Introduction à la programmation

S. Nicolay

Année académique 2009–2010

Chapitre 1

Quelques notions utiles

1.1 Codage

Définitions

Un *alphabet* est un ensemble L fini ; les éléments de L sont appelés les *caractères* de L et un ensemble ordonné de caractères est appelé un *mot*. La *longueur* d'un mot est le cardinal de l'ensemble le définissant. Un mot est donc *fini* si l'ensemble qui le défini est lui-même fini. Un mot formé des éléments ordonnés $a_1, a_2, \ldots, a_n$ de L sera noté $a_1 a_2 \cdots a_n$.

Soit M un ensemble de mots de L ; *coder* un ensemble[1] E consiste à trouver une application surjective de M dans E. Cette application est appelée un *codage*. La *base* du codage est le cardinal de L. Si l'application n'est pas bijective, le codage est dit *redondant*. Un codage est de *longueur fixe* si les mots de M sont tous de même longueur. Un élément de l'image inverse d'un élément de E est appelé une *représentation* de E.

Ainsi, coder l'ensemble des fleurs peut simplement consister à leur donner un nom. De même, on peut coder l'ensemble des étudiants de l'université en leur donnant un numéro. On peut aussi coder l'alphabet latin par le codage morse, coder l'ADN, *et cætera*.

Pour un codage de longueur fixe n et de base b, le cardinal de l'ensemble E doit être inférieur à b^n, i.e. b^n éléments au plus peuvent être représentés. Si le cardinal de E est effectivement b^n, on peut définir $b^n!$ codages différents de longueur n (à partir du même alphabet de cardinal b).

L'*ordre lexicographique* permet d'ordonner les mots construits sur un alphabet ordonné. Il peut être défini comme suit. Soit l'alphabet $L = \{l_1, \ldots, l_N\}$ muni d'un ordre total : $l_j < l_k$, $\forall j < k$. Le caractère l_1 est appelé *caractère neutre*. Soient m_1 et m_2 deux mots de L dont au moins un des caractères n'est pas le caractère neutre, $m_1 = a_1 a_2 \cdots a_{n_1}$ et $m_2 = b_1 b_2 \cdots b_{n_2}$, où $a_j, b_j \in L$, $\forall j$. Notons j_1 le plus petit indice tel que $a_{j_1} \neq l_1$ et j_2 le plus petit indice tel que $b_{j_2} \neq l_1$. Par définition, $m_1 = m_2$ si et seulement si $n_1 - j_1 = n_2 - j_2$ et $a_{j_1+j} = b_{j_2+j}$, $0 \leq j \leq n_1 - j_1$. Supposons que $m_1 \neq m_2$; on a $m_1 < m_2$ si $n_1 - j_1 < n_2 - j_2$. Si $n_1 - j_1 = n_2 - j_2$, soit k le plus petit indice tel que $a_{k+j_1-1} \neq b_{k+j_2-1}$. On a $m_1 < m_2$ si $a_{k+j_1-1} < b_{k+j_2-1}$. Dans tous les autres cas, on a $m_2 < m_1$.

Si $m_1 = a_1 \cdots a_{n_1}$ et $m_2 = b_1 \cdots b_{n_2}$ sont deux mots construits sur L, le mot $m = m_1 m_2$ est le mot construit sur L suivant, $m = a_1 \cdots a_{n_1} b_1 \cdots b_{n_2}$. On dit que le mot m est formé

[1] En pratique, les ensembles peuvent être considérés comme finis, puisque les ressources informatiques sont toujours finies.

par la *concaténation* des mots m_1 et m_2. De la même manière, si $l \in L$, le mot m formé par la concaténation du caractère l avec le mot m_1 est le mot $m = la_1 \cdots a_{n_1}$. Remarquons que par définition, $m = l_1 m$ pour tout mot m construit sur L. Un mot $m \neq l_1$ ne commençant pas par le caractère l_1, i.e. un mot m ne pouvant s'écrire $m = l_1 m'$, où m' est un mot de longueur au moins 1, est appelé *mot minimal*. Un codage ne faisant intervenir que des mots minimaux est appelé *codage minimal* et la représentation correspondante *représentation minimale*.

Exemples

Un codage binaire est un codage en base 2. On pose $L = \{0, 1\}$; les caractères sont appelés *bits*. On peut coder les nombres entiers compris entre 0 et 7 par un codage binaire de longueur fixe 3. Ainsi, par exemple,

nombre	représentation de longueur fixe	représentation minimale
0	000	0
1	001	1
2	010	10
3	011	11
4	100	100
5	101	101
6	110	110
7	111	111

Ce codage est un codage binaire particulier ; il est appelé codage binaire naturel (nous en reparlerons à la section suivante). Lorsque l'on travaille avec un codage de longueur fixe (ici 3), le bit le plus à gauche est appelé le *bit de poids fort* et le bit le plus à droite est le *bit de poids faible*.

Le codage de Gray évite le changement simultané de deux bits.

nombre	représentation
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Le codage DCB (Décimal Codé Binaire) : chaque chiffre d'un nombre est codé sur quatre bits.

nombre	représentation
0	0000
1	0001
2	0010
⋮	⋮
8	1000
9	1001
⋮	⋮
10	0001 0000
11	0001 0001
⋮	⋮
123	0001 0010 0011

Un codage alphanumérique est un codage d'un ensemble des symboles reconnus par un ordinateur par une application définie sur un sous-ensemble de l'ensemble des naturels. Ainsi, pour le *codage ASCII*, le symbole `A` est codé par le nombre 65, le symbole `e` par le nombre 101, le symbole `7` par 55, `!` par 33, *et cætera*. Pour ce codage, il existe 127 caractères reconnus (représentable par 7 bits). Le codage ASCII étendu comporte 255 caractères (représentable par 16 bits). Citons aussi le codage Unicode (UTF), plus récent.

1.2 Systèmes de numération

Soit b un nombre naturel strictement supérieur à 1. Pour tout nombre entier positif x, il existe un entier positif N, une suite $(x_k)_{k\in\mathbb{N}}$, avec $x_k \in \{0, 1, 2, \ldots, b-1\}$ et $x_k = 0$ si $k > N$ (on peut toujours supposer que $x_N \neq 0$, sauf si $x = 0$) telle que

$$x = \sum_{k=0}^{N} x_k b^k. \tag{1.1}$$

La suite $(x_k)_{k\in\mathbb{N}}$ est appelée *représentation en base* b de x. Si l'égalité (1.1) est vérifiée, nous écrirons $x = (x_N, x_{N-1}, \ldots, x_0)_b$. Étant donné un nombre entier positif x, les éléments d'une telle suite x_k peuvent successivement être déterminé grâce à l'*algorithme glouton*, défini comme suit.

- Soit N l'entier tel que $b^N \leq x < b^{N+1}$ et posons $y_N = x$.
- Soit x_N le plus grand naturel tel que $x_N b^N \leq y_N$ et posons $y_{N-1} = y_N - x_N b^N$. Par définition de N, $x_N < b$ et $y_{N-1} \geq 0$.
- Si y_{N-1} vaut 0, on pose $x_k = 0$ quelque soit $0 \leq k \leq N-1$ et on s'arrête. Sinon, on définit x_{N-1} comme le plus grand naturel tel que $x_{N-1}b^{N-1} \leq y_{N-1}$ et on pose $y_{N-2} = y_{N-1} - x_{N-1}b^{N-1}$.
- ...
- Si y_{N-j} ($j \in \mathbb{N}_0$) vaut 0, on pose $x_k = 0$ quelque soit $0 \leq k \leq N-j$ et on s'arrête. Sinon, on définit x_{N-j} comme le plus grand naturel tel que $x_{N-j}b^{N-j} \leq y_{N-j}$ et on pose $y_{N-j-1} = y_{N-j} - x_{N-j}b^{N-j}$.
- On recommence l'étape précédente avec $j+1$ en lieu et place de j.

En fait tout nombre réel positif peut être représenté de cette manière si on autorise l'indice k à prendre des valeurs négatives : il existe un entier N et une suite $(x_k)_{k\in\mathbb{Z}}$, avec

$x_k \in \{0, 1, 2, \ldots, b-1\}$ et $x_k = 0$ si $k > N$ telle que

$$x = \sum_{k=-\infty}^{N} x_k b^k. \tag{1.2}$$

Remarquons cependant que la représentation n'est plus nécessairement unique et que, si l'algorithme glouton peut être aisément adapté, il ne se termine pas nécessairement. Cette méthode peut aussi être généralisée aux bases réelles $b \in]1, +\infty[$.

Choisir un *système de numération*, i.e. une représentation en base b pour les nombres entiers positifs revient à définir un codage à partir de l'alphabet $L = \{0, 1, \ldots, b-1\}$ préservant l'ordre lexicographique, c'est-à-dire tel que si deux mots m_1 et m_2 représentant respectivement deux nombres x_1 et x_2 vérifient $m_1 < m_2$, alors $x_1 < x_2$. Le codage préservant cette relation est appelé le *codage naturel* en base b. Un codage naturel en base b peut aussi être définis pour les nombres réels positifs.

