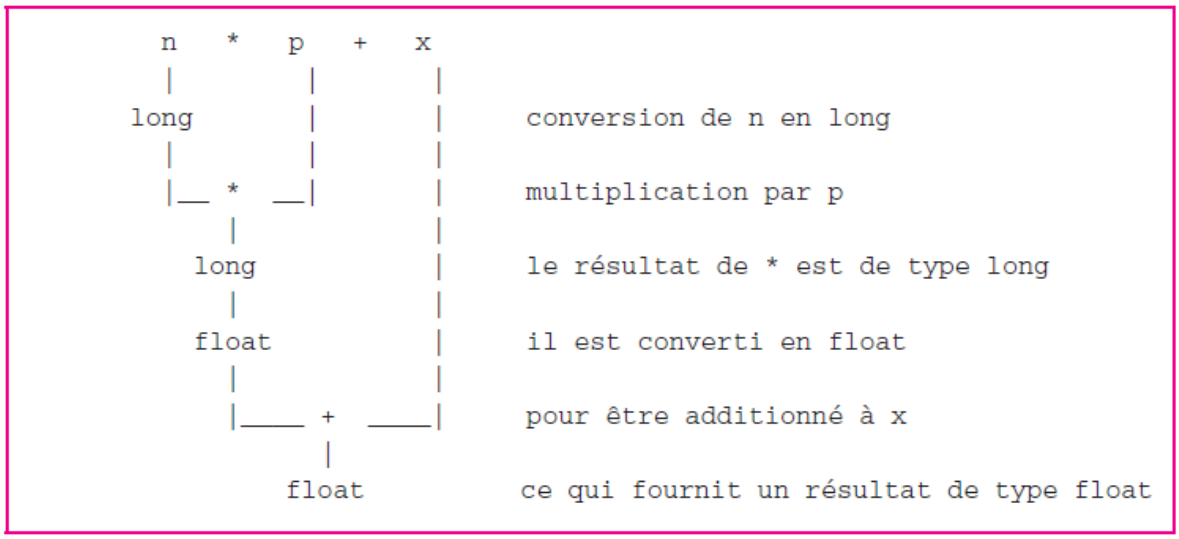
II.3.2. Conversions d'ajustement de type

Une conversion telle que **int → float** se nomme une « **conversion d'ajustement de type** ». Une telle conversion ne peut se faire que suivant une hiérarchie qui permet

de ne pas dénaturer la valeur initiale (on dit parfois que de telles conversions respectent l'intégrité des données).
A savoir : **int → long → float → double → long double**

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

Notez que le choix des conversions à mettre en œuvre est effectué en considérant un à un les opérandes concernés et non pas l'expression de façon globale. Par exemple, si n est de type int, p de type long et x de type float, l'expression : **n * p + x** sera évaluée suivant ce schéma :



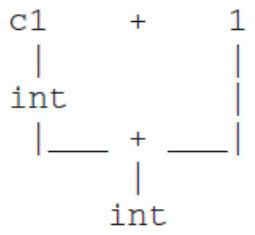
II.3.3. Cas du type char

A priori, vous pouvez être surpris de l'existence d'une conversion systématique de char en int et vous interroger sur sa signification. En fait, il ne s'agit que d'une question de point de vue. En effet, une valeur de type caractère peut être considérée de deux façons :

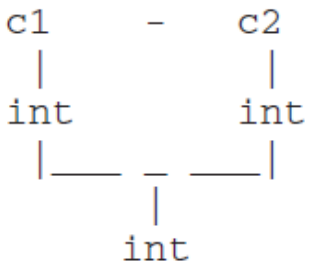
- comme le caractère concerné : a, Z, fin de ligne ;
- comme le code de ce caractère. C'est-à-dire un motif de 8 bits; or à ce dernier on peut toujours faire correspondre un nombre entier (le nombre qui, codé en binaire, fournit le motif en question). Par exemple, dans le code ASCII, le caractère E est représenté par le motif binaire 01000101, auquel on peut faire correspondre le nombre 65.

Effectivement, on peut dire qu'en quelque sorte le langage C confond facilement un caractère avec la valeur (entier) du code qui le représente.

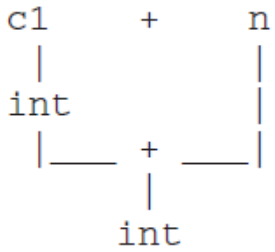
Voici quelques exemples d'évaluation d'expressions, dans lesquels on suppose que c1 et c2 sont de type char, tandis que n est de type int.



L'expression c1+1 fournit donc un résultat de type int, correspondant à la valeur du code du caractère contenu dans c1 augmenté d'une unité.



Ici, bien que les deux opérandes soient de type char, il y a quand même conversion préalable de leurs valeurs en int (promotions numériques).



Remarque :

- Théoriquement, en plus de ce qui vient d'être dit, il faut tenir compte de l'attribut de signe des caractères. Ainsi, lorsque l'on convertit un unsigned char en int, on obtient toujours un nombre entre 0 et 255, tandis que lorsque l'on convertit un signed char en int, on obtient un nombre compris entre -127 et 128. Nous y reviendrons en détail.
- Les arguments d'appel d'une fonction peuvent être également soumis à des conversions. Le mécanisme exact est toutefois assez complexe dans ce cas, car il tient compte de la manière dont la fonction a été déclarée dans le programme qui l'utilise (on peut trouver : aucune déclaration, une déclaration partielle ne mentionnant pas le type des arguments ou une déclaration complète dite prototype mentionnant le type des arguments).
  Lorsque le type des arguments n'a pas été déclaré, les valeurs transmises en argument sont soumises aux règles précédentes (donc, en particulier, aux promotions numériques) auxquelles il faut ajouter la promotion numérique float -> double. Or, précisément, c'est ainsi que sont traitées les valeurs que vous transmettez à printf (ses arguments n'étant pas d'un type connu à l'avance, il est impossible au compilateur d'en connaître le type !). Ainsi :

- tout argument de type char ou short est converti en int; autrement dit, le code %c s'applique aussi à un int : il affichera tout simplement le caractère ayant le code correspondant; de même on obtiendra la valeur numérique du code d'un caractère c en écrivant : printf ("%d", c),
- tout argument de type float sera converti en double (et cela dans toutes les versions du C); ainsi le code %f pour printf correspond-il à un double, et il n'est pas besoin de prévoir un code pour un float.

III. OPERATEURS RELATIONNELS

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

Mais le 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;
- les expressions comparées pourront être d'un type de base quelconque et elles seront soumises aux règles de conversion présentées dans le paragraphe précédent. Cela signifie qu'au bout du compte on ne sera amené à comparer que des expressions de type numérique.

Voici la liste des opérateurs relationnels existant en C. Remarquez bien la notation (==) de l'opérateur d'égalité, le signe = étant, comme nous le verrons, réservé aux affectations. Notez également que = utilisé par mégarde à la place de == ne conduit généralement pas à un diagnostic de compilation, dans la mesure où l'expression ainsi obtenue possède un sens (mais qui n'est pas celui voulu).

OPÉRATEUR	SIGNIFICATION
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

En ce qui concerne leurs priorités, il faut savoir que les quatre premiers opérateurs (<, <=, >, >=) sont de même priorité. Les deux derniers (== et !=) possèdent également la même priorité, mais celle-ci est inférieure à celle des précédents. Ainsi, l'expression : **a < b == c < d** est interprétée comme : **(a < b) == (c < d)** ce qui, en C, a effectivement une signification, étant donné que les expressions **a < b** et **c < d** sont, finalement, des quantités entières. En fait, cette expression prendra la valeur 1 lorsque les relations a < b et c < d auront toutes les deux la même valeur, c'est-à-dire soit lorsqu'elles

seront toutes les deux vraies, soit lorsqu'elles seront toutes les deux fausses. Elle prendra la valeur 0 dans le cas contraire.

D'autre part, ces opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques. Cela permet souvent d'éviter certaines parenthèses dans des expressions. Ainsi :
x + y < a + 2
est équivalent à :
(x + y) < (a + 2)

Remarque importante (c*omparaisons 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 (c1 et c2 étant de type char) :

- c1 == c2 sera vraie si c1 et c2 ont la même valeur, c'est-à-dire si c1 et c2 contiennent des caractères de même code, donc si c1 et c2 contiennent le même caractère,
- c1 == 'e' sera vraie si le code de c1 est égal au code de 'e', donc si c1 contient le caractère e.

Autrement dit, dans ces circonstances, l'existence d'une conversion char → int n'a guère d'influence. En revanche, pour les comparaisons d'inégalité, quelques précisions s'imposent. En effet, par exemple c1 < c2 sera vraie si le code du caractère de c1 a une valeur inférieure au code du caractère de c2. Le résultat d'une telle comparaison peut donc varier suivant le codage employé. Cependant, il faut savoir que, quel que soit ce codage :

- 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 classés par ordre naturel; on a toujours '2' < '5' ...

IV. OPERATEURS LOGIQUES

Le C dispose de trois opérateurs logiques classiques : **et** (noté **&&**), **ou** (noté **||**) et **non** (noté **!**).

Par exemples :

- **(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), et prend la valeur 0 (faux) dans le cas contraire.
- **(a<b) || (c<d)**
  prend la valeur 1 (vrai) si l'une au moins des deux conditions a<b et c<d est vraie (de valeur 1), et prend la valeur 0 (faux) dans le cas contraire.
- **! (a<b)**
  prend la valeur 1 (vrai) si la condition a<b est fausse (de valeur 0) et prend la valeur 0 (faux) dans le cas contraire. Cette expression est équivalente à : a>=b.

Il est important de constater que, ne disposant pas de type logique, le 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.

Le tableau suivant récapitule la situation.

OPERANDE 1	OPERATEUR	OPERANDE 2	RESULTAT
0	&&	0	0
0	&&	non nul	0
non nul	&&	0	0
non nul	&&	non nul	1
0	||	0	0
0	||	non nul	1
non nul	||	0	1
non nul	||	non nul	1
	!	0	1
	!	non nul	0

Ainsi, en C, si n et p sont des entiers, des expressions telles que : **n && p, n || p, !n** sont acceptées par le compilateur. Notez que l'on rencontre fréquemment l'écriture : **if (!n)** plus concise (mais pas forcément plus lisible) que : **if (n == 0)**

L'opérateur **!** a une priorité supérieure à celle de tous les opérateurs arithmétiques binaires et aux opérateurs relationnels. Ainsi, pour écrire la condition contraire de : **a == b**, il est nécessaire d'utiliser des parenthèses en écrivant : **!(a == b)**

En effet, l'expression : **!a == b**, serait interprétée comme : **(!a) == b**

L'opérateur **||** est moins prioritaire que **&&**. Tous deux sont de priorité inférieure aux opérateurs arithmétiques ou relationnels. Ainsi, les expressions utilisées comme exemples en début de ce paragraphe auraient pu, en fait, être écrites sans parenthèses :

a<b && c<d équivaut à (a<b) && (c<d)
a<b || c<d équivaut à (a<b) || (c<d)

Enfin, les deux opérateurs **&&** et **||** jouissent en C d'une propriété intéressante : **leur second opérande** (celui qui figure à droite de l'opérateur) **n'est évalué que si la connaissance de sa valeur est indispensable** pour décider si l'expression correspondante est vraie ou fausse. Par exemple, dans une expression telle que : a<b && c<d on commence par évaluer a<b. Si le résultat est faux (0), il est inutile d'évaluer c<d puisque, de toute façon, l'expression complète aura la valeur faux (0).

La connaissance de cette propriété est indispensable pour maîtriser des « constructions » telles que : **if (i<max && ((c=getchar()) != '\n'))**

En effet, le second opérande de l'opérateur &&, à savoir : c = getchar() != '\n' fait appel à la lecture d'un caractère au clavier. Celle-ci n'aura donc lieu que si la première condition (i<max) est vraie.

V. OPERATEUR 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**,
- possédait une valeur : celle de i après affectation, c'est-à-dire 5.

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.

Il n'est par contre 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

VI.1. Notion de lvalue

Nous voyons donc que cet opérateur d'affectation impose des restrictions sur son premier opérande. En effet, ce dernier doit être une référence à un emplacement mémoire dont on pourra effectivement modifier la valeur.

Dans les autres langages, on désigne souvent une telle référence par le nom de «variable» ; on précise généralement que ce terme recouvre par exemple les éléments de tableaux ou les composantes d'une structure. En langage C, cependant, la syntaxe du langage est telle que cette notion de variable n'est pas assez précise. Il faut introduire un mot nouveau : la **lvalue**. Ce terme désigne une « valeur à gauche », c'est-à-dire tout ce qui peut apparaître à gauche d'un opérateur d'affectation.

Certes, pour l'instant, vous pouvez trouver que dire qu'à gauche d'un opérateur d'affectation doit apparaître une *lvalue* n'apporte aucune information. En fait, d'une part, nous verrons qu'en C d'autres opérateurs que = font intervenir une *lvalue* ; d'autre part, au fur et à mesure que nous rencontrerons de nouveaux types d'objets, nous préciserons s'ils peuvent être ou non utilisés comme *lvalue*.

Pour l'instant, les seules *lvalue* que nous connaissons restent les variables de n'importe quel type de base déjà rencontré.

VI.2. Opérateur d'affectation possède une associativité de droite à gauche

Contrairement à tous ceux que nous avons rencontrés jusqu'ici, cet opérateur d'affectation possède une associativité de *droite à gauche*. C'est ce qui permet à une expression telle que : **i = j = 5**
d'évaluer d'abord l'expression j = 5 avant d'en affecter la valeur (5) à la variable j.

Bien entendu, la valeur finale de cette expression est celle de i après affectation, c'est-à-dire 5.

VI.3. Affectation peut entraîner une conversion

Là encore, la grande liberté offerte par le langage C en matière de mixage de types se traduit par la possibilité de fournir à cet opérateur d'affectation des opérandes de types différents.
Cependant, contrairement à ce qui se produisait pour les opérateurs rencontrés précédemment et qui mettaient en jeu des conversions implicites, il n'est plus question, ici, d'effectuer une quelconque conversion de la *lvalue* qui apparaît à gauche de cet opérateur. Une telle conversion reviendrait à changer le type de la *lvalue* figurant à gauche de cet opérateur, ce qui n'a pas de sens.

En fait, lorsque le type de l'expression figurant à droite n'est pas du même type que la *lvalue* figurant à gauche, il y a **conversion systématique** de la valeur de l'expression (qui est évaluée suivant les règles habituelles) dans le type de la *lvalue*. Une telle conversion **imposée** ne respecte plus nécessairement la hiérarchie des types qui est de rigueur dans le cas des conversions implicites. Elle peut donc conduire, suivant les cas, à une dégradation plus ou moins importante de l'information (par exemple lorsque l'on convertit un double en int, on perd la partie décimale du nombre).

Nous ferons le point sur ces différentes possibilités de conversions imposées par les affectations dans un autre paragraphe.

VII. OPERATEURS D'INCREMENTATION ET DE DECREMENTATION

VII.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 ». 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 variable 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 **préincrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postincrémentation** 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;
```

De la même manière, il existe un opérateur de décrémentation noté -- qui, suivant les cas, sera :

- un opérateur de **prédécrémentation** lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de **postdécrémentation** lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

VII.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 *lvalue* sur laquelle ils portent. Ainsi, l'expression suivante a un sens : 3 * i++ * j-- + k++
(si * avait été plus prioritaire que la postincrémentation, 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).

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.

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

VIII. LES OPERATEURS D'AFFECTATION ELARGIE

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 le 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 » : **+= -= *= /= %= |= ^= &= <<= >>=**

Les cinq derniers correspondent en fait à des « opérateurs de manipulation de bits » (|, ^, &, << et >>) que nous n'aborderons que dans le chapitre 13. 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.

Ne confondez pas l'opérateur de comparaison <= avec un opérateur d'affectation élargie.

Notez bien que les opérateurs de comparaison ne sont pas concernés par cette possibilité.

IX. LES CONVERSIONS FORCEES PAR UNE AFFECTATION

Nous avons déjà vu comment le compilateur peut être amené à introduire des conversions implicites dans l'évaluation des expressions. Dans ce cas, il applique les règles de promotions numériques et d'ajustement de type.

Par ailleurs, une affectation introduit une conversion d'office dans le type de la *lvalue* réceptrice, dès lors que cette dernière est d'un type différent de celui de l'expression correspondante.

Par exemple, si **n** est de type **int** et **x** de type **float**, l'affectation : **n = x + 5.3;** entraînera tout d'abord l'évaluation de l'expression située à droite, ce qui fournira une valeur de type float; cette dernière sera ensuite convertie en int pour pouvoir être affectée à n.

D'une manière générale, lors d'une affectation, toutes les conversions (d'un type numérique vers un autre type numérique) sont acceptées par le compilateur mais le résultat en est plus ou moins satisfaisant. En effet, si aucun problème ne se pose (autre qu'une éventuelle perte de précision) dans le cas de conversion ayant lieu suivant le bon sens de la hiérarchie des types, il n'en va plus de même dans les autres cas.

Par exemple, la conversion float -> int (telle que celle qui est mise en jeu dans l'instruction précédente) ne fournira un résultat acceptable que si la partie entière de la valeur flottante est représentable dans le type int. Si une telle condition n'est pas réalisée, non seulement le résultat obtenu pourra être différent d'un environnement à un autre mais, de surcroît, on pourra aboutir, dans certains cas, à une erreur d'exécution.

De la même manière, la conversion d'un int en char sera satisfaisante si la valeur de l'entier correspond à un code d'un caractère.

Sachez, toutefois, que les conversions d'un type entier vers un autre type entier ne conduisent, au pis, qu'à une valeur inattendue mais jamais à une erreur d'exécution.

X. OPERATEUR DE CAST

S'il le souhaite, le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'un opérateur un peu particulier nommé en anglais « **cast** ».

Si, par exemple, n et p sont des variables entières, l'expression : **(double) (n/p)** aura comme valeur celle de l'expression entière n/p convertie en double.

La notation (double) correspond en fait à un opérateur unaire dont le rôle est d'effectuer la conversion dans le type double de l'expression sur laquelle il porte.

Notez bien que cet opérateur force la conversion du *résultat* de l'expression et non celle des différentes valeurs qui concourent à son évaluation. Autrement dit, ici, il y a d'abord calcul, dans le type int, du quotient de n par p; c'est seulement ensuite que le résultat sera converti en double. Si n vaut 10 et que p vaut 3, cette expression aura comme valeur 3.
D'une manière générale, il existe autant d'opérateurs de « cast » que de types différents (y compris les types dérivés comme les pointeurs que nous rencontrerons ultérieurement). Leur priorité élevée (voir tableau en fin de chapitre) fait qu'il est généralement nécessaire de placer entre parenthèses l'expression concernée. Ainsi, l'expression : **(double) n/p** conduirait d'abord à convertir n en double; les règles de conversions implicites amèneraient alors à convertir p en double avant qu'ait lieu la division (en double). Le résultat serait alors différent de celui obtenu par l'expression proposée en début de ce paragraphe (avec les mêmes valeurs de n et de p, on obtiendrait une valeur de l'ordre de 3.33333...).

Bien entendu, comme pour les conversions forcées par une affectation, toutes les conversions numériques sont réalisables par un opérateur de « cast », mais le résultat en est plus ou moins satisfaisant (revoyez éventuellement le paragraphe précédent).

XI. OPERATEUR CONDITIONNEL

Considérons l'instruction suivante :

```
if (a>b)
        max = a;
else
        max = b;
```

Elle attribue à la variable max la plus grande des deux valeurs de a et de b. La valeur de max pourrait être définie par cette phrase : **Si a>b alors a sinon b**

En langage C, il est possible, grâce à l'aide de **l'opérateur conditionnel**, de traduire presque littéralement la phrase ci-dessus de la manière suivante : **max = a>b? a : b**

L'expression figurant à droite de l'opérateur d'affectation est en fait constituée de trois expressions (a>b, a et b) qui sont les trois opérandes de l'opérateur conditionnel, lequel se matérialise par *deux symboles séparés :* ? et :.

D'une manière générale, cet opérateur évalue la première expression qui joue le rôle d'une condition. Comme toujours en C, celle-ci peut être en fait de n'importe quel type. Si sa valeur est différente de zéro, il y a évaluation du second opérande, ce qui fournit le résultat; si sa valeur est nulle, en revanche, il y a évaluation du troisième opérande, ce qui fournit le résultat.

Voici un autre exemple d'une expression calculant la valeur absolue de 3*a + 1 :

```
3*a+1 > 0 ? 3*a+1 : -3*a-1
```

L'opérateur conditionnel dispose d'une faible priorité (il arrive juste avant l'affectation), de sorte qu'il est rarement nécessaire d'employer des parenthèses pour en délimiter les différents opérandes (bien que cela puisse parfois améliorer la

lisibilité du programme). Voici, toutefois, un cas où les parenthèses sont indispensables : z = (x=y) ? a : b

Le calcul de cette expression amène tout d'abord à affecter la valeur de y à x. Puis, si cette valeur est non nulle, on affecte la valeur de a à z. Si, au contraire, cette valeur est nulle, on affecte la valeur de b à z. Il est clair que cette expression est différente de : z = x = y ? a : b laquelle serait évaluée comme : z = x = (y ? a : b)

Bien entendu, une expression conditionnelle peut, comme toute expression, apparaître à son tour dans une expression plus complexe. Voici, par exemple, une instruction (notez qu'il s'agit effectivement d'une instruction, car elle se termine par un point-virgule) affectant à z la plus grande des valeurs de a et de b :
z = (a>b ? a : b);

De même, rien n'empêche que l'expression conditionnelle soit évaluée sans que sa valeur soit utilisée comme dans cette instruction : a>b ? i++ : i--;

Ici, suivant que la condition a>b est vraie ou fausse, on incrémentera ou on décrémentera la variable i.

XII. OPERATEUR SEQUENTIEL

Nous avons déjà vu qu'en C la notion d'expression était beaucoup plus générale que dans la plupart des autres langages. L'opérateur dit « séquentiel » va élargir encore un peu plus cette notion d'expression. En effet, celui-ci permet, en quelque sorte, d'exprimer *plusieurs calculs successifs au sein d'une même expression.* Par exemple : a * b, i + j est une expression qui évalue d'abord a*b, puis i+j et qui prend comme valeur la dernière calculée (donc ici celle de i+j). Certes, dans ce cas d'école, le calcul préalable de a*b est inutile puisqu'il n'intervient pas dans la valeur de l'expression globale et qu'il ne réalise aucune action.

En revanche, une expression telle que : i++, a + b peut présenter un intérêt puisque la première expression (dont la valeur ne sera pas utilisée) réalise en fait une incrémentation de la variable i.

Il en est de même de l'expression suivante : i++, j = i + k dans laquelle, il y a :

- évaluation de l'expression i++,
- évaluation de l'affectation j = i + k. Notez qu'alors on utilise la valeur de i après incrémentation par l'expression précédente.

Cet opérateur séquentiel, qui dispose d'une associativité de gauche à droite, peut facilement faire intervenir plusieurs expressions (sa faible priorité évite l'usage de parenthèses) : i++, j = i+k, j--

Certes, un tel opérateur peut être utilisé pour réunir plusieurs instructions en une seule. Ainsi, par exemple, ces deux formulations sont équivalentes :
i++, j = i+k, j--;
i++; j = i+k; j--;

Dans la pratique, ce n'est cependant pas là le principal usage que l'on fera de cet opérateur séquentiel. En revanche, ce dernier pourra fréquemment intervenir dans les instructions de choix ou dans les boucles; là où celles-ci s'attendent à trouver une seule expression, l'opérateur séquentiel permettra d'en placer plusieurs, et donc d'y réaliser plusieurs calculs ou plusieurs actions.

En voici deux exemples :
```
if (i++, k>0) ……
        remplace :
i++; if (k>0) ……
```

et :
```
for (i=1, k=0; …; … ) …….
        remplace :
i=1; for (k=0; …; … ) ……
```

Compte tenu de ce que l'appel d'une fonction n'est en fait rien d'autre qu'une expression, la construction suivante est parfaitement valide en C :
```
for (i=1, k=0, printf("on commence"); …; …) ……
```

Nous verrons même que, dans le cas des boucles conditionnelles, cet opérateur permet de réaliser des constructions ne possédant pas d'équivalent simple.

XIII. OPERATEUR SIZEOF

L'opérateur **sizeof**, dont l'emploi ressemble à celui d'une fonction, fournit la taille en octets (n'oubliez pas que l'octet est, en fait, la plus petite partie adressable de la mémoire).

Par exemple, dans une implémentation où le type int est représenté sur 2 octets et le type double sur 8 octets, si l'on suppose que l'on a affaire à ces déclarations :
```
int n;
double z;
```
- l'expression sizeof(n) vaudra 2,
- l'expression sizeof(z) vaudra 8.

Cet opérateur peut également s'appliquer à un type de nom donné. Ainsi, dans l'implémentation précédemment citée :
- sizeof(int) vaudra 2,
- sizeof(double) vaudra 8.

Quelle que soit l'implémentation, sizeof(char) vaudra toujours 1 (par définition, en quelque sorte).

Cet opérateur offre un intérêt :
- lorsque l'on souhaite écrire des programmes portables dans lesquels il est nécessaire de connaître la taille exacte de certains objets,
- pour éviter d'avoir à calculer soi-même la taille d'objets d'un type relativement complexe pour lequel on n'est pas certain de la manière dont il sera implémenté par le compilateur.

Ce sera notamment le cas des structures.

Exercice III.1
Soit les déclarations suivantes :
int n = 10 , p = 4;
long q = 2;
float x = 1.75;

Donner le type et la valeur de chacune des expressions suivantes :
a) n + q
b) n + x
c) n % p +q
d) n < p
e) n >= p
f) n > q
g) q + 3 * (n > p)
h) q && n
i) (q-2) && (n-10)
j) x * (q==2)
k) x *(q=5)

Exercice III.2
Écrire plus simplement l'instruction suivante :
z = (a>b ? a : b) + (a <= b ? a : b);

Exercice III.3
n étant de type int, écrire une expression qui prend la valeur :
-1 si n est négatif,
0 si n est nul,
1 si n est positif.

Exercice III.4
Quels résultats fournit le programme suivant ?