Lorsque la base b est sous-entendue, on représente plus communément le nombre $(x_N, x_{N-1}, \ldots, x_0)_b$ comme suit, $x_N x_{N-1} \cdots x_0$. Par exemple, le nombre $(1, 2, 3, 4)_{10}$ s'écrit implicitement 1234 ; en base 2, ce nombre s'écrit $(1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0)_2$, ou encore 10011010010. La représentation d'un nombre en base $b = 10$ est la *représentation décimale*, universellement adoptée par l'homme. La raison provient certainement du nombre de doigts que l'homme possède. En ce qui concerne l'ordinateur, la représentation des nombres est basée sur une succession d'états bipolaires (allumé/éteint). La base naturelle associée est donc la base 2 ; on parle de *représentation binaire*.

1.3 Représentation de nombres par des mots de longueur fixe

Cas des nombres entiers positifs

Si on souhaite représenter les nombres de l'ensemble $E = \{0, 1, 2, \ldots, n\}$ en base b, la longueur maximale des mots intervenant dans cette représentation, i.e. la valeur du plus grand nombre naturel N dans la somme (1.1) vaut exactement[2] $N = \lfloor \log_b n \rfloor$. On peut supposer que tous les mots sont de longueur $N+1$; en effet, comme nous l'avons déjà vu, si m est un mot de longueur $N' < N+1$, le mot de longueur $N+1$, $0 \cdots 0m$ obtenu en concaténant $N+1-N'$ zéros au mot m est équivalent à $m : 0 \cdots 0m = m$ et la suite $(x'_k)_k$ telle que $x'_k = x_k$ si $0 \leq k \leq N'$ et $x'_k = 0$ si $N' < k \leq N+1$ définit le même nombre x que la suite $(x_k)_k$, selon l'égalité (1.1). Ainsi, si E est un sous-ensemble fini de $\mathbb{N}$, on peut supposer que le codage naturel est de longueur fixe. Inversement, si les mots intervenant dans la représentation de E ont une longueur maximale N, E ne peut contenir plus de b^N éléments.

Nous avons vu que la base privilégiée de l'ordinateur est $b = 2$. Les bits sont généralement regroupés en puissance de 2 : $N = 2^{k+3}$, $k \in \mathbb{N}$. Une succession de 2^3 bits forment un octet, 2^{10} octets forme un kilo-octet (Ko), *et cætera*. Ainsi, avec un octet, on peut coder les entiers compris entre 0 et 255. Avec deux octets, les entiers compris entre 0 et 65535 peuvent être codés.

[2] Rappelons que $\lfloor x \rfloor = \max\{j \in \mathbb{N} : x \geq j\}$.

nombre d'octets	dénomination
2^{10}	kilo-octet (Ko)
2^{20}	mega-octet (Mo)
2^{30}	giga-octet (Go)
2^{40}	tera-octet (To)
2^{50}	péta-octet (Po)
2^{60}	exa-octet (Eo)
2^{70}	zetta-octet (Zo)
2^{80}	yotta-octet (Yo)

Cas des nombres entiers négatifs

Dorénavant, nous supposerons que $b = 2$. Nous venons de voir qu'avec N bits, on peut représenter les nombres entiers compris entre 0 et $2^N - 1$. Si l'on souhaite pouvoir représenter également les nombres négatifs, il est nécessaire d'adopter une convention. On convient que le bit de poids fort, i.e. le caractère le plus à gauche de la représentation de longueur fixe (ici N) en base 2 du nombre détermine la parité de ce nombre : s'il vaut 1, le nombre est négatif et s'il vaut 0, le nombre est positif.

Pour la *notation complémentée* à 1, prendre l'opposé d'un nombre, revient à inverser chaque bit de la représentation : le caractère 1 devient 0 et le caractère 0 devient 1. Avec cette convention, N bits permettent de représenter les nombres entiers compris entre $-2^{N-1} + 1$ et $2^{N-1} - 1$.

nombre	représentation en base 2 sur 1 octet (notation complémentée à 1)
1	00000001
−1	11111110
2	00000010
−2	11111101
⋮	⋮
127	01111111
−127	10000000

Pour la notation complémentée à 2, on obtient l'opposé d'un nombre entier en inversant chaque bit, puis en additionnant 1. Les nombres ainsi représentables avec N bits sont ceux compris entre -2^{N-1} et $2^{N-1} - 1$.

nombre	représentation en base 2 sur 1 octet (notation complémentée à 2)
−1	11111111
−2	11111110
⋮	⋮
−127	10000001
−128	10000000

Représentation des nombres réels

S'il existe plusieurs méthodes possibles pour représenter les nombres réels (où de les approximer), la *représentation en virgule flottante* est la plus usitée. Citons aussi la *représentation en virgule fixe* et l'utilisation d'un couple d'entier afin de manipuler les rationnels. Pour la représentation en virgule flottante un nombre x est représenté comme suit, $x = \pm b^n m$. Les composantes de la représentation sont donc le signe, la *mantisse* m et l'exposant n. La représentation est dite normalisée si $m \in [1, b[$. Rappelons que nous supposons ici $b = 2$.

Selon la norme *IEEE 754*, les nombres réels sont codés sur 32 bits (simple précision) ou 64 bits (double précision). Sur 32 bits, $x = (-1)^s 2^{n-127}(1 + m)$. Un bit est utilisé pour la parité s, 8 pour l'exposant n et 23 pour la mantisse m (ou plutôt pour la mantisse moins un). Le décalage de 127 dans l'exposant est appelé un excès (de 127) ; en double précision, l'excès est de 1023. Par exemple,

$$1\,10000001\,01000000000000000000000$$

représente le nombre -5, puisque $(-1)^1 2^{129-127}(1 + 1/4) = -5$. Il existe des éléments particuliers représentés avec la norme *IEEE 754*.

n	m	élément représenté
maximum	0	l'infini
maximum	$\neq 0$	NaN (pas un nombre)
0	0	0
0	$\neq 0$	nombre dénormalisé

Remarquons que tout nombre réel ne peut être représenté de cette manière. Ils sont en fait approximés. De même, la somme et le produit de deux nombres représentés en virgule flottante doit parfois être approximée. Par exemple, la multiplication de deux mantisses codée sur N bits donne un résultat sur $2N$ bits. Le produit de deux nombres représentés en virgule flottante peut donc ne pas être représentable en virgule flottante avec le même nombre de bits. Il existe plusieurs méthode d'arrondis (même si on se restreint aux arrondis *IEEE*) ; la méthode la plus répandue consiste à choisir le nombre y représentable en virgule flottante le plus proche du nombre x que l'on souhaite représenter (*arrondi au plus près*). Il peut aussi arriver que le nombre que l'on souhaite représenter ne puisse pas l'être avec le codage choisi. C'est notamment le cas si l'exposant du nombre à représenter en virgule flottante est supérieur à la valeur maximum (i.e. supérieur à 255 en simple précision). On parle de *débordement*.

1.4 Algèbre de Boole

Soit $B = \{0, 1\}$ (il s'agit de deux éléments particuliers et non de nombres entiers) et $E \supset B$. Soient les opérations binaires .&., .|. et l'opération unaire !. vérifiant les axiomes suivants, quelque soient a, b et $c \in E$:

commutativité	$a\|b = b\|a$	$a\&b = b\&a$
associativité	$(a\|b)\|c = a\|(b\|c)$	$(a\&b)\&c = a\&(b\&c)$
distributivité	$a\&(b\|c) = (a\&b)\|(a\&c)$	$a\|(b\&c) = (a\|b)\&(a\|c)$
élément neutre	$a\|0 = a$	$a\&1 = a$
complémentation	$a\|!a = 1$	$a\&!a = 0$

Remarquons que plus conventionnellement $a\&b$ est dénoté $a \wedge b$, $a|b$ est dénoté $a \vee b$ et $!a$ est dénoté $\neg a$. Nous avons adopté ici les convention du *langage C*. Les propriétés suivantes sont évidentes.

idempotence	$a\|a = a$	$a\&a = a$
absorption	$a\|(a\&b) = a$	$a\&(a\|b) = a$
dualité	$!(a\|b) = !a\&!b$	$!(a\&b) = !a\|!b$
élément absorbant	$a\|1 = 1$	$a\&0 = 0$

Ici, nous nous restreindrons à $E = B$. Cette algèbre peut être interprétée comme suit : 1 correspond à « vrai/allumé », 0 à « faux/éteint », | à « ou », & à « et », ! à « non ». Une *table de vérité* est un outil permettant de représenter un phénomène logique passif. Comme nous le verrons, ces outils sont couramment utilisés en électronique (porte logique) et en informatique (tests). Une table de vérité est un tableau qui représente des entrées (en colonne) et des états binaire (0/1). Une sortie, également représentée sous forme de colonne, est la résultante des états d'entrée, elle-même exprimée sous forme d'état binaire.

a	$!a$
0	1
1	0

a	b	$a\|b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a\&b$
0	0	0
0	1	0
1	0	0
1	1	1

Une *fonction booléenne* est une fonction de B^d dans B. Une telle fonction peut être décrite par sa table de vérité ou une expression booléenne. Il y a 2^{2^d} fonctions booléennes à d arguments. Traitons un exemple : soit f la fonction définie sur B^3 qui vaut 1 si et seulement si au moins deux de ses arguments sont égaux à 1. Nous l'appellerons fonction majorité. On a tôt fait de construire la table de vérité suivante.