```
#include <stdio.h>
main()
{
int n=10, p=5, q=10, r;
r = n == (p = q);
printf ("A : n = %d p = %d q = %d r = %d\n", n, p, q, r);
n = p = q = 5;
n += p += q;
printf ("B : n = %d p = %d q = %d\n", n, p, q);
q = n < p ? n++ : p++;
printf ("C : n = %d p = %d q = %d\n", n, p, q);
q = n > p ? n++ : p++;
printf ("D : n = %d p = %d q = %d\n", n, p, q);
```

SEQUENCE IV : LES ENTREES SORTIES CONVERSATIONNELLES

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

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

Nous nous limiterons ici à ce que nous avons appelé les « entrées-sorties conversationnelles ».

Plus tard, nous verrons que ces mêmes fonctions (moyennant la présence d'un argument supplémentaire) permettent également d'échanger des informations avec des fichiers.

En ce qui concerne la lecture au clavier, nous serons amenés à mettre en évidence certaines lacunes de scanf en matière de comportement lors de réponses incorrectes et à vous fournir quelques idées sur la manière d'y remédier.

I. 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. Nous avons toutefois mentionné le code de conversion relatif aux chaînes (qui ne seront abordées que dans le chapitre 8) et les entiers non signés (chapitre 13).

I.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 en notation 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.xxxxxx-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)

I.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.

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

printf ("%3d", n);	/* entier avec 3 caractères minimum */
n = 20	^20
n = 3	^^3
n = 2358	2358
n = -5200	-5200

printf ("%f", x);	/* notation décimale gabarit par défaut */ /* (6 chiffres après point) */
x = 1.2345	1.234500
x = 12.3456789	12.345679

printf ("%10f", x);	/* notation décimale - gabarit mini 10 */ /* (toujours 6 chiffres après point) */
x = 1.2345	^^1.234500
x = 12.345	^12.345000
x = 1.2345E5	123450.000000

printf ("%e", x);	/* notation exponentielle - gabarit par défaut */ /* (6 chiffres après point) */
x = 1.2345	1.234500e+000

x = 123.45	1.234500e+002
x = 123.456789E8	1.234568e+010
x = -123.456789E8	-1.234568e+010

I.3. Actions sur la précision

Pour les types flottants, on peut spécifier un nombre de chiffres (éventuellement inférieur à 6) après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle).

Ce nombre doit apparaître, précédé d'un point, avant le code de format (et éventuellement après le gabarit).

Voici quelques exemples :

printf ("%10.3f", x);	/* notation décimale, gabarit mini 10 */ /* et 3 chiffres après point */
x = 1.2345	^^^^^1.235
x = 1.2345E3	^^1234.500
x = 1.2345E7	12345000.000

printf ("%12.4e", x);	/* notation exponentielle, gabarit mini 12*/ /* et 4 chiffres après point */
x = 1.2345	^1.2345e+000
x = 123.456789E8	^1.2346e+010

Remarques :

- Le signe moins (-), placé immédiatement après le symbole % (comme dans %-4d ou %-10.3f), demande de cadrer l'affichage à gauche au lieu de le cadrer (par défaut) à droite; les éventuels espaces supplémentaires sont donc placés à droite et non plus à gauche de l'information affichée.
- Le caractère * figurant à la place d'un gabarit ou d'une précision signifie que la valeur effective est fournie dans la liste des arguments de printf. En voici un exemple dans lequel nous appliquons ce mécanisme à la précision :

printf ("%8.*f", n, x);	
n = 1 x = 1.2345	^^^^1.2
n = 3 x = 1.2345	^^^1.234

- La fonction printf fournit en fait une valeur de retour. Il s'agit du nombre de caractères qu'elle a réellement affichés (ou la valeur -1 en cas d'erreur). Par exemple, avec l'instruction suivante, on s'assure que l'opération d'affichage s'est bien déroulée : if (printf ("....", ...) != -1) .....

  De même, on obtient le nombre de caractères effectivement affichés par :
  n = printf ("....", ...)

I.4. Syntaxe de *printf*

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

printf (format, liste_d'expressions) ;

- format :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- liste_d'expressions : suite d'expressions séparées par des virgules d'un type en accord avec le code format correspondant.

Remarque :
Nous verrons que les deux notions de constante chaîne et de pointeur sur une chaîne sont identiques.

I.5. En cas d'erreur de programmation

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

- ***code de format en désaccord avec le type de l'expression***

Lorsque le code de format, bien qu'erroné, correspond à une information de **même taille** (c'est-à-dire occupant la même place en mémoire) que celle relative au type de l'expression, les conséquences de l'erreur se limitent à une *mauvaise interprétation de l'expression*. C'est ce qui se passe, par exemple, lorsque l'on écrit une valeur de type int en %u ou une valeur de type unsigned int en %d.
En revanche, lorsque le code format correspond à une information de **taille différente** de celle relative au type de l'expression, les conséquences sont généralement plus désastreuses, du moins si d'autres valeurs doivent être affichées à la suite. En effet, tout se passe alors comme si, dans la suite d'octets (correspondant aux différentes valeurs à afficher) reçue par printf, le repérage des emplacements des valeurs suivantes se trouvait soumis à un décalage.

- ***nombre de codes de format différent du nombre d'expressions de la liste***

Dans ce cas, il faut savoir que **C cherche toujours à satisfaire le contenu du format**.
Ce qui signifie que, si des expressions de la liste n'ont pas de code format, elles seront pas affichées. C'est le cas dans cette instruction où la valeur de p ne sera pas affichée :

printf ("%d", n, p);

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

I.6. La macro *putchar*

L'expression : **putchar(c)** 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 En toute rigueur, la fonction recherchée pourra porter un nom légèrement différent, par exemple _putchar; c'est ce nom qui figurera dans le message d'erreur fourni par l'éditeur de liens.

II. LES POSSIBILITES 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.

Ici, nous allons vous montrer le fonctionnement de scanf. Comme nous l'avons fait pour printf, nous vous présenterons d'abord les principaux codes de conversion. Là encore, nous avons également mentionné les codes de conversion relatifs aux chaînes et aux entiers non signés.

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.

II.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)

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.

II.2. Premières notions de tampon et de séparateurs

Lorsque scanf attend que vous lui fournissiez des données, l'information frappée au clavier est rangée temporairement dans l'emplacement mémoire nommé « tampon ». Ce dernier est exploré, caractère par caractère par scanf, au fur et à mesure des besoins. Il existe un pointeur qui précise quel est le prochain caractère à prendre en compte.

D'autre part, certains caractères dits « séparateurs » (ou « espaces blancs ») jouent un rôle particulier dans les données. Les deux principaux sont l'espace et la fin de ligne (\n). Il en existe trois autres d'un usage beaucoup moins fréquent : la tabulation horizontale (\t), la tabulation verticale (\v) et le changement de page (\f).

II.3 Les premières règles utilisées par *scanf*

Les codes de format correspondant à un nombre (c'est-à-dire tous ceux de la liste précédente, excepté %c et %s) entraînent d'abord l'avancement éventuel du pointeur jusqu'au premier caractère différent d'un séparateur. Puis scanf prend en compte tous les caractères suivants jusqu'à la rencontre d'un séparateur (en y plaçant le pointeur), du moins lorsque aucun gabarit n'est précisé (comme nous apprendrons à le faire dans le paragraphe 2.4) et qu'aucun caractère invalide n'est présent dans la donnée.

Quant au code de format %c, il entraîne la prise en compte du caractère désigné par le pointeur (même s'il s'agit d'un séparateur comme espace ou fin de ligne), et le pointeur est simplement avancé sur le caractère suivant du tampon.

Voici quelques exemples dans lesquels nous supposons que n et p sont de type int, tandis que c est de type char. Nous fournissons, pour chaque appel de scanf, des exemples de réponses possibles (^ désigne un espace et @ une fin de ligne) et, en regard, les valeurs effectivement lues.

scanf ("%d%d", &n, &p);	
12^25@	n = 12 p = 25
^12^^25^^@	n = 12 p = 25
12@ @ ^25@	 n = 12 p = 25

scanf ("%c%d", &c, &n);	
a25@	c = 'a' n = 25

a^^25@	c = 'a' n = 25

scanf ("%d%c", &n, &c);	
12 a@	n = 12 c = ' '

Notez que, dans ce cas, on obtient bien le caractère « espace » dans c. Nous verrons dans le paragraphe 2.5 comment imposer à scanf de sauter quand même les espaces dans ce cas.

Remarque :
Le code de format précise la nature du travail à effectuer pour transcoder une partie de l'information frappée au clavier, laquelle n'est en fait qu'une suite de caractères (codés chacun sur un octet) pour fabriquer la valeur (binaire) de la variable correspondante. Par exemple, %d entraîne en quelque sorte une double conversion : suite de caractères -> nombre écrit en décimal -> nombre codé en binaire; la première conversion revient à faire correspondre un nombre entre 0 et 9 à un caractère représentant un chiffre. En revanche, le code %c demande simplement de ne rien faire puisqu'il suffit de recopier tel quel l'octet contenant le caractère concerné.

II.4. Imposition d'un gabarit maximal

Comme dans les codes de format de **printf**, on peut, dans un code de format de scanf, préciser un gabarit. Dans ce cas, le traitement d'un code de format s'interrompt soit à la rencontre d'un séparateur, soit lorsque le nombre de caractères indiqués a été atteint (attention, les séparateurs éventuellement sautés auparavant ne sont pas comptabilisés !).

Voici un exemple :

scanf ("%3d%3d", &n, &p)	
12^25@	n = 12 p = 25
^^^^^12345@	n = 123 p = 45
12@ 25@	n = 12 p = 25

II.5. Rôle d'un espace dans le format

Un espace entre deux codes de format demande à scanf de faire avancer le pointeur au prochaincaractère différent d'un séparateur. Notez que c'est déjà ce qui se passe lorsque l'on aaffaire à un code de format correspondant à un type numérique. En revanche, cela n'était pasle cas pour les caractères, comme nous l'avons vu au paragraphe 2.3.

Voici un exemple :

scanf ("%d^%c", &n, &c);	/* ^ désigne un espace */ /* %d^%c est différent de %d%c */
12^a@	n = 12 c = 'a'
12^^^a@	n = 12 c = 'a'

12@a@	n = 12 c = 'a'

II.6. Cas où un caractère invalide apparaît dans une donnée

Voyez cet exemple, accompagné des valeurs obtenues dans les variables concernées :

scanf ("%d^%c", &n, &c);	/* ^ désigne un espace */
12a@	n = 12 c = 'a'

Ce cas fait intervenir un mécanisme que nous n'avons pas encore rencontré. Il s'agit d'un troisième critère d'arrêt du traitement d'un code format (les deux premiers étaient : rencontre d'un séparateur ou gabarit atteint).

Ici, lors du traitement du code %d, scanf rencontre les caractères 1, puis 2, puis a. Ce caractère a ne convenant pas à la fabrication d'une valeur entière, scanf interrompt son exploration et fournit donc la valeur 12 pour n. L'espace qui suit %d dans le format n'a aucun effet puisque le caractère courant est le caractère a (différent d'un séparateur). Le traitement du code suivant, c'est-à-dire %c, amène scanf à prendre ce caractère courant (a) et à l'affecter à la variable c.

D'une manière générale, dans le traitement d'un code de format, scanf arrête son exploration du tampon dès que l'une des trois conditions est satisfaite :

- rencontre d'un caractère séparateur,
- gabarit maximal atteint (s'il y en a un de spécifié),
- rencontre d'un caractère invalide, par rapport à l'usage qu'on veut en faire (par exemple un point pour un entier, une lettre autre que E ou e pour un flottant,...). Notez bien l'aspect relatif de cette notion de caractère invalide.

II.7. Arrêt prématuré de *scanf*

Voyez cet exemple, dans lequel nous utilisons, pour la première fois, la valeur de retour de la fonction scanf.

compte = scanf ("%d^%d^%c", &n, &p, &c);			/* ^ désigne un espace */	
12^25^b@	n = 12	p = 25	c = 'b'	compte = 3
12b@	n = 12	p inchangé	c inchangé	compte = 1
b@	n indéfini	p inchangé	c inchangé	compte = 0

La valeur fournie par scanf n'est pas comparable à celle fournie par printf puisqu'il s'agit cette fois du **nombre de valeurs convenablement lues**. Ainsi, dans le premier cas, il n'est pas surprenant de constater que cette valeur est égale à 3.
En revanche, dans le deuxième cas, le caractère b a interrompu le traitement du premier code %d.
Dans le traitement du deuxième code (%d), scanf a rencontré d'emblée ce caractère b, toujours invalide pour une valeur numérique. Dans ces conditions, scanf se trouve dans l'incapacité d'attribuer une valeur à p (puisque ici, contrairement à ce qui s'est passé pour n, elle ne dispose d'aucun caractère correct). Dans un tel cas, scanf **s'interrompt sans chercher à lire d'autres valeurs** et fournit, en retour, le nombre

de valeurs correctement lues jusqu'ici, c'est-à-dire 1. Les valeurs de p et de c restent inchangées (éventuellement indéfinies).

Dans le troisième cas, le même mécanisme d'arrêt prématuré se produit dès le traitement du premier code de format, et le nombre de valeurs correctement lues est 0.

Ne confondez pas cet arrêt prématuré de *scanf* avec le troisième critère d'arrêt de traitement d'un code de format. En effet, les deux situations possèdent bien la même cause (un caractère invalide par rapport à l'usage que l'on souhaite en faire), mais seul le cas où scanf n'est pas en mesure de fabriquer une valeur conduit à l'arrêt prématuré.

Ici, nous avons vu la signification d'un espace introduit entre deux codes de format. En toute rigueur, vous pouvez introduire à un tel endroit n'importe quel caractère de votre choix. Dans ce cas, sachez que lorsque scanf rencontre un caractère (x par exemple) dans le format, il le compare avec le caractère courant (celui désigné par le pointeur) du tampon. S'ils sont égaux, il poursuit son travail (après avoir avancé le pointeur) mais, dans le cas contraire, il y a **arrêt prématuré**. Une telle possibilité ne doit toutefois être réservée qu'à des cas bien particuliers.

II.8. La syntaxe de *scanf*

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

- format :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- liste_d_adresses : liste de *lvalue*, séparées par des virgules, d'un type en accord avec le code de format correspondant.

II.9. Problèmes de synchronisation entre l'écran et le clavier

Voyez cet exemple de programme accompagné de son exécution alors que nous avons répondu **: 12^25@** à la première question posée.

L'écran et le clavier semblent mal synchronisés

```
#include <stdio.h>
main()
{
int n, p;
printf ("donnez une valeur pour n : ");
scanf ("%d", &n);
printf ("merci pour %d\n", n);
printf ("donnez une valeur pour p : ");
scanf ("%d", &p);
printf ("merci pour %d", p);
}
```

```
donnez une valeur pour n : 12 25
merci pour 12
donnez une valeur pour p : merci pour 25
```

Vous constatez que la seconde question (donnez une valeur pour p) est apparue à l'écran, maisle programme n'a pas attendu que vous frappiez votre réponse pour vous afficher la suite. Vousnotez alors qu'il a bien pris pour p la seconde valeur entrée au préalable, à savoir 25.

En fait, comme nous l'avons vu, les informations frappées au clavier ne sont pas traitées instantanément par scanf mais mémorisées dans un tampon. Jusqu'ici, cependant, nous n'avions pas précisé quand scanf s'arrêtait de mémoriser pour commencer à traiter. Il le fait tout naturellement à la rencontre d'un caractère de fin de ligne généré par la frappe de la touche « return », dont le rôle est aussi classiquement celui d'une validation. Notez que, bien qu'il joue le rôle d'une validation, ce caractère de fin de ligne est quand même recopié dans le tampon; il pourra donc éventuellement être lu en tant que tel.

L'élément nouveau réside donc dans le fait que scanf reçoit une information découpée en lignes (nous appelons ainsi une suite de caractères terminée par une fin de ligne). Tant que son traitement n'est pas terminé, elle attend une nouvelle ligne (c'est d'ailleurs ce qui se produisait dans notre premier exemple dans lequel nous commencions par frapper « return »).

Par contre, lorsque son traitement est terminé, s'il existe une partie de ligne non encore utilisée, celle-ci est conservée pour une prochaine lecture. Autrement dit, le tampon n'est pas vidé à chaque nouvel appel de scanf. C'est ce qui explique le comportement du programme précédent.

II.10. En cas d'erreur

Dans le cas de **printf**, la source unique d'erreur résidait dans les fautes de programmation. Dans le cas de **scanf**, en revanche, il peut s'agir, non seulement d'une faute de programmation, mais également d'une mauvaise réponse de l'utilisateur.

II.10.1. Erreurs de programmation

Comme dans le cas de printf, ces erreurs peuvent être de deux types :

- **Code de format en désaccord avec le type de l'expression**
  Si le code de format, bien qu'erroné, correspond à un type de longueur égale à celle de la *lvalue* mentionnée dans la liste, les conséquences se limitent, là encore, à l'introduction d'une mauvaise valeur. Si, en revanche, la *lvalue* a une taille inférieure à celle correspondant au type mentionné dans le code format, il y aura écrasement d'un emplacement mémoire consécutif à cette *lvalue*. Les conséquences en sont difficilement prévisibles.

- **Nombre de codes de format différent du nombre d'éléments de la liste**
  Comme dans le cas de printf, il faut savoir que scanf **cherche toujours à satisfaire le contenu du format**. Les conséquences sont limitées dans le cas

où le format comporte moins de codes que la liste; ainsi, dans cette instruction, on ne cherchera à lire que la valeur de n : scanf ("%d", &n, &p);
En revanche, dans le cas où le format comporte plus de codes que la liste, on cherchera à affecter des valeurs à des emplacements (presque) aléatoires de la mémoire. Là encore, les conséquences en seront pratiquement imprévisibles.

II.10.2. Mauvaise réponse de l'utilisateur

Nous avons déjà vu ce qui se passait lorsque l'utilisateur fournissait trop ou trop peu d'information par rapport à ce qu'attendait scanf.

De même, nous avons vu comment, en cas de rencontre d'un caractère invalide, il y avait arrêt prématuré. Dans ce cas, il faut bien voir que ce caractère non exploité reste dans le tampon pour une prochaine fois. Cela peut conduire à des situations assez cocasses telles que celle qui est présentée dans cet exemple (l'impression de ^C représente, dans l'environnement utilisé, une interruption du programme par l'utilisateur) :

```
main()
{
int n;
do
{ printf ("donnez un nombre : ");
scanf ("%d", &n);
printf ("voici son carré : %d\n", n*n);
}
while (n);
}
```

```
donnez un nombre : 12
voici son carré : 144
donnez un nombre : &
voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
^C
```

Fort heureusement, il existe un remède à cette situation. Nous ne pourrons vous l'exposer complètement que lorsque nous aurons étudié les chaînes de caractères.

II.11. 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.

Exercice IV.1

Quels seront les résultats fournis par ce programme ?

```
#include <stdio.h>
main ()
{ int n = 543;
int p = 5;
float x = 34.5678;
printf ("A : %d %f\n", n, x);
printf ("B : %4d %10f\n", n, x);
printf ("C : %2d %3f\n", n, x);
printf ("D : %10.3f %10.3e\n", x, x);
printf ("E : %*d\n", p, n);
printf ("F : %*.*f\n", 12, 5, x);
}
```

Exercice IV.2

Quelles seront les valeurs lues dans les variables *n* et *p* (de type *int*), par l'instruction suivante ?

```
scanf ("%4d %2d", &n, &p);
```

lorsqu'on lui fournit les données suivantes (le symbole ^ représente un espace et le symbole @ représente une fin de ligne, c'est-à-dire une validation) ?

a) 12^45@
b) 123456@
c) 123456^7@
d) 1^458@
e) ^^^4567^^8912@

SEQUENCE V : LES INSTRUCTIONS DE CONTROLE

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

- de la possibilité d'effectuer des **choix**, de se comporter différemment suivant les circonstances (celles-ci pouvant être, par exemple, une réponse de l'utilisateur, un résultat de calcul…),
- de la possibilité d'effectuer des **boucles**, autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

Tous les langages disposent d'instructions, nommées *instructions de contrôle*, permettant de réaliser ces choix ou ces boucles. Suivant le cas, celles-ci peuvent être:

- basées essentiellement sur la notion de branchement (conditionnel ou inconditionnel); c'était le cas, par exemple, des premiers Basic,
- ou, au contraire, traduire fidèlement les structures fondamentales de la programmation structurée; cela était le cas, par exemple, du langage Pascal bien que, en toute rigueur, ce dernier dispose d'une instruction de branchement inconditionnel GOTO.

Sur ce point, le langage C est quelque peu hybride. En effet d'une part, il dispose d'instructions structurées permettant de réaliser :

- des choix : instructions **if ... else** et **switch**,
- des boucles : instructions **do ... while**, **while** et **for**.

Mais, d'autre part, la notion de branchement n'en est pas totalement absente puisque, comme nous le verrons :

- il dispose d'instructions de branchement inconditionnel : **goto**, **break** et **continue**,
- l'instruction switch est en fait intermédiaire entre un choix multiple parfaitement structuré (comme dans Pascal) et un aiguillage multiple (comme dans Fortran).

Ce sont ces différentes instructions de contrôle du langage C que nous nous proposons d'étudier dans ce chapitre.

I. 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.

I.1. Les 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'en C, la notion d'instruction est en quelque sorte récursive. 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.
D'autre part, un bloc joue le même rôle syntaxique qu'une instruction simple (point-virgule compris). Évitez donc d'ajouter des points-virgules intempestifs à la suite d'un bloc.

I.2. Syntaxe de l'instruction if

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

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

- expression*:* expression quelconque
- instruction_1et instruction_1 : instructions quelconques, c'est-à-dire :
  - simple (terminée par un point-virgule),
  - bloc,
  - instruction structurée.

Remarque :
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.

I.3. Exemples

L'expression conditionnant le choix est quelconque. La richesse de la notion d'expression en C fait que celle-ci peut elle-même réaliser certaines actions. Ainsi :

```
if ( ++i < limite) printf ("OK");  est équivalent à : i = i + 1;
                                                      if ( i < limite ) printf ("OK");
```

Par ailleurs :

```
if ( i++ < limite ) ……      est équivalent à :      i = i + 1;
                                                    if ( i-1 < limite ) ……
```

De même :

```
if ( ( c=getchar() ) != '\n' ) ……      peut remplacer :      c = getchar();
                                                             if ( c != '\n' ) ……
```

En revanche :

```
if (++i<max && ((c=getchar()) != '\n')) ….
```

n'est **pas équivalent à :** ++i;

```
                                    c = getchar();
                                    if ( i<max && ( c!= '\n' ) ) ……
```

car, comme nous l'avons déjà dit, l'opérateur && n'évalue son second opérande que lorsque cela est nécessaire. Autrement dit, dans la première formulation, l'expression : c = getchar() n'est pas évaluée lorsque la condition ++i<max est fausse; elle l'est, en revanche, dans la deuxième formulation.

I.4. Imbrication des instructions if

Nous avons déjà mentionné que les instructions figurant dans chaque partie du choix d'une instruction pouvaient être absolument quelconques. En particulier, elles peuvent, à leur tour, renfermer d'autres instructions if. Or, compte tenu de ce que cette instruction peut comporter ou ne pas comporter de **else**, il existe certaines situations où une ambiguïté apparaît. C'est le cas dans cet exemple :

```
if (a<=b)      if (b<=c)              printf ("ordonné");
               else printf ("non ordonné");
```

Est-il interprété comme le suggère cette présentation ?

```
if (a<=b)    if (b<=c)              printf ("ordonné");
               else printf ("non ordonné");
```

ou bien comme le suggère celle-ci ?

```
if (a<=b) if (b<=c)                 printf ("ordonné");
               else printf ("non ordonné");
```

La première interprétation conduirait à afficher "non ordonné" lorsque la condition a<=b est fausse, tandis que la seconde n'afficherait rien dans ce cas. La règle adoptée par le langage C pour lever une telle ambiguïté est la suivante :

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

Dans notre exemple, c'est la seconde présentation qui suggère le mieux ce qui se passe.

Voici un exemple d'utilisations de if imbriqués. Il s'agit d'un programme de facturation avec remise. Il lit en donnée un simple prix hors taxes et calcule le prix TTC correspondant (avec un taux de TVA constant de 18,6 %). Il établit ensuite une remise dont le taux dépend de la valeur ainsi obtenue, à savoir :

- 0 % pour un montant inférieur à 1 000 F
- 1 % pour un montant supérieur ou égal à 1 000 F et inférieur à 2 000 F
- 3 % pour un montant supérieur ou égal à 2 000 F et inférieur à 5 000 F
- 5 % pour un montant supérieur ou égal à 5 000 F

Ce programme est accompagné de deux exemples d'exécution.

```
#define TAUX_TVA 18.6
main()
{
double ht, ttc, net, tauxr, remise;

printf("donnez le prix hors taxes : ");
scanf ("%lf", &ht);

ttc = ht * ( 1. + TAUX_TVA/100.);

if ( ttc < 1000.)
      tauxr = 0;
else
    if ( ttc < 2000 )
        tauxr = 1.;
    else
         if ( ttc < 5000 )
                tauxr = 3.;
        else
                tauxr = 5.;

remise = ttc * tauxr / 100.;
net = ttc - remise;
printf ("prix ttc %10.2lf\n", ttc);
printf ("remise %10.2lf\n", remise);
printf ("net à payer %10.2lf\n", net);
}
```

```
donnez le prix hors taxes : 500
prix ttc 593.00
remise 0.00
net à payer 593.00
```

```
_____________________________________________
donnez le prix hors taxes : 4000
prix ttc 4744.00
remise 142.32
net à payer 4601.68
```

II. INSTRUCTION SWITCH

II.1. Exemples d'introduction de l'instruction switch

a) Premier exemple

Voyez ce premier exemple de programme accompagné de trois exemples d'exécution.

```
main()
{
int n;
printf ("donnez un entier : ");
scanf ("%d", &n);
switch (n)
{ case 0 : printf ("nul\n");
break;
case 1 : printf ("un\n");
break;
case 2 : printf ("deux\n");
break;
}
printf ("au revoir\n");
}
```

Premier exemple d'instruction switch *(suite)*

```
donnez un entier : 0
nul
au revoir
_____
donnez un entier : 2
deux
au revoir
_____
donnez un entier : 5
au revoir
```

L'instruction switch s'étend ici sur huit lignes (elle commence au mot switch). Son exécution se déroule comme suit. On commence tout d'abord par évaluer l'expression figurant après le mot switch (ici n). Puis, on recherche dans le *bloc* qui suit s'il existe une « *étiquette* » de la forme « *case x* » correspondant à la valeur ainsi obtenue. Si c'est le cas, on se branche à l'instruction figurant après cette étiquette. Dans le cas contraire, on passe à l'instruction qui suit le bloc.

Par exemple, quand n vaut 0, on trouve effectivement une étiquette case 0 et l'on exécute l'instruction correspondante, c'est-à-dire : **printf ("nul");**
On passe ensuite, naturellement, à l'instruction suivante, à savoir, ici : break; Celle-ci demande en fait de sortir du bloc. Notez bien que le rôle de cette instruction est fondamental. Voyez, à titre d'exemple, ce que produirait ce même programme en l'absence d'instructions break :

```
main()
{
int n;
printf ("donnez un entier : ");
scanf ("%d", &n);
switch (n)
{ case 0 : printf ("nul\n");
case 1 : printf ("un\n");
case 2 : printf ("deux\n");
}
printf ("au revoir\n");
}
```

```
donnez un entier : 0
nul
un
deux
au revoir
_____
donnez un entier : 2
deux
au revoir
```

b) Étiquette default

Il est possible d'utiliser le mot-clé **default** comme étiquette à laquelle le programme se branchera dans le cas où aucune valeur satisfaisante n'aura été rencontrée auparavant.

En voici un exemple :

```
main()
{
int n;
printf ("donnez un entier : ");
scanf ("%d", &n);
switch (n)
{ case 0 : printf ("nul\n");
break;
case 1 : printf ("un\n");
break;
case 2 : printf ("deux\n");
break;
```

```
default : printf ("grand\n");
}
printf ("au revoir\n");
}
```

```
donnez un entier : 2
deux
au revoir
_______
donnez un entier : 25
grand
au revoir
```

c) Exemple plus général

D'une manière générale, on peut trouver :

- plusieurs instructions à la suite d'une étiquette,
- des étiquettes sans instructions, c'est-à-dire, en définitive, plusieurs étiquettes successives (accompagnées de leurs deux-points).

Voyez cet exemple, dans lequel nous avons volontairement omis certains break.

```
main()
{
int n;
printf ("Donnez un entier : ");
scanf ("%d", &n);
switch (n)
   {
      case 0 : printf ("Nul\n");
               break;
      case 1 :
      case 2 : printf ("Petit\n");
      case 3 :
      case 4 :
      case 5 : printf ("Moyen\n");
               break;
      default : printf ("Grand\n");
   }
}
```

```
Donnez un entier : 1
Petit
Moyen
_______
Donnez un entier : 4
Moyen
_______
```

```
Donnez un entier : 25
Grand
```

II.2. Syntaxe de l'instruction switch

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

- expression : expression entière quelconque,
- constante : expression constante d'un type entier quelconque (char est accepté car il sera converti en int),
- suite_d'instructions : séquence d'instructions quelconques.

Remarque :
Les crochets ([et]) signifient que ce qu'ils renferment est facultatif.

III. L'INSTRUCTION DO ... WHILE

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

III.1. Exemple d'introduction de l'instruction do ... while

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

```
donnez un nb >0 : -3
vous avez fourni -3
donnez un nb >0 : -9
vous avez fourni –9
donnez un nb >0 : 12
vous avez fourni 12
réponse correcte
```

L'instruction :

```
do
  {
      .......
  }
  while (n<=0);
```

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

On ne sait pas a priori combien de fois une telle boucle sera répétée. Toutefois, de par sa nature même, elle est toujours parcourue au moins une fois. En effet, la condition qui régit cette boucle n'est examinée qu'à la fin de chaque répétition (comme le suggère d'ailleurs le fait que la « partie while » figure en fin).

Notez bien que la sortie de boucle ne se fait qu'après un parcours complet de ses instructions et non dès que la condition mentionnée devient fausse. Ainsi, ici, même après que l'utilisateur a fourni une réponse convenable, il y a exécution de l'instruction d'affichage :

```
printf ("vous avez fourni %d", n);
```

III.2. Syntaxe de l'instruction do... while

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

- 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.

- Lorsque l'instruction à répéter se limite à une seule instruction simple, n'omettez pas le point-virgule qui la termine. Ainsi :

```
do
   c = getchar()
while ( c != 'x');
```

est incorrecte. Il faut absolument écrire :

```
do
  c = getchar();
while ( c != 'x');
```

- N'oubliez pas que, là encore, l'expression suivant le mot while peut être aussi élaborée que vous le souhaitez et qu'elle permet ainsi de réaliser certaines actions. Nous en verrons quelques exemples dans le paragraphe suivant.

- L'instruction à répéter peut être vide (mais quand même terminée par un point-virgule). Ces constructions sont correctes :

```
do;
while (…);

do
{

}
while ( … );
```

- La construction :
```
do
  {

  }
while (1);
```
représente une boucle infinie; elle est syntaxiquement correcte, bien qu'elle ne présente en pratique aucun intérêt. En revanche : do instruction while (1); pourra présenter un intérêt dans la mesure où, comme nous le verrons, il sera possible d'en sortir éventuellement par une instruction break.

- Si vous connaissez Pascal, vous remarquerez que cette instruction do... while correspond au repeat... until avec, cependant, une condition exprimée sous forme contraire.

III.3. Exemples

a) L'exemple proposé au paragraphe 3.1 peut également s'écrire :

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

ou encore :

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

même :

```
do
  {

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

Notez bien que la condition de poursuite doit être la dernière expression évaluée, compte tenu du fonctionnement de l'opérateur séquentiel.

b) L'instruction :

```
do
  {

  }
while ( (c=getchar()) != 'x' );
```

lit des caractères au clavier jusqu'à ce qu'elle ait obtenu le caractère x. Elle est équivalente à :

```
do
        c = getchar();
while ( c != 'x' );
```

IV. L'INSTRUCTION WHILE

Voyons maintenant la deuxième façon de réaliser une boucle conditionnelle, à savoir l'instruction while.