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

On vérifie aussitôt que $f(a,b,c) = (!a\&b\&c)|(a\&!b\&c)|(a\&b\&!c)|(a\&b\&c)$.

1.5 Les unités fonctionnelles d'un ordinateur

Un *bus* est un ensemble de fils électriques par lesquels transitent les informations entrent les unités.

La *mémoire* est un vecteur dont chaque composante est accessible via une *adresse* (une adresse détermine donc une position en mémoire). Les opérations permises sur la mémoire sont les opérations de lecture et d'écriture.

L'*unité arithmétique logique* (UAL) est l'organe de l'ordinateur chargé d'effectuer les calculs. L'UAL peut être vue comme une fonction de trois paramètres : une opération et deux arguments ; l'UAL renvoie alors un résultat.

L'*unité de commande* régit l'ordinateur via quelques signaux de contrôle en se basant sur les instructions à exécuter et la nature du résultat des opérations sur l'UAL, sans prendre en considération les données traitées ou les résultats produits. L'unité de commande comprends notamment une *horloge*, chargée de synchroniser l'ensemble des actions d'un ordinateurs. Elle cadence les instructions avec une fréquence constante : l'horloge divise le temps en battement de même durée, appelés *cycles*.

L'*unité centrale* (UC) exécute les programmes en chargeant les instructions. Elle est constituée d'une UAL et d'une unité de commande.

L'*unité d'entrée/sortie* sert d'interface avec les périphériques (clavier, écran,...). Les opérations associées (lecture et/ou écriture) dépendent du périphérique.

Un ordinateur peut être vu comme l'assemblage d'une mémoire, d'une UAL, d'une unité de commande, d'une unité d'entrée/sortie et d'un bus.

1.6 Qu'est-ce qu'un programme

Définitions de base

La notion d'algorithme est souvent associée aux ordinateurs. La trace des premiers algorithmes remonte à l'antiquité (le calcul du plus grand commun diviseur de deux nombres est décrit dans le livre 7 des *Éléments d'Euclide*). Un algorithme est une méthode qui permet une résolution systématique d'un problème, en décrivant les diverses étapes menant à une solution. En d'autres termes, étant donné un problème, un *algorithme* de résolution est une liste ordonnée d'opérations dont l'application aux données du problème conduit à une solution de ce problème. Vu leur caractère systématique, les algorithmes sont particulièrement bien adaptés à un traitement par ordinateur. Pour ce faire, les algorithmes doivent être rédigés ou transcrits dans un langage de programmation.

Un *langage de programmation* est un langage formel. Ce langage formel est construit sur un alphabet ; pour qu'un mot appartienne au langage, il doit être formé à partir d'un lexique associé au langage et vérifier certaines contraintes. Ces dernières sont exprimées dans une grammaire formelle : c'est la syntaxe du langage de programmation. En associant une sémantique à un langage de programmation, on définit le comportement de la machine qui sera associé aux différentes constructions du langage. Une *instruction* est un mot du langage, représentant un ordre. Un *programme* est une suite ordonnée d'instructions.

Genèse d'un programme exécutable

Pour les langages de programmation algorithmiques, tels que le *Java*, le *C*, le *Pascal*, le *Fortran* ou l'*ADA*, le programme source, i.e. la suite d'instructions entrée par l'utilisateur est généralement transformé en un programme exécutable. Plus précisément, le programme de l'utilisateur est traduit en un autre langage, moins abstrait et plus proche de l'ordinateur, le langage machine. On appelle *compilateur* un logiciel appliquant cette transformation. Un compilateur fonctionne par analyse–synthèse : au lieu de remplacer chaque instruction du langage source par une instruction équivalente du langage machine, il commence par analyser le texte source pour en construire une représentation intermédiaire qu'il traduit à son tour en langage machine. Le compilateur est séparé en

- une partie frontale, parfois appelée « souche », qui lit le texte source et produit la représentation intermédiaire,
- une partie finale, qui parcourt cette représentation pour produire le langage machine.

Dans un compilateur idéal, la partie frontale est indépendante du langage machine (qui peut varier d'un ordinateur à l'autre, ou plutôt d'une architecture à l'autre), tandis que la partie finale est indépendante du langage source. Certains compilateurs effectuent des traitements substantiels sur la forme intermédiaire, indépendants à la fois du langage source et du langage machine. Les étapes de la compilation sont en général les suivantes :

1. analyse lexicale,
2. analyse syntaxique,
3. analyse sémantique,
4. transformation du code source en code intermédiaire,
5. optimisation du code intermédiaire,
6. traduction du code intermédiaire en code objet, avec éventuellement l'insertion de données de débogage et d'analyse de l'exécution,
7. édition des liens.

Le *fichier objet* contient non seulement la traduction en langage machine du texte en langage source, mais aussi diverses informations déductibles du programme source (exigences mémoires, appel à des fichiers prés-compilés[3] ou des bibliothèques[4],...). L'*édition des liens* est un processus qui permet de créer le programme en langage machine proprement dit à partir du fichier objet ; il lie les fichiers objets avec les fichiers pré-compilés d'une ou plusieurs bibliothèques. Remarquons que certains langages plus récents (*Java*, *.NET*) ne font pas appel à l'édition des liens et résolvent les adresses dynamiquement, au prix d'un temps de calcul plus important.

Remarquons enfin que certains langages ne sont pas du tout compilés mais *interprétés*. Citons par exemple les langages *Gambas*, *Perl*, *PostScript*, *Python*, *Ruby*, *SQL* et *Tcl*. Les langages interprétés sont généralement plus lents que les langages compilés.

1.7 Le langage *C*

Le langage *C* a été créé au milieu des années 70 par Dennis Ritchie. Ce langage partage avec le *Pascal* un ancêtre commun : l'*Algol*. Si le *Pascal* possède des avantages pédagogiques certains, il n'est pas aussi puissant que le *C*. Ce dernier présente les caractéristiques d'un langage de haut niveau, mais offre aussi de nombreuses possibilités du langage machine (c'est d'ailleurs une des raisons de sa création). C'est en *C* qu'est rédigé le système d'exploitation *UNIX*. Le *C* est aussi renommé pour sa portabilité.

En *C*, les instructions sont terminées par un point-virgule.

[3]Un fichier pré-compilé est une partie de programme traduit en code objet. Le programmeur peut ainsi découper un programme en plusieurs sous-programmes, qui seront liés lors de l'édition des liens.

[4]Nous définirions la notion de bibliothèque par la suite. Pour l'instant, bornons-nous simplement à préciser que le programmeur peut faire appel dans son programme à des pans de programmes spécifiques (des fonctions), indépendants du programme source lui-même. Nous traiterons par exemple le cas de la fonction `printf` en *C*.

Chapitre 2

Variables et opérateurs

2.1 Variables

Principes

Une variable est une donnée dont la valeur est modifiable. Pour le programme, une *variable* est la donnée de trois attributs :

- un identificateur (le nom de la variable), qui est un mot défini par le programmeur,
- une adresse, qui localise une zone précise de la mémoire (une suite de bits) où seront codées et mémorisées les valeurs prise par la grandeur représentée par la variable,
- un type, qui précise la nature des valeurs en question où plus précisément la manière dont doit être interprétée la suite de bits correspondant à la zone mémoire allouée à la variable (cette suite peut représenter un entier, un réel, un caractère alphanumérique,...).

Ces trois attributs sont déterminés dans un programme au moyen de déclarations de type, de déclaration de variable et d'affectation. L'affectation d'une valeur à une variable est l'action consistant à placer en mémoire (celle allouée à la variable) une valeur effective. C'est donc par l'intermédiaire des variables que l'on peut stocker et accéder à des données en mémoire. C'est le compilateur et non le programme qui détermine l'emplacement d'une variable en mémoire.