IV.1. Exemple d'introduction de l'instruction *while*

```
main()
{
  int n, som;
  som = 0;

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

printf ("somme obtenue : %d", som);
}
```

```
donnez un nombre : 15
donnez un nombre : 25
donnez un nombre : 12
donnez un nombre : 60
somme obtenue : 112
```

La construction : **while (som<100)** répète l'instruction qui suit (ici un bloc) tant que la condition mentionnée est vraie (différente de zéro), comme le ferait do... while. En revanche, cette fois, la condition de poursuite est examinée **avant** chaque parcours de la boucle et non après. Ainsi, contrairement à ce qui se passait avec do... while, une telle boucle peut très bien n'être parcourue aucune fois si la condition est fausse dès qu'on l'aborde (ce qui n'est pas le cas ici).

IV.2. Syntaxe de l'instruction *while*

while (expression)
    instruction

- 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).
- 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.
- 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. Ainsi, cette construction :
  while (printf ("donnez un nombre : "), scanf ("%d", &n), som<=100)
  som += n;
  n'est pas équivalente à celle de l'exemple d'introduction.
- La construction :
  while (expression1, expression2);
  est équivalente à :
  do expression1
  while (expression2);
  Par exemple, ces deux instructions sont équivalentes :

```
while ((c=getchar()) != 'x' )
  {
  }

do
  {
  }
while ((c=getchar()) != 'x');
```

IV. L'INSTRUCTION FOR

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

IV.1. Exemple d'introduction de l'instruction for

Considérez ce programme :

```
main()
{
   int i;

   for ( i=1; i<=5; i++ )
     {
        printf ("bonjour ");
```

```
        printf ("%d fois\n", i);
    }
}
```

```
bonjour 1 fois
bonjour 2 fois
bonjour 3 fois
bonjour 4 fois
bonjour 5 fois
```

La ligne : for **(i=1; i<=5; i++)** comporte en fait trois expressions. La première est évaluée (une seule fois) avant d'entrer dans la boucle. La deuxième conditionne la poursuite de la boucle. Elle est évaluée **avant** chaque parcours. La troisième, enfin, est évaluée à la fin de chaque parcours.

Le programme précédent est équivalent au suivant :

Remplacement d' une boucle for *par une boucle* while

```
main()
{
   int i;
   i = 1;

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

```
bonjour 1 fois
bonjour 2 fois
bonjour 3 fois
bonjour 4 fois
bonjour 5 fois
```

Là encore, la généralité de la notion d'expression en C fait que ce qui était expression dans la première formulation (for) devient instruction dans la seconde (while).

V.2. Syntaxe de l'instruction for

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

Les crochets [et] signifient que leur contenu est facultatif.

- D'une manière générale, nous pouvons dire que :
  for(expression_1; expression_2; expression_3)

instruction

est équivalent à :

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

- Chacune des trois expressions est facultative. Ainsi, ces constructions sont équivalentes à l'instruction for de notre premier exemple de programme :

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

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

- Lorsque l'expression_2 est absente, elle est considérée comme vraie.
- Là encore, la richesse de la notion d'expression en C permet de grouper plusieurs actions dans une expression. Ainsi :
  for (i=0, j=1, k=5; ...; ...)
  est équivalent à :

```
j=1; k=5;
for ( i=0; ...; ... )
```

ou encore à :

```
i=0; j=1; k=5;
for (; ...; ...)
```

De même :

```
for (i=1; i <= 5; printf("fin de tour"), i++ )
{
        Instructions
 }
```

est équivalent à :

```
for ( i=1; i<=5; i++ )
{
        instructions
        printf ("fin de tour");
}
```

En revanche :

```
for (i=1, printf("on commence"); printf("début de tour"), i<=5; i++)
{
        Instructions
}
```

n'est pas équivalent à :

```
printf ("on commence");
for ( i=1; i<=5; i++ )
  {
        printf ("début de tour");
        instructions
 }
```

car, dans la première construction, le message début de tour est affiché après le dernier tour tandis qu'il ne l'est pas dans la seconde construction.

- Les deux constructions :

```
for (;; );
for (;; )
  {
  }
```

sont syntaxiquement correctes. Elles représentent des boucles infinies de corps vide (n'oubliez pas que, lorsque la seconde expression est absente, elle est considérée comme vraie). En pratique, elles ne présentent aucun intérêt.
En revanche, cette construction :

```
for (;; )
        instruction
```

est une boucle a priori infinie dont on pourra éventuellement sortir par une instruction break (comme nous le verrons dans le paragraphe suivant).

Remarque
Contrairement à ce qui se passe dans beaucoup de langages, les trois instructions de boucle du langage C sont des boucles conditionnelles. En effet, l'instruction for, basée sur une condition, n'est pas l'équivalent strict de la « répétition avec compteur» (ce qui est le cas du for du Pascal et du Basic ou du do du Fortran), même si c'est généralement celle que l'on utilise en C pour jouer un tel rôle.

VI. LES INSTRUCTIONS DE BRANCHEMENT INCONDITIONNEL : BREAK, CONTINUE ET GOTO

Ces trois instructions fournissent des possibilités diverses de branchement inconditionnel. Les deux premières s'emploient principalement au sein de boucles tandis que la dernière est d'un usage libre mais peu répandu, à partir du moment où l'on cherche à structurer quelque peu ses programmes.

VI.1. L'instruction break

Nous avons déjà vu le rôle de break au sein du bloc régi par une instruction switch. Le langage C autorise également l'emploi de cette instruction dans une boucle. Dans ce cas, elle sert à interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. Bien entendu, cette instruction n'a d'intérêt que si son exécution est conditionnée par un choix; dans le cas contraire, en effet, elle serait exécutée dès le premier tour de boucle, ce qui rendrait la boucle inutile.

Voici un exemple montrant le fonctionnement de break :

```
main()
{
   int i;

   for ( i=1; i<=10; i++ )
     {
        printf ("début tour %d\n", i);
        printf ("bonjour\n")

        if ( i==3 )
           break;
        printf ("fin tour %d\n", i);
     }
printf ("après la boucle");
}
```

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

Remarque :
En cas de boucles imbriquées, break fait sortir de la boucle la plus interne. De même si break apparaît dans un switch imbriqué dans une boucle, elle ne fait sortir que du switch.

VI.2. L'instruction continue

L'instruction continue, quant à elle, permet de passer prématurément au tour de boucle suivant. En voici un premier exemple avec for :

```
main()
{
```

```
   int i;

   for ( i=1; i<=5; i++ )
     {
        printf ("début tour %d\n", i);
        if (i<4)
           continue;
        printf ("bonjour\n");
     }
}
```

```
début tour 1
début tour 2
début tour 3
début tour 4
bonjour
début tour 5
bonjour
```

Et voici un second exemple avec do… while :

```
main()
{ int n;
do
   {
        printf ("donnez un nb>0 : ");
        scanf ("%d", &n);
        if (n<0)
           {
              printf ("svp >0\n");
              continue;
           }
       printf ("son carré est : %d\n", n*n);
   }
while(n);
}
```

```
donnez un nb>0 : 4
son carré est : 16
donnez un nb>0 : -5
svp >0
donnez un nb>0 : 2
son carré est : 4
donnez un nb>0 : 0
son carré est : 0
```

Remarque

- Lorsqu'elle est utilisée dans une boucle for, cette instruction continue effectue bien un branchement sur l'évaluation de l'expression de fin de parcours de boucle (nommée expression_2 dans la présentation de sa syntaxe), et non après.
- En cas de boucles imbriquées, l'instruction *continue* ne concerne que la boucle la plus interne.

VI.3. L'instruction goto

Elle permet classiquement le branchement en un emplacement quelconque du programme.

Voyez cet exemple qui simule, dans une boucle for, l'instruction break à l'aide de l'instruction goto (ce programme fournit les mêmes résultats que celui présenté comme exemple de l'instruction break).

```
main()
{
int i;
for (i=1; i<=10; i++)
  {
     printf ("début tour %d\n", i);
     printf ("bonjour\n");
     if (i==3)
          goto sortie;
     printf ("fin tour %d\n", i);
  }
sortie : printf ("après la boucle");
}
```

```
début tour 1
bonjour
fin tour 1
début tour 2
bonjour
fin tour 2
début tour 3
bonjour
après la boucle
```

Exercice V.1

Soit le petit programme suivant :

```
#include <stdio.h>
main()
{
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);
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction for :

- une instruction while,
- une instruction do... while.

Exercice V.2

Calculer la moyenne de notes fournies au clavier avec un dialogue de ce type :
note 1 : 12
note 2 : 15.25
note 3 : 13.5
note 4 : 8.75
note 5 : -1
moyenne de ces 4 notes : 12.37
Le nombre de notes n'est pas connu a priori et l'utilisateur peut en fournir autant qu'il le désire. Pour signaler qu'il a terminé, on convient qu'il fournira une note fictive négative. Celle-ci ne devra naturellement pas être prise en compte dans le calcul de la moyenne.

Exercice V.13

Afficher un triangle rempli d'étoiles, s'étendant sur un nombre de lignes fourni en donnée et se présentant comme dans cet exemple :

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

Exercice V.4

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

Exercice V.5

Écrire un programme qui détermine la *n*-ième valeur un (*n* étant fourni en donnée) de la « suite de Fibonacci » définie comme suit :
u1 = 1
u2 = 1
un = un-1 + un-2 pour n>2

Exercice V.6

Écrire 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

3	I	3	6	9	12	15	12	21	24	27	30
4	I	4	8	12	16	20	24	28	32	36	40
5	I	5	10	15	20	25	30	35	40	45	50
6	I	6	12	18	24	30	36	42	48	54	60
7	I	7	14	21	28	35	42	49	56	63	70
8	I	8	16	24	32	40	48	56	64	72	80
9	I	9	18	27	36	45	54	63	72	81	90
10	I	10	20	30	40	50	60	70	80	90	100

SEQUENCE VI : LES TABLEAUX ET LES CHAINES DE CARACTERES

Dans ce chapitre, nous allons voir comment déclarer un tableau, comment l'initialiser, comment faire référence à un de ses éléments. Du point de vue algorithmique, quand on utilise des tableaux, on a besoin d'instructions itératives.

I. LES TABLEAUX DE NOMBRES (INT ou FLOAT)

Les tableaux correspondent aux matrices en mathématiques. Un tableau est caractérisé par sa taille et par ses éléments.

I.1. Les tableaux à une dimension

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.

I.1.1. Déclaration

Pour déclarer un tableau dont les éléments ont un type de base :

- partir de la déclaration de variable ayant un type de base;
- ajouter entre crochets le nombre d'éléments du tableau après le nom.

type nom[dim];

Cette déclaration signifie que le compilateur réserve dim places en mémoire pour ranger les éléments du tableau.

Exemple
int compteur[10]; le compilateur réserve des places pour 10 entiers.
float nombre[20]; le compilateur réserve des places pour 20 réels.

Remarque
dim est nécessairement une **VALEUR NUMERIQUE**. Ce ne peut être en aucun cas une combinaison des variables du programme. Ainsi, cette construction :

```
#define N 50
.....
int t[N];
float h[2*N-1];
```

est correcte.
En revanche, elle ne le serait pas (en C) 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 (elle sera cependant acceptée en C++).
Les points importants sont les suivants :

- les index des éléments d'un tableau vont de 0 à N - 1;
- la taille d'un tableau doit être connue statiquement par le compilateur.

Impossible donc d'écrire :
int t[n];
où **n** serait une variable.

I.1.2. Initialisation

Il est possible d'initialiser un tableau avec une liste d'expressions constantes séparées par des virgules, et entourée des signes { et }

Exemple 1
```
int liste[10] = {1, 2, 4, 8, 16, 32, 64,128, 256, 528};
float nombre[4] = {2.67, 5.98, -8, 0.09};
```

Exemple 2
```
#define N  5
int t[N] = {1, 2, 3, 4, 5};
```

On peut donner moins d'expressions constantes que le tableau ne comporte d'éléments. Dans ce cas, les premiers éléments du tableau seront initialisés avec les valeurs indiquées, les autres seront initialisés à zéro.

Exemple
```
#define N 10
int t[N] = {1, 2};
```

Les éléments d'indice 0 et 1 seront initialisés respectivement avec les valeurs 1 et 2; les autres éléments seront initialisés à zéro.

Il n'existe malheureusement pas de facteur de répétition, permettant d'exprimer « initialiser n éléments avec la même valeur v ». Il faut soit mettre n fois la valeur v; soit initialiser le tableau par des instructions.

I.1.3. Référence à un élément d'un tableau

Un élément du tableau est repéré par son indice. En langage C les tableaux commencent à l'indice 0. L'indice maximum est donc dim-1.

Dans sa forme la plus simple, une référence à un élément de tableau a la syntaxe suivante :

nom-de-tableau [expression]

«expression» doit être une valeur entière, et peut être la rencontrée aussi bien en partie gauche qu'en partie droite d'affectation.

Exemple 1

```
compteur[2] = 5;
nombre[i] = 6.789;
printf("%d", compteur[i]);
scanf("%f", &nombre[i]);
```

Exemple 2

```
#define N 10
int t[N];
```

on peut écrire :

```
x = t[i];          /* référence à l'élément d'indice i du tableau t */
t[i+j] = k;        /* affectation de l'élément d'indice i+j du tableau t */
```

Exercice

Saisir 10 réels, les ranger dans un tableau. Calculer et afficher la moyenne et l'écart-type.

Exercice

Un programme contient la déclaration suivante :

```
int tab[10] = {4,12,53,19,11,60,24,12,89,19};
```

Compléter ce programme de sorte à afficher les adresses des éléments du tableau.

Exercice

Un programme contient la déclaration suivante:

```
int tab[20] = {4, -2, -23, 4, 34, -67, 8, 9, -10, 11, 4, 12, -53, 19, 11,
                    -60, 24, 12, 89, 19};
```

Compléter ce programme de sorte d'afficher les éléments du tableau avec la présentation suivante :

```
4     -2    -23    4     34
-67   8     9      -10   11
4     12    -53    19    11
-60   24    12     89    19
```

I.2. LES TABLEAUX A PLUSIEURS DIMENSIONS

I.2.1. Les tableaux à deux dimensions

I.2.1.1. Déclaration

```
type nom[dim1][dim2];
```

Exemples

```
int compteur[4][5];
float nombre[2][10];
```

I.2.1.2. Initialisation

Il est possible d'initialiser un tableau avec une liste d'expressions constantes séparées par des virgules, et entourée des signes { et } selon les lignes et les colonnes.

Exemple 1

```
int x[2][3] = { {1,5,7}, {8,4,3} };  /* 2 lignes et 3 colonnes */
```

Exemple 2

Voyez ces deux exemples équivalents (nous avons volontairement choisi des valeurs consécutives pour qu'il soit plus facile de comparer les deux formulations) :

```
int tab [3][4] = {{ 1, 2, 3, 4} ,
                  { 5, 6, 7, 8},
                  { 9,10,11,12 }}

int tab [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

La première forme revient à considérer notre tableau comme formé de trois tableaux de quatre éléments chacun. La seconde, elle, exploite la manière dont les éléments sont effectivement rangés en mémoire et elle se contente d'énumérer les valeurs du tableau suivant cet ordre. Là encore, à chacun des deux niveaux, les dernières valeurs peuvent être omises. Les déclarations suivantes sont correctes (mais non équivalentes) :

```
int tab [3][4] = {{1, 2 } , { 3, 4, 5}};
int tab [3][4] = {1, 2, 3, 4, 5};
```

I.2.1.3. Référence à un élément d'un tableau

Un élément du tableau est repéré par ses indices. En langage C les tableaux commencent aux indices 0. Les indices maximum sont donc dim1-1, dim2-1.

Appel

```
nom[indice1][indice2]
```

Exemple

```
compteur[2][4] = 5;
nombre[i][j] = 6.789;
printf("%d", compteur[i][j]);
scanf("%f", &nombre[i][j]);
```

Les éléments d'un tableau sont rangés suivant l'ordre obtenu en faisant varier le dernier indice en premier. (Pascal utilise le même ordre, Fortran utilise l'ordre opposé).

Nous verrons que cet ordre a une incidence dans au moins trois circonstances :

- lorsque l'on omet de préciser certaines dimensions d'un tableau,
- lorsque l'on souhaite accéder à l'aide d'un pointeur aux différents éléments d'un tableau,
- lorsque l'un des indices « déborde ». Suivant l'indice concerné et les valeurs qu'il prend, il peut y avoir débordement d'indice sans sortie du tableau.

Par exemple, avec un tableau t de 5 x 3 éléments, vous voyez que la notation t[0][5] désigne en fait l'élément t[1][2]. Par contre, la notation t[5][0] désigne un emplacement situé juste au-delà du tableau.

Bien entendu, les différents points évoqués, dans le paragraphe I.1. à propos des tableaux à une dimension, restent valables dans le cas des tableaux à plusieurs dimensions.

Exercice VI.2
Saisir une matrice d'entiers 2x2, calculer et afficher son déterminant.

I.2.2. Les tableaux à plus de deux dimensions

On procède de la même façon en ajoutant les éléments de dimensionnement ou les indices nécessaires.

IV. LES CHAINES DE CARACTERES

Certains langages (Java, Basic, anciennement Turbo Pascal) disposent d'un véritable type chaîne. Les variables d'un tel type sont destinées à recevoir des suites de caractères qui peuvent évoluer, à la fois en contenu et en longueur, au fil du déroulement du programme. Elles peuvent être manipulées d'une manière globale, en ce sens qu'une simple affectation permet de transférer le contenu d'une variable de ce type dans une autre variable de même type.

D'autres langages (Fortran, Pascal standard) ne disposent pas d'un tel type chaîne. Pour traiter de telles informations, il est alors nécessaire de travailler sur des tableaux de caractères dont la taille est nécessairement fixe (ce qui impose à la fois une longueur maximale aux chaînes et ce qui, du même coup, entraîne une perte de place mémoire). La manipulation de telles informations est obligatoirement réalisée caractère par caractère et il faut, de plus, prévoir le moyen de connaître la longueur courante de chaque chaîne.

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

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

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

V.1. Déclaration

char nom[dim];	ou bien	char *nom; nom = (char*)malloc(dim);

Exemple

char texte[dim];	ou bien	char *texte; texte = (char*)malloc(10);

Le compilateur réserve (dim-1) places en mémoire pour la chaîne de caractères. En effet, il ajoute toujours le caractère NUL ('\0') à la fin de la chaîne en mémoire.

V.2. 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 revanche, C vous autorise à initialiser votre tableau de caractères à l'aide d'une chaîne constante. Ainsi, vous pourrez écrire :
char ch[20] = "bonjour";

Cela sera parfaitement équivalent à une initialisation de ch réalisée par une énumération debcaractères (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 0 de fin).

V.3. 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" };
```

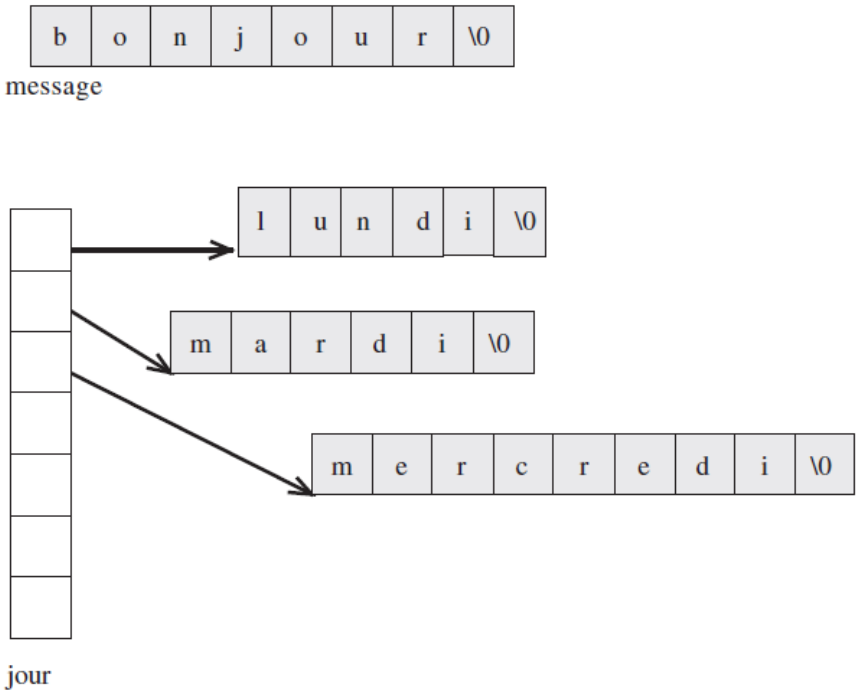
Cette déclaration réalise donc à la fois la création des 7 chaînes constantes correspondant aux 7 jours de la semaine et l'initialisation du tableau jour avec les 7 adresses de ces 7 chaînes.
Voici un exemple employant cette déclaration (nous y avons fait appel, pour l'affichage d'une chaîne, au code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

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

printf ("Donnez un entier entre 1 et 7 : ");
scanf ("%d", &i);
printf ("Le jour numéro %d de la semaine est %s", i, jour[i-1] );
}
```

```
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 (comme le faisait adr dans le paragraphe 1.1). Le schéma ci- après récapitule les deux situations.



I.4. Lecture et écriture des chaînes

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

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

Voyez cet exemple de programme :

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

```
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. En revanche, 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.

Remarque

- 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. La norme prévoit toutefois que si l'on applique l'opérateur & à un nom de tableau, on obtient l'adresse du tableau. Autrement dit, &nom est équivalent à nom.

- 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
- 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.
- Nous nous sommes limité ici aux entrées-sorties conversationnelles. Les autres possibilités seront examinées dans le chapitre consacré aux fichiers.
- 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...
- Etant donné 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 gets) : 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.

I.5. Fiabilisation de la lecture au clavier : le couple gets sscanf

Nous avons vu, dans le chapitre concernant les entrées-sorties conversationnelles, les problèmes posés par **scanf** en cas de réponse incorrecte de la part de l'utilisateur. Il est possible de régler la plupart de ces problèmes en travaillant en deux temps :

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

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

```
#include <stdio.h>
#define LG 80
main()
{
int n, compte;
char c;
char ligne [LG+1];
```

```
do
{ printf ("donnez un entier et un caractère : ");
gets (ligne);
compte = sscanf (ligne, "%d %c", &n, &c);
}
while (compte<2 );
printf ("merci pour %d %c\n", n, c);
}
```

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

Remarque
Nous avons prévu ici des lignes de 80 caractères au maximum. Nous risquons donc de voir le tableau ligne « déborder » si l'utilisateur fournit une réponse plus longue. Dans la pratique, on peut augmenter la valeur de LG, notamment lorsque, comme c'est souvent le cas, on a affaire à une implémentation où les lignes frappées au clavier ont une taille maximale. Si l'on cherche à réaliser un programme« portable », on préférera une solution qui consiste à remplacer gets par fgets (stdin,...) dont nous parlerons dans le chapitre consacré aux fichiers. La démarche restera identique à celle présentée ici.

Exercice VI.6
Saisir une chaîne de caractères, afficher les éléments de la chaîne et leur adresse (y compris le dernier caractère '\0').

Exercice VI.6
Saisir une chaîne de caractères. Afficher le nombre de e et d'espaces de cette chaîne.

Fonctions permettant la manipulation des chaînes

Les bibliothèques fournies avec les compilateurs contiennent de nombreuses fonctions de traitement des chaînes de caractères. En BORLAND C++, elles appartiennent aux bibliothèques string.h ou stdlib.h.

En voici quelques exemples

- Générales (string.h):

void *strcat(char *chaine1, char *chaine2)	concatène les 2 chaînes, résultat dans chaine1, renvoie l'adresse de chaine1.
int strlen(char *chaine)	renvoie la longueur de la chaine ('\0' non comptabilisé).
void *strrev(char *chaine)	inverse la chaîne et, renvoie l'adresse de la chaine inversée.

- Comparaison (string.h):

int strcmp(char *chaine1,char *chaine2)	renvoie un nombre: - positif si la chaîne1 est supérieure à la chaine2 (au sens de l'ordre alphabétique) - négatif si la chaîne1 est inférieure à la chaîne2 - nul si les chaînes sont identiques.

- Copie (string.h):

void *strcpy(char *chaine1, char *chaine2)	recopie chaine2 dans chaine1 et renvoie l'adresse de chaîne1

- **Recopie (string.h):**

Ces fonctions renvoient l'adresse de l'information recherchée en cas de succès, sinon le pointeur NULL (c'est à dire le pointeur de valeur 0 ou encore le pointeur faux).

void *strchr(chaine,caractère)	recherche le caractère dans la chaîne.
void *strrchr(chaine,caractère)	idem en commençant par la fin.
void *strstr(chaîne,sous-chaîne)	recherche la sous-chaine dans la chaîne.

- **Conversions (stdlib.h)**

int atoi(char *chaîne)	convertit la chaîne en entier

float atof(char *chaine)	convertit la chaîne en réel
	Exemple
	printf("ENTRER UN TEXTE: ");
	gets(texte);
	n = atoi(texte);
	printf("%d", n); /* affiche 123 si texte vaut "123" */
	/* affiche 0 si texte vaut "bonjour" */

void *itoa(int n, char *chaîne, int base)	convertit un entier en chaîne: base: base dans laquelle est exprimé le nombre, cette fonction renvoie l'adresse de la chaîne. **Exemple** itoa(12,texte,10); texte vaut "12"

Pour tous ces exemples, la notation void* signifie que la fonction renvoie un pointeur (l'adresse de l'information recherchée), mais que ce pointeur n'est pas typé. On peut ensuite le typer à l'aide de l'opérateur cast.

Exemple

```
int *adr;
char texte[10] = "BONJOUR";
adr = (int*)strchr(texte, 'O');
```

Exercice VI-7

L'utilisateur saisit le nom d'un fichier. Le programme vérifie que celui-ci possède l'extension .PAS

SEQUENCE VII : 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 le C autorise effectivement la compilation séparée de tels modules.

I. LA FONCTION : LA SEULE SORTE DE MODULE EXISTANT EN C

Dans certains langages, on trouve deux sortes de modules, à savoir : les **fonctions** et les **procédures**.

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

En C, il n'existe qu'une seule sorte de module, nommé **fonction** (il en ira de même en C++ et en Java, langage dont la syntaxe est proche de celle de C). 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 réservée 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.

Par ailleurs, nous verrons qu'en C plusieurs fonctions peuvent partager des informations, autrement que par passage d'arguments. Nous retrouverons la notion classique de « variables globales ».

Enfin, l'un des atouts du langage C réside dans la possibilité de **compilation séparée**. Celle-ci permet de découper le programme source en plusieurs parties, chacune de ces parties pouvant comporter une ou plusieurs fonctions. Certains auteurs emploient parfois le mot « module » pour désigner chacune de ces parties (stockées dans un fichier); dans ce cas, ce terme de module devient synonyme de fichier source. Cela facilite considérablement le développement et la mise au point de grosses applications. Cette possibilité crée naturellement quelques contraintes supplémentaires, notamment au niveau des variables globales que l'on souhaite partager entre différentes parties du programme source (c'est d'ailleurs ce qui justifiera l'existence de la déclaration **extern**).

Pour garder une certaine progressivité dans notre exposé, nous supposerons tout d'abord que nous avons affaireà un programme source d'un seul tenant (ce qui ne nécessite donc pas de compilation séparée). Nous présenterons ainsi la structure générale d'une fonction, les notions d'arguments, de variables globales et locales.

II. EXEMPLE DE DEFINITION ET D'UTILISATION D'UNE FONCTION

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.