En *C*, les règles que doit respecter un nom de variable sont les suivantes : un nom est une suite de un ou plusieurs caractères ; ces caractères peuvent être des lettres, des chiffres ou le caractère de soulignement « _ », le premier caractère ne pouvant être un chiffre. La longueur du nom peut être quelconque, mais en général seuls les premiers caractères sont pris en compte par le compilateur (en général, les 8 ou 32 premiers caractères servent à identifier la variable). On veillera donc à définir des variables dont les noms se différencient sur les 8 premiers caractères. Il existe une seconde restriction concernant l'attribution du nom d'une variable. En effet, certains mots ont une signification particulière et ne peuvent donc être employés pour identifier une variable. C'est ce qu'on appelle les *mots réservés*. Les mots réservés suivants sont standard.

type	occupation mémoire	plage de valeurs
`char`	1 octet	$[-128, 127] \cap \mathbb{N}$
`unsigned char`	1 octet	$[0, 255] \cap \mathbb{N}$
`int`	2 ou 4 octets	
`unsigned int`	2 ou 4 octets	
`short`	2 octets	$[-32768, 32767] \cap \mathbb{N}$
`unsigned short`	2 octets	$[0, 65535] \cap \mathbb{N}$
`long`	4 octets	$[-2\ 147\ 483\ 647, 2\ 147\ 483\ 648] \cap \mathbb{N}$
`unsigned long`	4 octets	$[0, 4\ 294\ 967\ 295] \cap \mathbb{N}$
`float`	4 octets	$[3,4 \cdot 10^{-38}, 3,4 \cdot 10^{38}]$
`double`	8 octets	$[1,7 \cdot 10^{-308}, 1,7 \cdot 10^{308}]$
`long double`	10 octets	$[3,4 \cdot 10^{-4932}, 3,4 \cdot 10^{4932}]$

TAB. 2.1 – Occupation mémoire et plages de valeurs des types élémentaires en *C*. Rappelons que les nombres réels ne peuvent pas tous être représentés de manière exacte.

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

Il en existe d'autres, propres aux implémentations du langage. Citons par exemple `near`, `far` et `huge`. Cette liste n'est pas exhaustive (citons encore `fortran`, `pascal`,...).

Définition de variables dans un programme

Une variable doit être définie avant de pouvoir être utilisée. La définition permet de définir le nom et le type de la variable. C'est aussi au moment de la définition qu'une partie de la mémoire est réservée ; le nombre de bits alloué dépend du type (cf. Tableau 2.1). La syntaxe est la suivante :

```
type nom1 [,nom2 [,nom3[,...]]];
```

Les caractères contenus entre crochets sont facultatifs. On a ainsi définit une (`nom1`) ou plusieurs variables (`nom2`, `nom3`) du type `type`. Par exemple, pour définir une variable de type `int` (entier) nommée `j`, on écrira

```
int j;
```

Un espace mémoire de 2 octets sera alloué à la variable `j`. De manière similaire, on peut définir 3 variables de type `float` comme suit.

```
float valeur1,exposant,valeur2;
```

Pour chacune de ces variables, nommées `valeur1`, `exposant` et `valeur2`, un espace mémoire de 4 octets sera alloué, soit un total de 12 octets.

Affectations

En *C*, l'affectation d'une valeur à une variable se fait au moyen de l'opérateur symbolisé par `=`. Ainsi

```
x=5;
```

affecte la valeur 5 à la variable identifiée par `x`. C'est donc un ordre d'affectation et non l'affirmation « la valeur actuellement affectée à la variable `x` est 5 ».

Le fait de définir une variable ne lui affecte aucune valeur. Si la commande

```
int j;
```

réserve une partie de la mémoire afin de pouvoir affecter une valeur à la variable `j`, on ne peut préjuger de la valeur associée à `j` après cette définition. En fait, cette valeur est aléatoire et dépend principalement du programme qui utilisait précédemment la portion de la mémoire maintenant affectée à `j`. Pour qu'une variable possède une valeur dès sa définition, il faut l'initialiser. On peut par exemple écrire

```
int j=0;
```

Cette commande, en plus de définir la variable `j`, lui affecte la valeur 0. Si la valeur initiale que prend cette variable ne doit plus être changée par la suite, on peut utiliser le mot réserver `const`. Si on veut affecter la valeur 0 à `j` de manière permanente, on écrira

```
const int j=0;
```

La variable ainsi définie est non modifiable.

Considérons les instructions suivantes

```
char caractere;
int i=5,j;
float exposant;

caractere='Z';
j=i;
exposant=3.14;
i=3;
```

Les trois premières lignes définissent quatre variables de type différent dont une (la variable identifiée par `i`) est initialisée (avec la valeur 5). L'instruction `caractere='Z' ;` modifie la valeur associée à `caractere`. La valeur de cette variable est celle associée au caractère `Z` dans la table ASCII (c'est-à-dire la valeur 90). L'instruction `j=i ;` affecte à la variable `j` la valeur associée à la variable `i`, mais ne lie en aucune manière ces deux variables. Après cette instruction, ces deux variables possèdent donc la même valeur 5. De la même manière, `exposant=3.14 ;` affecte à la variable `exposant` la valeur $3,14$ et `i=3 ;` affecte à `i` la valeur 3. Cette dernière instruction ne concerne pas la variable `j` qui conserve sa valeur (5). Après ces instructions les variables ont donc les valeurs suivantes.

identifiant de la variable	valeur associée
`caractere`	90
`i`	3
`j`	5
`exposant`	$3,14$

Remarquons que si l'instruction `j=exposant ;` ne générera aucune erreur auprès du compilateur, elle est cependant ambiguë. En effet, j étant de type `int`, il ne peut contenir la valeur $3,14$. C'est ce qu'on appelle un problème de conversion de type. Nous en reparlerons par la suite.

2.2 Expressions et opérateurs

Expressions

Une *expression* est une construction synthétique qui est associée à une valeur et un type ; les expressions font intervenir des opérandes, c'est-à-dire des variables, des constantes, ainsi que des appels à des fonctions (nous définirons les fonctions dans la suite) et des opérateurs. Un opérande isolé est une expression. Les expressions peuvent avoir des *effets de bord* : l'évaluation d'une expression peut entraîner l'exécution d'actions (comme par exemple[1] l'affichage à l'écran de certaines valeurs). Voici quelques expressions

```
j=2;
i=j;
j=i+5;
i=((5*i+4)/2)& 1;
j++;
```

Les opérateurs sont ici représentés par les symboles `=`, `+`, `*`, `/`, `&` et `++`. Le type d'une expression dépend du type de ses opérandes ; il est par exemple naturel de supposer que la somme de deux variables de type `int` soit de type `int`.

Opérateurs

Le langage *C* dispose de plus de quarante opérateurs. Il existe des opérateurs unaires, des opérateurs binaires et même un opérateur ternaire : l'opérateur conditionnel.

Les *opérateurs arithmétiques* procèdent à des opérations arithmétiques sur leurs opérandes.

opérateur	signification	exemple
`+`	addition	`x+y`
`-`	soustraction	`x-y`
`*`	multiplication	`x*y`
`/`	division	`x/y`
`%`	modulo	`x%y`
`-`	opposé	`-x`

Remarquons que tous ces opérateurs sont des opérateurs binaires, à l'exception du dernier, qui est unaire.

Les *opérateurs de comparaison* sont des opérateurs binaires servant à tester la nature de la relation entre les valeurs associées à deux opérandes.

[1] Ainsi l'expression consistant en une assignation est utilisée pour son effet de bord qui consiste à affecter une nouvelle valeur à une variable.

opérateur	signification	exemple
`==`	égal	`x==y`
`!=`	différent	`x !=y`
`<=`	inférieur ou égal	`x<=y`
`>=`	supérieur ou égal	`x>=y`
`<`	strictement inférieur	`x<y`
`>`	strictement supérieur	`x>y`

Les opérandes d'une comparaison ne doivent pas nécessairement être du même type. Du point de vue logique, le résultat d'une assertion telle que `x==y` est vrai ou faux (selon que la valeur associée à `x` est égale, ou non, à la valeur associée à `y`). Ce résultat pourrait donc être codé sur un bit, mais, contrairement aux langages *Fortran* et *Pascal* par exemple, il n'existe pas de variable définie sur un bit en *C*. L'expression `x==y` est de type `int`. La valeur 0 est associée à la valeur logique faux et toute autre valeur que 0 correspond à la valeur logique vrai, la valeur d'une expression relative à une comparaison vraie étant 1. Ainsi, l'expression `1+2` est vraie car la valeur associée est 3, valeur qui diffère de 0 ; il en va de même pour l'expression `2`. Par contre, les expressions suivantes sont associées à la valeur logique faux : `0`, `x=0`, `2-2`, `x*0`. Puisque, par définition, la valeur associée à l'expression `2==3` est 0, l'instruction `x=(2==3)` associe à la variable `x` la valeur 0 (nous verrons que les parenthèses servent à préciser l'ordre dans lequel les opérations doivent s'effectuer). Par contre, puisque 2 est différent de 3, l'expression `x=(2 !=3)` associe à `x` la valeur 1 (puisque l'expression `2 !=3` est vraie).