```
#include <stdio.h>
/***** le programme principal (fonction main) *****/
main()
```

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

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

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

/*************** la fonction fexple ***************/
float fexple (float x, int b, int c)
{
  float val; /* déclaration d'une variable "locale" à fexple
  val = x * x + b * x + c;
  return val;
}
```

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

float fexple (float x, int b, int c)

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

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

Si on s'intéresse au corps de la fonction, on y rencontre tout d'abord une déclaration:
float val;
Celle-ci précise que, pour effectuer son travail, notre fonction a besoin d'une variable de type **float** nommée **val**. *On dit que val est une variable locale à la fonction fexple*, de même que les variables telles que n, p, y ... sont des variables locales à la fonction main (mais comme jusqu'ici nous avions affaireà 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 **x2 + bx + c**. Notez bien l'aspect arbitraire du nom des arguments; on obtiendrait la même définition de fonction avec, par exemple:

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

Examinons maintenant la fonction **main**. Vous constatez qu'on y trouve une déclaration : **float fexple (float, int, int);**

Elle sert à prévenir le compilateur que **fexple** est une fonction et elle lui précise le type de ses arguments ainsi que celui de sa valeur de retour. Nous reviendrons plus loin en détail sur le rôle d'une telle déclaration.

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

III. FONCTIONS ET LEURS DECLARATIONS

III.1. Différentes façons de déclarer (ou de ne pas déclarer) une fonction

Dans notre exemple du paragraphe 2, nous avions fourni la définition de la fonction **fexple après** celle de la fonction main. Mais nous aurions pu tout aussi bien faire l'inverse :

```
float fexple (float x, int b, int c)
{
....
}

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

En toute rigueur, dans ce cas, la déclaration de la fonction **fexple (ici, dans main) est facultative**, car, lorsqu'il traduit la fonction main, le compilateur connaît déjà la fonction fexple. Néanmoins, nous vous déconseillons d'omettre la déclaration de fexple dans ce cas. En effet, il est tout à fait possible qu'ultérieurement vous soyez amené à modifier votre programme source ou même à l'éclater en plusieurs fichiers source comme l'autorisent les possibilités de compilation séparée du langage C.

Par ailleurs, le langage C (mais pas le C++) vous permet d'effectuer des déclarations partielles en ne mentionnant pas le type des arguments; ainsi, dans notre exemple du paragraphe 2, nous pourrions déclarer fexple de cette façon dans la fonction main:
float fexple ();

Qui plus est, C vous autorise à ne pas déclarer du tout une fonction qui renvoie une valeur de
type int (là encore, ce sera interdit en C++ ainsi qu'en C99).
Nous ne saurions trop vous conseiller d'éviter de telles possibilités. Toutefois, sachez que vous risquez d'employer la dernière sans y prendre garde. En effet, toute fonction que vous utiliserez sans l'avoir déclarée sera considérée par le compilateur comme ayant des arguments quelconques et fournissant un résultat de type int. Les conséquences en seront différentes suivant que ladite fonction est ou non fournie dans le même fichier source. Dans le premier cas, on obtiendra bien une erreur de compilation; dans le second, en revanche, les conséquences n'apparaîtront (de manière plus ou moins voilée) que lors de l'exécution !

La déclaration complète d'une fonction porte le nom de **prototype**. Il est possible, dans un prototype, de faire figurer des noms d'arguments, lesquels sont alors totalement arbitraires; cette possibilité a pour seul intérêt de pouvoir écrire des prototypes qui sont identiques à l'entête de la fonction (au point-virgule près), ce qui peut en faciliter la création automatique.

Dans notre exemple du paragraphe 2, notre fonction **fexple** aurait pu être **déclarée** ainsi :
float fexple (float x, int b, int c);

III.2. Où placer la déclaration d'une fonction

La tendance la plus naturelle consiste à placer la déclaration d'une fonction à l'intérieur des déclarations de toute fonction l'utilisant; c'est ce que nous avons fait jusqu'ici. Et, de surcroît, dans tous nos exemples précédents, la fonction utilisatrice était la fonction main elle-même !

Dans ces conditions, nous avions affaire à une déclaration locale dont la portée était limitée à la fonction où elle apparaissait.
Mais il est également possible d'utiliser des déclarations globales, en les faisant apparaître **avant la définition de la première fonction**. Par exemple, avec :

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

```
void f1 (…)
{ …..
}
```

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

III.3. À quoi sert la déclaration d'une fonction

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

- Si la définition de la fonction se trouve dans le même fichier source (que ce soit avant ou après la déclaration), il s'assure que les arguments muets ont bien le type défini dans le prototype. Dans le cas contraire, il signale une erreur.
- Lorsqu'il rencontre un appel de la fonction, il met en place d'éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype. Par exemple, avec notre fonction fexple, un appel tel que :
  fexple (n+1, 2*x, p)
  sera traduit par :
  - l'évaluation de la valeur de l'expression n+1 (en int) et sa conversion en float,
  - l'évaluation de la valeur de l'expression 2*x (en float) et sa conversion en int; il y a donc dans ce dernier cas une conversion dégradante.

Remarque

- Rappelons que, lorsque le compilateur ne connaît pas le type des arguments d'une fonction, il utilise des règles de conversions systématiques : char *et* short -> int et float -> double.
  La fonction printf est précisément dans ce cas.
- Compte tenu de la remarque précédente, seule une fonction déclarée avec un prototype pourra recevoir un argument de type **float**, **char** ou **short**.

IV. Retour sur les fichiers en-tête

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

Ce point se révèle fort utile :

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

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

```
float x, y;
…..
```

```
y = sqrt (x);
…..
```

sans les faire précéder d'une quelconque directive #include.

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

On se trouve en présence des conséquences habituelles d'une mauvaise interprétation de type.

Un premier remède consiste à placer dans votre module la déclaration :
double sqrt(double);

mais encore faut-il que vous connaissiez de façon certaine le type de cette fonction.

Une meilleure solution consiste à placer, en début de votre programme, la directive :
#include <math.h>

laquelle incorporera automatiquement le prototype approprié (entre autres choses).

V. QUELQUES REGLES

V.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 » (de l'anglais : *formal parameter)*. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.

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

V.2. L'instruction return

Voici quelques règles générales concernant cette instruction :

- L'instruction **return** peut mentionner n'importe quelle expression. Ainsi, nous aurions pu définir la fonction fexple précédente d'une manière plus simple :
  ```
  float fexple (float x, int b, int c)
  {
  ```

```
    return (x * x + b * x + c);
}
```

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

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

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

- Si le type de l'expression figurant dans return est différent du type du résultat tel qu'il a été déclaré dans l'en-tête, le compilateur mettra automatiquement en place des instructions de conversion.
  Il est toujours possible de ne pas utiliser le résultat d'une fonction, même si elle en produit un. C'est d'ailleurs ce que nous avons fait fréquemment avec printf ou scanf.
  Bien entendu, cela n'a d'intérêt que si la fonction fait autre chose que de 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 !).

V.3. Cas des fonctions sans valeur de retour ou sans arguments

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

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

et voici quelle serait sa déclaration :

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

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction return. Certains compilateurs ne détecteront toutefois pas l'erreur. Quand une fonction ne reçoit aucun argument, on place le mot-clé **void** (le même que précédemment, mais avec une signification différente!) à la place de la liste d'arguments (attention, en C++, la règle sera différente : on se contentera de ne rien mentionner dans la liste d'arguments).

Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type float (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire!) :
float tirage (void)

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

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

et sa déclaration sera :
void message (void);

En toute rigueur, la fonction main est une fonction sans argument et sans valeur de retour. Elle devrait donc avoir pour en-tête « void main (void) ». Certains compilateurs fournissent d'ailleurs un message d'avertissement (« warning ») lorsque vous vous contentez de l'en-tête usuel main.

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

```
#include <stdio.h>
main()
{
void affiche_carres(int, int);        /* prototype de affiche_carres */
void erreur(void);                    /* prototype de erreur */
int debut = 5, fin = 10;
.....
affiche_carres(debut, fin);
.....
if (...) erreur ();
}

void affiche_carres(int d, int f)
{
int i;
 for(i=d; i<=f; i++)
  {
     printf ("%d a pour carré %d\n", i, i*i);
  }
void erreur (void)
{
printf ("*** erreur ***\n");
```

```
}
```

V.4. Les anciennes formes de l'en-tête des fonctions

Dans la première version du langage C, telle qu'elle a été définie par Kernighan et Ritchie, avant la normalisation par le comité ANSI, l'en-tête d'une fonction s'écrivait différemment de ce que nous avons vu ici. Par exemple, l'en-tête de notre fonction fexple aurait été :

```
float fexple (x, b, c)
float x;
int b, c;
```

La norme ANSI autorise les deux formes, lesquelles sont actuellement acceptées par la plupart des compilateurs. Toutefois, seule la forme moderne, c'est-à-dire celle que nous avons présentée précédemment, sera autorisée par C++.

Remarque :
L'habitude veut que les en-têtes écrits sous l'ancienne forme le soient sur plusieurs lignes comme dans notre exemple. Mais rien ne nous empêcherait de l'écrire sous cette forme :

```
float fexple (x, b, c) float x; int b, c;
```

VI. EN C, LES ARGUMENTS SONT TRANSMIS PAR VALEUR

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

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

```
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*.
- La norme n'impose aucun ordre pour l'évaluation des différents arguments d'une fonction lorsde son appel. En général, ceci est de peu d'importance, excepté dans une situation (fortement déconseillée !) telle que :

```
int i = 10;
        ...
f (i++, i);      /* on peut calculer i++ avant i --> f (10, 11) */
                          /* ou après i --> f (10, 10) */
```

VII. LES VARIABLES GLOBALES

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

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

Remarque :
La norme ANSI ne parle pas de variables globales, mais de variables externes. Le terme « global » illustre plutôt le partage entre plusieurs fonctions tandis que le terme « externe » illustre plutôt le partage entre plusieurs fichiers source. En C, une variable globale est partagée par plusieurs fonctions; elle peut être (mais elle n'est pas obligatoirement) partagée entre plusieurs fichiers source.

VII.1. Exemple d'utilisation de variables globales

Voyez cet exemple de programme.

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

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

La variable i a été déclarée en dehors de la fonction main. Elle est alors connue de toutes les fonctions qui seront compilées par la suite au sein du même programme source. Ainsi, ici, le programme principal affecte à i des valeurs qui se trouvent utilisées par la fonction **optimist**.

Notez qu'ici la fonction **optmist** se contente d'utiliser la valeur de i mais rien ne l'empêche de la modifier. C'est précisément ce genre de remarque qui doit vous inciter à n'utiliser les variables globales que dans des cas limités. En effet, toute variable globale peut être modifiée insidieusement par n'importe quelle fonction. Lorsque vous aurez à écrire des fonctions susceptibles de modifier la valeur de certaines variables, il sera beaucoup plus judicieux de prévoir d'en transmettre l'adresse en argument (comme vous apprendrez à le faire dans le prochain chapitre). En effet, dans ce cas, l'appel de la fonction montrera explicitement quelle est la variable qui risque d'être modifiée et, de plus, ce sera la seule qui pourra l'être.

VII.2. La portée des variables globales

Les variables globales ne sont connues du compilateur que dans la partie du programme source suivant leur déclaration. On dit que leur **portée** (ou encore leur **espace de validité**) est limitée à la partie du programme source qui suit leur

déclaration (n'oubliez pas que, pour l'instant, nous nous limitons au cas où l'ensemble du programme est compilé en une seule fois).

Ainsi, voyez, par exemple, ces instructions :

```
main()
{
  ....
}

int n;
float x;

fct1 (...)
{
  ....
}

fct2 (...)
{
....
}
```

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

VII.3. La classe d'allocation des variables globales

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

VIII. LES VARIABLES LOCALES

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

VIII.1. La portée des variables locales

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

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

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

VIII.2. Les variables locales automatiques

Par défaut, les variables locales ont une durée de vie limitée à celle d'**une exécution** de la fonction dans laquelle elles figurent. Plus précisément, leurs emplacements ne sont pas définis de manière permanente comme ceux des variables globales. Un nouvel espace mémoire leur est alloué à chaque entrée dans la fonction et libéré à chaque sortie. Il sera donc généralement différent d'un appel au suivant. On traduit cela en disant que la **classe d'allocation** de ces variables est **automatique**. Nous aurons l'occasion de revenir plus en détail sur cette gestion dynamique de la mémoire. Pour l'instant, il est important de noter que la conséquence immédiate de ce mode d'allocation est que les valeurs des variables locales ne sont pas conservées d'un appel au suivant (on dit aussi qu'elles ne sont pas « rémanentes »). Nous reviendrons un peu plus loin (paragraphe 11.2) sur les éventuelles initialisations de telles variables.

D'autre part, les valeurs transmises en arguments à une fonction sont traitées de la même manière que les variables locales. Leur durée de vie correspond également à celle de la fonction.

VIII.3. Les variables locales statiques

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

En voici un exemple :

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

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

La variable locale i a été déclarée de classe « statique ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro**.

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée i qui n'aurait alors aucun rapport avec la variable i de fct.

VIII.4. Le cas des fonctions récursives

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

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

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

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

```
else return(1);
}
```

Il faut bien voir qu'alors chaque appel de fac entraîne une allocation d'espace pour les variables locales et pour son argument n (apparemment, fct ne comporte aucune variable locale; en réalité, il lui faut prévoir un emplacement destiné à recevoir sa valeur de retour). Or chaque nouvel appel de **fac**, à l'intérieur de fac, provoque une telle allocation, sans que les emplacements précédents soient libérés.

Il y a donc un empilement des espaces alloués aux variables locales, parallèlement à un empilement des appels de la fonction. Ce n'est que lors de l'exécution de la première instruction return que l'on commencera à « dépiler » les appels et les emplacements et donc à libérer de l'espace mémoire.

IX. LA COMPILATION SEPAREE ET SES CONSEQUENCES

Si le langage C est effectivement un langage que l'on peut qualifier d'opérationnel, c'est en partie grâce à ses possibilités dites de **compilation séparée**. En C, en effet, il est possible de compiler séparément plusieurs programmes (fichiers) source et de rassembler les modules objet correspondants au moment de l'édition de liens. D'ailleurs, dans certains environnements de programmation, la notion de *projet* permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable.
Cette notion de projet fait intervenir précisément les fichiers à considérer; généralement, il est possible de demander de créer le programme exécutable, en ne recompilant que les sources ayant subi une modification depuis leur dernière compilation.

Indépendamment de ces aspects techniques liés à l'environnement de programmation considéré, les possibilités de compilation séparée ont une incidence importante au niveau de la portée des variables globales. C'est cet aspect que nous nous proposons d'étudier maintenant. Dans le paragraphe suivant, nous serons alors en mesure de faire le point sur les différentes classes d'allocation des variables.