Les *opérateurs logiques* effectuent les opérations classiques de la logique des prédicats.

opérateur	signification	exemple
`&&`	connexion ET	`x&&y`
`||`	connexion OU	`x||y`
`!`	négation	`!x`

Le non logique (`!`) est un opérateur unaire. L'évaluation d'une expression comportant un opérateur logique donne la valeur 0 si l'expression est fausse et 1 si elle est vraie. Ainsi, `x&&y` vaut 1 si ni `x` ni `y` ne sont associés à la valeur 0 et 0 sinon. De la même manière, `x||y` vaut 1 si et seulement si la valeur associée à `x` ou la valeur associée à `y` (ou les deux) est différente de 0. Enfin, `!x` vaut 0 si et seulement si `x` n'est pas associé à la valeur 0. Remarquons que les expressions `!x` et `x==0` sont logiquement et numériquement équivalentes. Les expressions `x` et `x !=0` sont quand à elles logiquement équivalentes.

Les *opérateurs de bits* effectuent sur les bits de deux opérandes de type entier des opérations de connexions semblables à celles qu'effectuent les opérateur logiques sur les expressions. L'opération est appliquée en parallèle (donc en même temps) sur chaque bit individuel. Outre les opérations logiques ET, OU et NON, on peut également effectuer des opérations de décalage.

opérateur	signification	exemple
`&`	connexion binaire ET	`x&y`
`|`	connexion binaire OU	`x|y`
`^`	connexion binaire OU exclusif	`x^y`
∼	non binaire	`x∼y`
`>>`	décalage vers la droite	`x>>y`
`<<`	décalage vers la gauche	`x<<y`

Les opérateurs de bits exécutent les opérations logique ET, OU, OU exclusif et NON sur les bits correspondant, pris un à un, de leurs opérandes entiers. Rappelons que l'on a les tables de vérité suivantes.

bit1	bit2	∼bit1	bit1&bit2	bit1|bit2	bit1^bit2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

À titre d'exemple, considérons les instructions suivantes.

```
char x,y,z_et,z_ou,z_ou_ex,z_non;
x=10;
y=50;
z_et=x&y;
z_ou=x|y;
z_ou_ex=x^y;
z_non=~x;
```

On à tôt fait de vérifier, qu'après exécution, les valeurs associées au variable sont données par ce tableau.

identifiant	valeur
`z_et`	2
`z_ou`	58
`z_ou_ex`	56
`z_non`	245

De par leur nature, les opérateurs `&` et `|` permettent respectivement de désactiver et d'activer des bits. L'opérateur `>>` (resp. `<<`) décale vers la droite (resp. vers la gauche) tous les bits de l'opérande de gauche d'un nombre d'unité égal à la valeur de l'opérande de droite. En ce qui concerne l'opérateur `<<`, les positions binaires laissées libres sur la droite sont mises à zéro. Bien sûr, un décalage de n positions vers la gauche revient à multiplier par 2^n (s'il n'y a pas de débordement). Pour l'opérateur `>>`, les positions laissées libres sur la gauche sont mises à zéro si l'opérande de droite est de type non signé (`unsigned <type>`). Si l'opérande de droite est signé, les positions binaires libres sont remplies par des copies du bit de signe (0 si l'opérande de droite est positif et 1 s'il est négatif).

Les *opérandes d'affectation* mettent dans l'opérande de gauche une valeur faisant intervenir la valeur de l'opérande de droite. L'opérande de gauche doit être ce que l'on appelle une *lvalue*, i.e. une expression désignant une adresse de la mémoire (une variable par exemple). Une constante n'est pas une lvalue et ne peut donc être l'opérande de gauche d'une affectation.

opérateur	signification	exemple
`=`	affectation	`x=y`
`+=`	affectation avec addition	`x+=y`
`-=`	affectation avec soustraction	`x-=y`
`*=`	affectation avec multiplication	`x*=y`
`/=`	affectation avec division	`x/=y`
`%=`	affectation avec opération de reste	`x%=y`
`>>=`	affectation avec décalage à droite	`x>>=y`
`<<=`	affectation avec décalage à gauche	`x<<=y`
`&=`	affectation avec connexion binaire ET	`x&=y`
`|=`	affectation avec connexion binaire OU	`x|=y`
`^=`	affectation avec connexion OU exclusif	`x^=y`
`++`	incrémentation de 1	`x++` ou `++x`
`--`	décrémentation de 1	`x--` ou `--x`

L'affectation simple réalisée par l'opérateur `=` peut être décomposée en deux étapes : le membre de droite de l'affectation est d'abord évalué, puis cette valeur est attribuée au membre de gauche. L'expression relative à une affectation doit posséder une valeur ; cette valeur est celle de son membre de gauche. Ainsi après les instructions suivantes,

```
int a,b,c,d,e;
a=b=c=d=e=0;
```

toutes les variables seront associées à la valeur 0. Les affectations combinées mélangent l'opération d'affectation avec une opération arithmétique ou de bits.

opération d'affectation	opérations simples correspondantes
`x+=y`	`x=x+y`
`x-=y`	`x=x-y`
`x*=y`	`x=x*y`
`x/=y`	`x=x/y`
`x%=y`	`x=x%y`
`x>>=y`	`x=x>>y`
`x<<=y`	`x=x<<y`
`x&=y`	`x=x&y`
`x|=y`	`x=x|y`
`x^=y`	`x=x^y`

Les instructions d'incrémentation `x++ ;` et `++x ;` sont équivalentes à l'instruction `x=x+1 ;`. Cependant expressions `x++` (post-incrémentation) et `++x` (pré-incrémentation) diffèrent entre elles par leurs effets de bords (la valeur d'une des opérandes est modifiée pendant le traitement de l'expression). Considérons les instruction qui suivent.

```
int j=1,k;
k=j++;
```

Après ces instructions la valeur de la variable `j` est 2 et celle de `k` est 1. En effet, dans la dernière instruction, la valeur de `j` est d'abord affectée à la variable `k`. La variable `j` est ensuite incrémentée. Maintenant, à la fin des instruction

```
int j=1,k;
k=++j;
```

les deux variables `j` et `k` se voient attribuer la valeur 2 : dans la dernière instruction, la variable `j` est d'abord incrémentée, puis cette valeur est affectée à `k`. Les instructions de décrémentations `--x ;` et `x-- ;` sont équivalentes à l'instruction `x=x-1 ;`. Les mêmes remarques que pour l'opérateur d'incrémentation s'appliquent à l'opérateur de décrémentation.

L'*opérateur conditionnel* « `?` » est le seul opérateur ternaire du *C*. Une instruction utilisant l'opérateur conditionnel obéit à la syntaxe suivante :

```
expression1 ? expression2 : expression3
```

Si `expression1` est associé à la valeur 0, i.e. si cette expression est fausse, alors `expression3` est évalué et l'expression complète prend la valeur de `expression3` ; dans l'autre cas, `expression2` est évalué et l'expression complète prend la valeur de `expression2`. Par exemple, l'expression `(x>y) ? x : y` prend la valeur de `x` si la valeur de `x` est supérieure à celle de `y` et la valeur de `y` sinon.

L'*opérateur séquentiel* « `,` » permet de rassembler syntaxiquement deux expressions en une seule. Ainsi les instruction `x++ ;` et `y++ ;` peuvent être rassemblées en une seule instruction : `x++,y++ ;`.

L'*opérateur de dimension* `sizeof` est un opérateur unaire qui calcule l'occupation mémoire, en octets, d'une variable ou d'un type donné. Les deux syntaxes possibles sont

```
sizeof(<expression>)
```

ou

```
sizeof(<type>)
```

Après les définitions

```
short s;
long l;
double d;
```

les expressions

```
sizeof(s)
sizeof(l)
sizeof(d)
```

ont les valeurs 2, 4 et 8 respectivement.

L'*opérateur d'adressage* `&` est un opérateur unaire (à ne pas confondre avec l'opérateur binaire `&`) fournissant l'adresse de son opérande (qui doit être une lvalue).

L'*opérateur de parenthèsage* permet de regrouper librement les expressions comportant d'autres opérateurs. En d'autres termes, cet opérateur détermine l'ordre d'évaluation au sein d'une expression. Ainsi l'expression `5*7+3` est associée à la valeur 38, alors que l'expression `5*(7+3)` est associée à la valeur 50.

Les *opérateurs de champ* `.` et `->` servent à sélectionner des composantes dans des données complexes (structures). L'*opérateur d'indirection* `*` (à ne pas confondre avec l'opérateur binaire de multiplication) est un opérateur unaire permettant d'accéder à la valeur

pointée par une adresse. Il est très usités lorsque l'on manipule les pointeurs (nous en reparlerons dans un chapitre dédié).