Notez que, à partir du moment où l'on parle de compilation séparée, il existe au moins (ou il a existé) deux programmes source; dans la suite, nous supposerons qu'ils figurent dans des fichiers, de sorte que nous parlerons toujours de fichier source.

Pour l'instant, voyons l'incidence de cette compilation séparée sur la portée des variables globales.

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

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

source 1	source 2
int x;	fct2()

main()
{
.....
}
fct1()
{
.....
}
	{
	.....
	}
	fct3()
	{
	.....
	}

Il ne semble pas possible, dans les fonctions fct2 et fct3 de faire référence à la variable globale x déclarée dans le premier fichier source (alors qu'aucun problème ne se poserait si l'on réunissait ces deux fichiers source en un seul, du moins si l'on prenait soin de placer les instructions du second fichier à la suite de celles du premier).

En fait, le langage C prévoit une déclaration permettant de spécifier qu'une variable globale a déjà été définie dans un autre fichier source. Celle-ci se fait à l'aide du mot-clé extern. Ainsi, en faisant précéder notre second fichier source de la déclaration :
extern int x;

il devient possible de mentionner la variable globale x (déclarée dans le premier fichier
source) dans les fonctions fct2 et fct3.

Remarque

- Cette déclaration extern n'effectue **pas de réservation d'emplacement de variable**. Elle ne fait que préciser que la variable globale x est définie par ailleurs et elle en précise le type.
- Nous n'avons considéré ici que la façon la plus usuelle de gérer des variables globales (celle-ci est utilisable avec tous les compilateurs, qu'ils soient d'avant ou d'après la norme). La norme prévoit d'autres possibilités, au demeurant fort peu répandues et, de surcroît, peu logiques (doubles déclarations, mot extern utilisé même pour la réservation d'un emplacement, définitions potentielles).

IX.2. Les variables globales et l'édition de liens

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

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

D'autre part, après compilation du second fichier source, on trouve dans le module objet correspondant, une indication mentionnant qu'une certaine variable de nom x provient de l'extérieur et qu'il faudra en fournir l'adresse effective.
Ce sera effectivement le rôle de l'éditeur de liens que de retrouver dans le premier module objet l'adresse effective de la variable x et de la reporter dans le second module objet.
Ce mécanisme montre que s'il est possible, par mégarde, de réserver des variables globales de même nom dans deux fichiers source différents, il sera, par contre, en général, impossible d'effectuer correctement l'édition de liens des modules objet correspondants (certains éditeurs de liens peuvent ne pas détecter cette anomalie). En effet, dans un tel cas, l'éditeur de liens se trouvera en présence de deux adresses différentes pour un même identificateur, ce qui est illogique.

IX.3. Les variables globales cachées - la déclaration static

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

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

fct()
{
  .....
}
```

Sans la déclaration **static**, a serait une variable globale ordinaire. Par contre, cette déclaration demande qu'aucune trace de a ne subsiste dans le module objet résultant de ce fichier source. Il sera donc impossible d'y faire référence depuis une autre source par **extern**.

Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source. Cette possibilité de cacher des variables globales peut s'avérer précieuse lorsque vous êtes amené à développer un ensemble de fonctions d'intérêt général qui doivent partager des variables globales. En effet, elle permet à l'utilisateur éventuel de ces fonctions de ne pas avoir à se soucier des noms de ces variables globales puisqu'il ne risque pas alors d'y accéder par mégarde. Cela généralise en quelque sorte à tout un fichier source la notion de variable locale telle qu'elle était définie pour les fonctions. Ce sont d'ailleurs de telles possibilités qui permettent de développer des logiciels en C, en utilisant une philosophie orientée objet.

Exercice VIII.1

Écrire :

- une fonction, nommée f1, se contentant d'afficher "bonjour" (elle ne possèdera 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.

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

Exercice VIII.2

Qu'affiche le programme suivant ?

```
int n=5;
main()
{
void fct (int p);
int n=3;
fct(n);
}
void fct(int p)
{
printf("%d %d", n, p);
}
```

Exercice VIII.3

Écrire 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.

ICI

SEQUENCE VII : LES STRUCTURES ET LES ENUMERATIONS

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.

Quant au type énumération, il s'agit d'un cas particulier de type entier. Sa présentation (tardive) dans ce chapitre ne se justifie que parce que sa déclaration et son utilisation sont très proches de celles du type structure.

I. LES STRUCTURES

I.1. La déclaration d'une structure

Voyez tout d'abord cette déclaration :

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

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

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

Par exemple :
struct enreg art1;

réserve un emplacement nommé art1 « de type enreg » destiné à contenir deux entiers et un flottant.

De manière semblable :
struct enreg art1, art2;

réserve deux emplacements art1 et art2 du type enreg.

Remarque :
Bien que ce soit peu recommandé, sachez qu'il est possible de regrouper la définition du modèle de structure et la déclaration du type des variables dans une seule instruction comme dans cet exemple :

```
struct enreg
  {
      int numero;
      int qte;
      float prix;
  }
art1, art2;
```

Dans ce dernier cas, il est même possible d'omettre le nom de modèle (enreg), à condition, à condition que l'on n'ait pas à déclarer par la suite des variables de ce type.

I.2. L'utilisation d'une structure

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

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

I.2.1. L'utilisation des champs d'une structure

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

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

```
art1.numero = 15;          // affecte la valeur 15 au champ numero de la
                           // structure art1.
printf ("%e", art1.prix);  // affiche, suivant le code format %e, la valeur
                           // du champ prix de la structure art1.
scanf ("%e", &art2.prix);  // lit, suivant le code format %e, une valeur qui
                           // sera affectée au champ prix de la structure
                           // art2. Notez bien la présence de l'opérateur &.
art1.numero++              // incrémente de 1 la valeur du champ numero
                           // de la structure art1.
```

Remarque
La priorité de l'opérateur «.» est très élevée, de sorte qu'aucune des expressions ci-dessus ne nécessite de parenthèses (voyez le tableau du chapitre 3, « Les opérateurs et les expressions en langage C »).

I.2.2. L'utilisation globale d'une structure

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

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

I.3. L'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 (on retrouve là les mêmes restrictions que pour les tableaux).

Voici un exemple d'initialisation de notre structure art1, au moment de sa déclaration :
struct enreg art1 = {100, 285, 2000};

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

I.4. 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.

I.4.1. 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;

De même :
typedef int * ptr;

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

Notez bien que cette déclaration est plus puissante qu'une substitution telle qu'elle pourrait être réalisée par la directive #define. Nous n'en ferons pas ici de description exhaustive, et cela d'autant plus que son usage tend à disparaître. À titre indicatif, sachez, par exemple, qu'avec la déclaration :

typedef int vecteur[3];

les déclarations suivantes sont équivalentes :

int v[3], w[3]; vecteur v, w;

I.4.2. Application aux structures

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

```
struct enreg
  {
      int numero;
      int qte;
      float prix;
   };
typedef struct enreg s_enreg;
s_enreg art1, art2;
```

ou encore, plus simplement :

```
typedef struct
  {
      int numero;
      int qte;
      float prix;
  } s_enreg;
s_enreg art1, art2;
```

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.

I.5. 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... Il peut même s'agir de pointeurs sur des structures du type de la structure dans laquelle ils apparaissent.

De même, un tableau peut être constitué d'éléments qui sont eux-mêmes des structures. Voyons ici quelques situations classiques.

I.5.1. Structure comportant des tableaux

Soit la déclaration suivante :

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

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

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

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

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

La notation :
employe.heures[4]

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

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

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

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

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

À titre indicatif, voici un exemple d'initialisation d'une structure de type personne lors de sa déclaration :
struct personne emp = {"Dupont", "Jules", {8, 7, 8, 6, 8, 0, 0, 8}};

I.5.2. Les tableaux de structures

Voyez ces déclarations :
```
struct point
{
        char nom;
        int x;
        int y;
};
struct point courbe [50];
```

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

Notez bien que point est un nom de modèle de structure, tandis que courbe représente effectivement un tableau de 50 éléments du type point.

Si i est un entier, la notation : **courbe[i].nom** représente le nom du point de rang i du tableau courbe. Il s'agit donc d'une valeur de type **char**.

Notez bien que la notation : courbe.nom[i] n'aurait pas de sens.
De même, la notation : **courbe[i].x** désigne la valeur du champ x de l'élément de rang i du tableau courbe.

Par ailleurs : **courbe[4]** représente la structure de type point correspondant au cinquième élément du tableau courbe.

Enfin **courbe** est un identificateur de tableau, et, comme tel, désigne son adresse de début.

Là encore, voici, à titre indicatif, un exemple d'initialisation (partielle) de notre variable courbe, lors de sa déclaration :
struct point courbe[50]= {{'A', 10, 25}, {'M', 12, 28},, {'P', 18,2} };

I.5.3. Structures comportant d'autres structures

Supposez que, à l'intérieur de nos structures **employe** et **courant** définies dans le paragraphe I.5.1, nous ayons besoin d'introduire deux dates : la date d'embauche et la date d'entrée dans le dernier poste occupé. Si ces dates sont elles-mêmes des structures comportant trois champs correspondant au jour, au mois et à l'année, nous pouvons alors procéder aux déclarations suivantes :

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

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

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

La notation :
employe.date_embauche.annee
représente l'année d'embauche correspondant à la structure employe. Il s'agit d'une valeur de type int.

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

I.6. À propos de la portée du modèle de structure

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

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

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

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

main()
{
struct enreg x;
....
}

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

En revanche, il n'est pas possible, dans un fichier source donné, de faire référence à un modèle défini dans un autre fichier source. Notez bien qu'il ne faut pas assimiler le nom de modèle d'une structure à un nom de variable; notamment, il n'est pas possible, dans ce cas, d'utiliser de déclaration **extern**. En effet, la déclaration **extern** s'applique à des identificateurs susceptibles d'être remplacés par des adresses au niveau de l'édition de liens. Or un modèle de structure représente beaucoup plus qu'une simple information d'adresse et il n'a de signification qu'au moment de la compilation du fichier source où il se trouve.

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.

Le même problème de portée se pose pour les synonymes définis par typedef. Les mêmes solutions peuvent y être apportées par l'emploi de #include.

I.7. Transmission d'une structure en argument d'une fonction

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

I.7.1. 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;
        float b;
};

main()
{
struct enreg x;

void fct (struct enreg y);
x.a = 1; x.b = 12.5;
printf ("\navant appel fct : %d %e",x.a,x.b);
fct (x);
printf ("\nau retour dans main : %d %e", x.a, x.b);
}

void fct (struct enreg s)
{
s.a = 0; s.b=1;
printf ("\ndans fct : %d %e", s.a, s.b);
}
```

```
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.

I.7.2. 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 noté -> :

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

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

I.8. Transmission d'une structure en valeur de retour d'une fonction

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

```
struct enreg fct (...)
{
  struct enreg s;          /* structure locale à fct */
  .....
  return s;                        /* dont la fonction renvoie la valeur */
}
```

Notez bien que s aura dû soit être créée localement par la fonction (comme c'est le cas ici), soit éventuellement reçue en argument.

Naturellement, rien ne vous interdit, par ailleurs, de réaliser une fonction qui renvoie comme résultat un pointeur sur une structure. Toutefois, il ne faudra pas oublier qu'alors la structure en question ne peut plus être locale à la fonction; en effet, dans

ce cas, elle n'existerait plus dès l'achèvement de la fonction… (mais le pointeur continuerait à pointer sur quelque chose d'inexistant !). Notez que cette remarque vaut pour n'importe quel type autre qu'une structure.

II. LES ENUMERATIONS

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

II.1. Exemples introductifs

Considérons cette déclaration :
enum couleur {jaune, rouge, bleu, vert};

Elle définit un type énumération nommé couleur et précise qu'il comporte quatre valeurs possibles désignées par les identificateurs jaune, rouge, bleu et vert. Ces valeurs constituent les constantes du type couleur.

Il est possible de déclarer des variables de type couleur :
```
enum couleur c1, c2;          /* c1 et c2 sont deux variables */
                                    /* de type enum couleur */
```

Les instructions suivantes sont alors tout naturellement correctes :
```
c1 = jaune;          /* affecte à c1 la valeur jaune */
c2 = c1;             /* affecte à c2 la valeur contenue dans c1 */
```

Comme on peut s'y attendre, les identificateurs correspondant aux constantes du type couleur ne sont pas des *lvalue* et ne sont donc pas modifiables :
```
jaune = 3;          /* interdit : jaune n'est pas une lvalue */
```

II.2. Propriétés du type énumération

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

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

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

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

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

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

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

```
enum booleen {faux, vrai};
enum couleur {jaune, rouge, bleu, vert};
enum booleen drapeau;
enum couleur c;
.....

c = drapeau;      /*OK bien que drapeau et c ne soit pas d'un même type*/
drapeau = 3 * c + 4;          /* accepté */
```

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

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

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

Les entiers négatifs sont permis comme dans :

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

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

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

Remarque

- Comme dans le cas des structures ou des unions, on peut mixer la définition d'un type énuméré et la déclaration de variables utilisant le type. Par exemple, ces deux instructions :
  enum couleur {jaune, rouge, bleu, vert};
  enum couleur c1, c2;

  peuvent être remplacées par :

enum couleur {jaune, rouge, bleu, vert} c1, c2;

Dans ce cas, on peut même utiliser un type anonyme, en éliminant l'identificateur de type :
enum {jaune, rouge, bleu, vert} c1, c2;

Cette dernière possibilité présente moins d'inconvénients que dans le cas des structures ou des unions, car aucun problème de compatibilité de type ne risque de se poser.

- Compte tenu de la manière dont sont utilisées les structures, il était permis de donner deux noms identiques à des champs de structures différentes. En revanche, une telle possibilité ne peut plus s'appliquer à des identificateurs définis dans une instruction enum. Considérez cet exemple :
  enum couleur {jaune, rouge, bleu, vert};
  enum bois_carte {rouge, noir}; /*erreur : rouge déjà défini*/
  int rouge; /*erreur : rouge déjà défini*/

  Bien entendu, la portée de tels identificateurs est celle correspondant à leur déclaration (fonction ou partie du fichier source suivant cette déclaration).

Exercice .1
Écrire un programme qui :

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

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

  Le nombre d'éléments du tableau sera fixé par une instruction **#define**.
- affiche à l'écran l'ensemble des informations précédentes.

Exercice .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.

FIN DU SUPPORT