Nous avons déjà vu que certains opérateurs sont prioritaires (par exemple, l'opérateur de multiplication à la priorité sur l'opérateur d'addition). Les opérateurs sont classés selon quinze niveaux de priorités, les opérateurs de niveau 15 étant les opérateurs avec une priorité absolue.

désignation	symbole	priorité	ordre
parenthèse	`()`, `[]`	15	de gauche à droite
opérateur de champs	`.`, `->`	15	de gauche à droite
opérateur cast	`(<type>)`	14	de droite à gauche
opérateur de taille	`sizeof`	14	de droite à gauche
opérateur d'adresse	`&`	14	de droite à gauche
opérateur d'indirection	`*`	14	de droite à gauche
opérateur de négation			
arithmétique	`-`	14	de droite à gauche
logique	`!`	14	de droite à gauche
binaire	$\sim$	14	de droite à gauche
incrément	`++`	14	de droite à gauche
décrément	`--`	14	de droite à gauche
multiplication	`*`, `/`	13	de gauche à droite
opérateur d'addition	`+`, `-`	12	de gauche à droite
opérateur de décalage	`>>`, `<<`	11	de gauche à droite
comparaison	`<`, `<=`, `>`, `>=`	10	de gauche à droite
	`==`, `!=`	9	de gauche à droite
opérateurs de bit	`&`	8	de gauche à droite
	`^`	7	de gauche à droite
	`|`	6	de gauche à droite
opérateurs logiques	`&&`	5	de gauche à droite
	`||`	4	de gauche à droite
opérateur conditionnel	`exp1 ?exp2 :exp3`	3	de droite à gauche
opérateurs d'affectation	`=`, `+=`, `-=`, `*=`, `/=`	2	de droite à gauche
	`%=`, `>>=`, `<<=`, `&=`, `|=`, `^=`	2	de droite à gauche
opérateur séquentiel	`,`	1	de droite à gauche

Conversions de type

La valeur de la variable `x` après l'instruction `x=2/4 ;` dépend de son type. Si `x` est de type `float`, la valeur de `x` sera $0,5$. Si `x` est de type `int`, la valeur associée à cette variable ne peut pas être $0,5$ puisque dans ce cas, `x` ne peut prendre que des valeurs entières. De manière plus générale on peut se demander quel sera la valeur associée à `x` après des instructions telles que

```
x=i*j/k;
```

En effet si les variables `i`,`j` et `k` sont de type `int`, le type de l'expression `i*j/k` sera-t-il lui aussi `int` ou dépendra-t-il du type de la variable `x` ? Le type de l'expression influence bien évidemment la valeur qui sera associée à `x` suite à l'instruction.

Considérons l'évaluation d'une expression faisant intervenir un opérateur binaire. Si plusieurs opérateurs binaires entrent en jeu, les règles qui suivent seront appliquées successivement à chaque opérateur, suivant la propriété qui leur est associée.

1. Les opérandes de type `char` et `short` sont convertis en `int`. De même les types `unsigned char` et `unsigned short` sont convertis en `unsigned int`.

2. Si l'un des opérandes est de type `long double`, l'autre est converti en `long double`. Le résultat de l'opération est également du même type.

3. Si aucun des opérandes n'est de type `long double`, alors si un des opérandes est de type `double` l'autre est converti en `double` et le résultat est aussi de type `double`.

4. Si aucun des opérandes n'est du type `long double` ou `double`, alors si un des opérandes est de type `float`, l'autre est converti en `float` et le résultat est aussi de type `float`.

5. Si aucun des opérandes n'est d'un type réel, alors si un des opérandes est de type `unsigned long`, l'autre est converti en `unsigned long` et le résultat est aussi de type `unsigned long`.

6. Si aucun des opérandes n'est d'un type réel ou `unsigned long`, alors si un des opérandes est de type `long`, l'autre est converti en `long` et le résultat est aussi de type `long`.

7. Si aucun des opérandes n'est d'un type rée ou `long`/`unsigned long`, alors si un des opérandes est de type `unsigned int`, l'autre est converti en `unsigned int` et le résultat est aussi de type `unsigned int`.

8. Dans les autres cas, les opérandes sont de type `int` et le résultat est de type `int`.

Il ne s'agit en fait que d'une règle de priorité. Remarquons que sur les anciens systèmes, les opérandes de type `float` étaient systématiquement convertis en `double`. Les dernières normes *ANSI*[2] autorisent dorénavant le calcul en virgule flottante avec simple précision.

Considérons les affectations. Le type de l'expression du membre de droite doit être harmonisé avec le type du membre de gauche. Le tableau qui suit résume la manière dont un type est converti en un autre.

[2] ANSI pour American National Standards Institute.

conversion	méthode
`double` → `float`	arrondi éventuel
`float` → `double`	valeur non modifiée
`float` → `long`	les décimales sont perdues. Le résultat est indéfini si la valeur ne peut être mémorisée sur 4 octets
`float` → `int`	la conversion est réalisée comme suit : `float` → `long`, puis `long` → `int`
`float` → `short`	`float` → `long`, puis `long` → `short`
`float` → `char`	`float` → `long`, puis `long` → `char`
`long` → `float`	arrondi si la valeur est trop grande pour la mantisse
`int` → `float`	`int` → `long`, puis `long` → `float`
`short` → `float`	`short` → `long`, puis `long` → `float`
`char` → `float`	`char` → `long`, puis `long` → `float`
`long` → `int`	les bits excédentaires à gauche sont omis
`long` → `short`	idem
`long` → `char`	idem
`int` → `short`	idem
`int` → `char`	idem
`short` → `char`	idem
`int` → `long`	valeur non modifiée
`short` → `long`	idem
`char` → `long`	idem
`short` → `int`	idem
`char` → `int`	idem
`char` → `short`	idem

Le programmeur peut lui-même expliciter le type d'une expression grâce à l'opérateur unaire `cast`. La syntaxe est la suivante.

```
(<type>) <expression>
```

Cette instruction convertit la valeur de l'expression `<expression>` pour qu'elle soit du type `<type>`. Par exemple, considérons les lignes

```
float x=1.25;
y=(int) x;
```

Après ces instructions, la valeur associée à la variable `y` sera 1 quel que soit le type de `y`.

Chapitre 3

Premiers programmes

3.1 Commentaires

Les commentaires permettent d'insérer des explications dans un programme afin d'en faciliter la compréhension. Un commentaire est une suite de caractères placée entre les délimitateurs « `/*` »et « `*/` » :

```
/* Ceci est un commentaire */
```

Les caractères placés entre les délimiteurs (ainsi que les délimiteurs) sont ignorés lors de la compilation. Tous les caractères peuvent être placés en commentaire à l'exception bien sûr du délimiteur de fin de commentaire (`*/`) ; les commentaires ne peuvent donc être imbriqués. Un commentaire ne peut scinder un mot *C*, mais peut être placé après un caractère espace, tabulation ou saut de ligne. Si les commentaires sont courts, il peuvent être introduit comme suit,

```
// Ceci est un commentaire
```

Les caractères compris entre le délimitateur `//` et le saut de ligne ne seront pas interprétés par le compilateur.

3.2 Entrées et sorties standards

La fonction `printf`

Nous allons ici introduire une fonction particulière, sans avoir même défini la notion de fonction. Il n'est pas nécessaire de maîtriser ce concept pour pouvoir utiliser à bon escient la fonction `printf`, si utile pour communiquer avec l'utilisateur du programme machine. Nous supposerons pour l'instant qu'une fonction est une expression (elle est donc associée à un type et une valeur).

La fonction `printf` écrit vers la sortie standard (`stdout`) une suite de données formatées par ses arguments.

```
int printf ( const char *<format>, ... );
```

En cas de succès, la fonction retourne le nombre de caractères effectivement écrits. Dans le cas contraire, un nombre négatif est retourné. Comme nous le verrons, la fonction

nécessite autant d'arguments additionnels que le nombre spécifié dans `<format>`, par l'intermédiaire des spécificateurs. La fonction `printf` se trouve dans la bibliothèque `stdio`

Voici un exemple simple qui a pour effet d'écrire « bonjour » à la sortie standard (typiquement l'écran).

```
printf("bonjour \n");
```

Le mot à afficher est placé entre les guillemets `"`. Les caractères « `\n` » ne seront pas affichés tels quels vers la sortie standard ; ils symbolisent un saut de ligne. Si la sortie standard est l'écran, le curseur se placera au début de la ligne suivante en lisant ces caractères. Pour afficher la valeur associée à une expression, il faut placer entre les guillemets des caractères spéciaux et passer comme argument l'expression. Les caractères spéciaux commencent par le caractère `%` et se terminent par un spécificateur, différent selon le type de la variable et le format souhaité ; ainsi

```
int x;

x=(5<<1)+1;
printf("la valeur de x est %d \n",x);
```

devrait afficher

```
la valeur de x est 11
```

à la sortie standard (l'écran le plus souvent). Si la variable `x` est du type float, il faut remplacer `%d` par `%f`. Par exemple

```
int x;
float y;

x=(5<<1)+1;
y=40.33
printf("une variable vaut %d \n et l'autre %f \n",x,y);
```

devrait afficher

```
une variable vaut 11
et l'autre 40.33
```

Les divers spécificateurs sont les suivants.

specificateur	sortie
`c`	caractère
`d` ou `i`	entier signé sous forme décimale
`e`	réel en notation scientifique (mantisse/exposant)
`E`	idem, mais la mantisse et l'exposant seront séparé par un caractère `E` et non un caractère `e`
`f`	réel en notation virgule flottante
`g`	choisit la plus courte des expressions entre `%e` et `%f`
`G`	idem, mais remplace éventuellement le caractère `e` par `E`
`o`	octal signé
`s`	chaîne de caractères
`u`	nombre décimal non signé
`x`	entier sous forme hexadécimale
`X`	idem, mais utilise des majuscules
`p`	adresse du pointeur
`n`	rien n'est affiché ; l'argument doit être un pointeur vers un entier signé, le nombre de caractères écrits sera placé dans la mémoire ainsi pointée
`%`	écrit le caractère `%` dans la sortie standard

La suite de caractères « `%<spécificateur>` » est remplacée à l'écriture par la valeur de l'expression entrée en argument ; ce n'est donc pas le programmeur qui décide, dans le cas d'un type réel par exemple, du nombre de chiffres après la virgule à afficher. En fait, ces paramètres peuvent être contrôlés si on remplace `%spécificateur` par le format plus général

`%[drapeau][largeur][.precision][longueur]specificateur`

drapeau	signification
`-`	justifie à gauche en supposant que la largeur est *width* (par défaut, c'est la justification à droite qui est choisie)
`+`	fait précéder la valeur par un signe `+` ou `-` par défaut, seuls les nombres négatifs sont précédés d'un signe
(espace)	s'il n'y a pas de signe à écrire, un caractère espace est affiché avant la valeur
`#`	utilisé avec les spécificateurs `o`, `x` ou `X`, la valeur est précédée par les caractères `0`, `0x` ou `0X` respectivement si elle est différente de 0 utilisé avec `e`, `E` ou `f`, la sortie contiendra le point des valeurs décimales, même si la valeur est entière par défaut, si la valeur est entière, le point n'est pas écrit utilisé avec `g` ou `G`, le résultat est le même que pour `e` ou `E`, à ceci près que les zéros terminaux ne sont pas enlevés
`0`	place des zéros à gauche afin de formater le nombre

Le nombre minimum de caractère à écrire (pour la valeur) est spécifié par `largeur`. Si la valeur à écrire est moins longue que le nombre `largeur`, des caractères espace sont affichés en début (des zéros sont concaténés à la représentation). La valeur n'est pas tronquée. Si `largeur` est le caractère `*`, alors la largeur sera précisée par un paramètre entier, placé avant l'expression à écrire.

Pour les spécificateurs de type entier (`d`, `i`, `o`, `u`, `x`, `X`), `precision` spécifie le nombre minimum de caractères à écrire. Si la longueur du nombre à écrire est plus courte que le nombre `precision`, le résultat est complété avec des zéros. La valeur n'est pas tronquée, même si le résultat est plus long. Pour les spécificateurs `e`, `E` et `f`, `precision` représente le nombre de chiffres à afficher après le point séparant la partie entière de la partie décimale. Pour les spécificateurs `g` et `G`, `precision` représente le nombre maximum de chiffre significatifs pouvant être écrits. Enfin pour `s`, `precision` est le nombre maximum de caractères pouvant être écrits (par défaut tous les caractères sont écrits, jusqu'à que le caractère fin de chaîne soit rencontré). La valeur par défaut de `precision` est 1. Si un point « `.` » est écrit sans valeur explicite de précision (`.` à la place de `.precision`), `precision` vaut 0. Si `precision` est le caractère « `*` », alors la précision devra être précisée par un paramètre entier, placé avant l'expression à écrire.

`longueur`	signification
`h`	l'argument est interprété comme étant de type `short int` ou `unsigned short int` (ne s'applique qu'aux spécificateurs `i`, `d`, `o`, `u`, `x` et `X`).
`l`	l'argument est interprété comme étant de type `long int` ou `unsigned long int` pour les spécificateurs de type entiers et comme étant de type `double` pour les spécificateurs de type réel
`L`	l'argument est interprété comme étant de type `long double` pour les spécificateurs de type réel

Nous sommes maintenant en mesure d'écrire notre premier programme complet.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main()
/*c'est ici que l'on place le corps du programme*/
{
  printf ("caracteres : %c %c \n", 'a', 65);
  printf ("entiers : %d %ld\n", 1977, 650000L);
  printf ("precede de caracteres espace : %10d \n", 1977);
  printf ("precede de zeros : %010d \n", 1977);
  printf ("systemes de numerations differents :
   %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
  printf ("reels : %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
  printf ("largeur en parametre : %*d \n", 5, 10);
  printf ("%s \n", "une chaine de caracteres");

  return 0;
/*lorsque main se termine, on retourne en general 0*/
}
```

On aura, à la sortie standard, les lignes suivantes.

```
caracteres: a A
```

```
entiers : 1977 650000
precede de caracteres espaces :       1977
precede de zeros : 0000001977
systemes de numerations differents : 100 64 144 0x64 0144
reels : 3.14 +3e+000 3.141600E+000
largeur en parametre :    10
une chaine de caracteres
```

La fonction `scanf`

La fonction `scanf` lit des valeurs dans l'entrée standard stdin (généralement le clavier) et les place en mémoire selon des types spécifiés par l'argument `<format>` à des adresses spécifiées par les autres arguments. Ces arguments devraient pointer vers des objets déjà alloués et de type spécifié par `<format>`.

```
int  scanf ( const char *<format> , ... );
```

En cas de succès, la fonction retourne le nombre d'éléments effectivement lus (ce nombre peut être inférieur à celui attendu, si une erreur à la lecture intervient). Si une erreur se produit avant qu'aucune donnée n'aie été lue, la fonction retourne `EOF` (qui est une constante, définie par une macro).

La fonction `scanf` se trouve dans la bibliothèque `stdio`.

Les caractères blancs lus sont ignorés (caractères espace, retour à la ligne, tabulation,...). Tout caractère autre qu'un caractère blanc placé dans `<format>` ne faisant pas partie d'une spécification (une spécification commence avec le caractère `%`) induit la lecture d'un caractère depuis l'entrée standard. Si ces deux caractères (le caractère lu et le caractère considéré dans `<format>`) coïncident, le caractère lu n'est pas pris en compte et le caractère suivant dans `<format>` est pris en considération. Si ces caractères ne correspondent pas, la fonction `scanf` s'arrête. Pour la fonction `scanf`, une spécification de format est de la forme

```
%[*][largeur][longueur]specificateur
```

L'astérisque `*` signifie que la donnée doit être lue depuis l'entrée standard mais pas prise en considération. Le nombre maximum de caractères pouvant être lus pendant l'opération est spécifié par `largeur`. Pour les autres composantes `longueur` et `specificateur`, elles ont été décrites avec l'introduction de la commande `printf`.

Un exemple très simple, consiste à demander à l'utilisateur d'entrer un nombre puis à afficher ce nombre.

```
int x;
scanf ("%d",&x);
printf("%d \n",x)
```

On constate que la syntaxe de la fonction `scanf` est similaire à celle de `printf`, à ceci près que le caractère `&` devance `x` dans la fonction `scanf`. C'est l'opérateur d'adresse ; en fait, ce n'est pas la valeur de `x` qui est passée en argument (dans ce cas, `x` ne serait pas précédé par `&`), mais l'adresse mémoire associée à `x`. En effet, la fonction `scanf` ne demande pas une valeur mais une adresse mémoire pour placer la valeur lue dans l'entrée standard (c'est-à-dire, le plus souvent, la valeur entrée au clavier). Dans ce cas, `scanf` va

placer à l'adresse associée à `x` la valeur lue. Au final, puisque la valeur de `x` est la valeur placée dans la mémoire associée, la valeur de `x` sera bien celle lue dans l'entrée standard et la fonction `printf` affichera la valeur entrée par l'utilisateur à la sortie standard.

Voici un exemple de programme faisant intervenir `scanf`.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 char chaine [80];
 int i;

 printf ("entrez votre nom : ");
 scanf ("%s",chaine);
 printf ("entrez votre age : ");
 scanf ("%d",&i);
 printf ("Monsieur/madame. %s , age : %d ans.\n",chaine,i);
 printf ("entrez un nombre hexadecimal : ");
 scanf ("%x",&i);
 printf ("vous avez entre %#x (%d).\n",i,i);

 return 0;
/*lorsque main se termine, on retourne en general 0*/
}
```

Voici un exemple de ce que pourrait entrer l'utilisateur et de la manière dont réagirait le programme.

```
entrez votre nom : Dupont
entrez votre age : 22
monsieur/madame Dupont , age : 22 ans.
entrez un nombre hexadecimal : ff
vous avez entre 0xff (255).
```

Chapitre 4

Structures de contrôle

Les structure de contrôle sont des structures qui peuvent être décrites à l'aide d'un petit nombre d'entre elles, les structure de base. Ces structures de bases sont l'enchaînement, l'alternative et la répétitions. L'*enchaînement* consiste simplement en l'écriture des instructions les unes à la suite des autres, dans l'ordre dans lequel elles doivent être exécutées. La plupart des ordinateurs sont conçus pour exécuter automatiquement l'enchînement (comme en attestent les programmes précédemment présentés).

4.1 L'alternative

L'alternative permet de programmer un test et de choisir les actions à réaliser en fonction du résultat du test. En théorie, la forme générale du test est la suivante,

si *condition* alors *action1*
sinon *action2*

En *C*, cette structure prend la forme

```
if (<expression>) <instruction1>;
[else <instruction2>;]
```

Si `<expression>` est vrai (i.e. si la valeur associée est non-nulle), alors l'instruction `<instruction1>` est exécutée. Sinon `expression` est faux (i.e. la valeur associée vaut zéro) et l'instruction `<instruction2>` est exécutée. L'instruction `else` peut être omise ; dans ce cas, si `<expression>` est vrai, alors l'instruction `<instruction1>` est exécutée, sinon aucune action n'est exécutée. Après cette structure, le programme enchaîne la commande suivante. Voici un exemple.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 float x,y;

 printf ("entrez x : ");
```

```
 scanf ("%f",&x);
 printf ("entrez y : ");
 scanf ("%f",&y);
 if (x>y)
  printf ("x est strictement plus grand que y \n");
 else printf ("x est inferieur ou egal a y \n");

 printf ("fin du programme \n");

 return 0;
/*lorsque main se termine, on retourne en general 0*/
}
```

Ce programme compare les valeurs associées à `x` et `y`. Ce qu'il va afficher à la sortie standard dépend des valeurs des variables `x` et `y`. Il existe deux possibilités.

```
entrez x : 5.2
entrez y : 3.5
x est strictement plus grand que y
fin du programme
```

```
entrez x : 5.2
entrez y : 7
x est inferieur ou egal a y
fin du programme
```

Supposons maintenant que la ligne comprenant l'instruction `else` ne soit pas présente dans le programme. On aurait alors

```
entrez x : 5.2
entrez y : 3.5
x est strictement plus grand que y
fin du programme
```

```
entrez x : 5.2
entrez y : 7
fin du programme
```

Si la valeur associée à `y` est supérieure ou égale à celle associée à `x`, le programme ne fait rien. Si l'on souhaite effectuer plusieurs instructions lorsque la condition est remplie (ou lorsqu'elle n'est pas remplie), il convient d'utiliser les délimiteurs « `{` » et « `}` ».

```
if (<expression>)
{
       <instruction1a>;
       <instruction1b>;
       ...
}
[else
{
```

```
        <instruction2a>;
        <instruction2b>;
        ...
}]
```

Voici un programme qui, après avoir demandé à l'utilisateur d'entrer les valeurs de deux variables `x` et `y`, affiche le résultat de la division `x/y`, pour autant que `y` soit non-nul. Si la valeur de `y` est zéro, un « message d'erreur » est affiché.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 float x,y;

 printf ("entrez x : ");
 scanf ("%f",&x);
 printf ("entrez y : ");
 scanf ("%f",&y);
 if (y!=0)
 {
       printf ("x/y= %f \n",x/y);
 }
 else printf ("y est nul, la division n'est pas definie \n");

 return 0;
}
```

4.2 La répétition

La répétition permet d'effectuer plusieurs fois consécutives la même opération. Le plus souvent le nombre de répétitions doit être fini. Il existe conceptuellement deux méthodes : prescrire le nombre de fois qu'il faudra exécuter l'action ou subordonner l'exécution de l'action à une condition. Cette seconde méthode englobe la première.

La structure `while`

La structure `while` permet de faire répéter une suite d'instructions tant qu'une certaine condition est vraie. La syntaxe est la suivante.

```
while (<expression>) <instruction>;
```

ou

```
while (<expression>)
{
       <instruction(s)>;
}
```

La potion `while(<expression>)` est appelée l'*en-tête de boucle* ; Les instructions `<instruction(s)>` forment le *corps de boucle*. Le fonctionnement de la boucle se décompose en deux phases :

1. l'expression `<expression>` est évaluée,
2. si elle est vraie, les instructions `<instruction(s)>` sont exécutées, puis on retourne au point 1.

La boucle `while` se termine dès que l'expression est fausse (i.e. vaut zéro). Ainsi, le programme

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 int z=3;

 while (z>0)
 {
       printf ("%d ",z);
       z--;
 }
 printf ("\n");

 return 0;
}
```

affichera

```
3 2 1
```

Remarquons que le programme peut être écrit de manière plus compacte.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 int z=3;

 while (!z) printf ("%d ",z--);
 printf ("\n");

 return 0;
}
```

La structure `do while`

La structure `do while` est similaire à la structure `while`, à ceci près que la structure `do while` teste la condition après avoir exécuté les instructions du corps de la boucle. La syntaxe est la suivante.

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

ou

```
do
{
       <instruction(s)>;
}
while (<expression>);
```

Dans le programme précédent, la structure `while` peut être remplacée par `do while` sans incidence sur le résultat.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 int z=3;

 do
 {
       printf ("%d ",z);
       z--;
 }
 while (z>0);
 printf ("\n");

 return 0;
}
```

Cependant, imaginons maintenant que la variable `z` est initialisée à zéro et non plus avec la valeur 3. Le programme utilisant la structure `while` n'affichera pas de nombre à l'écran, car dès la première évaluation de l'expression, on obtient la valeur 0 (i.e. la condition n'est pas vérifiée). Cependant, dans le programme utilisant la boucle `do while`, les instructions sont d'abord exécutées une première fois et la valeur `0` est ainsi écrite à la sortie standard ; `z` est ensuite décrémenté et vaut donc -1. Enfin, l'expression `z>0` est évaluée et comme elle est fausse, la boucle s'arrête. Le programme

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 int z=0;

 while (z==0);
```

```
 {
       printf ("entrez un nombre non nul : ");
       scanf("%d",&z);
 }

 printf ("l'inverse de du nombre est %f \n",1./z);

 return 0;
}
```

calcule l'inverse d'un nombre (de type `int`) entré par l'utilisateur. Si le nombre entré est nul, le programme redemande un nombre non-nul. Remarquons que dans l'expression `1./z`, le fait que le nombre 1 soit écrit avec le caractère `.` signifie qu'il doit être considéré comme étant de type `float`. L'expression `1./z` est donc elle aussi de type `float`. Ce programme peut être ré-écrit en utilisant la structure `do while`.

```
#include <stdio.h>
/*fait appel a la librairie stdio*/

int main ()
/*c'est ici que l'on place le corps du programme*/
{
 int z;

 do
 {
       printf ("entrez un nombre non nul : ");
       scanf("%d",&z);
 }
 while (z==0);

 printf ("l'inverse de du nombre est %f \n",1./z);

 return 0;
}
```

Dans ce dernier programme, la variable `z` n'est pas besoin d'être initialisée à zéro.

La structure `for`

Comme la structure `while`, la structure `for` est une boucle qui teste une condition avant d'exécuter les instructions qui en dépendent. Ces instructions sont répétées tant que la condition est remplie (i.e. est vraie). La syntaxe de l'instruction `for` est la suivante.

```
for (<expression_init> ; <expression_cond> ; <expression_reinit>)
       <instruction>;
```

ou

```
for (<expression_init> ; <expression_cond> ; <expression_reinit>)
{
       <instruction(s)>;
}
```

La première ligne représente l'en-tête de boucle, suivi par le corps, contenant les instruction à exécuter. L'en-tête contient les éléments suivants, séparés par des points-virgules.

- une expression dont le but principal est d'initialiser les variables de contrôle (`<expression_init>`),
- la condition de bouclage (`<expression_cond>`),
- une expression permettant de modifier les variables de contrôle (`<expression_reinit>`).

La structure `for` constitue une alternative syntaxique à la boucle `while`. Dans la boucle `for`, tous les éléments relatifs au contrôle de la boucle sont rassemblés dans l'en-tête. Lors de l'exécution de la boucle `for`, la première phase consiste généralement en une initialisation de variables. Ensuite la condition de bouclage est testée ; si l'expression est vraie, les instructions du corps de la boucle sont exécutées. Ensuite, les variables de contrôle sont modifiées et la condition de bouclage est de nouveau contrôlée. Si l'expression est à nouveau vraie, on répète les instructions du corps de la boucle. Dès qu'elle est fausse, la boucle s'arrête. L'expression d'initialisation n'est donc exécutée qu'une seule fois. Voici un exemple simple, affichant les entiers compris entre 1 et 10 par ordre croissant

```
#include <stdio.h>

int main ()
{
 int i;

 for(i=1 ; i<11 ; i++)
       printf("%d \n",i);

 return 0;
}
```

Le programme suivant calcule la somme des nombres impairs compris entre 1 et une valeur entrée par l'utilisateur.

```
#include <stdio.h>

int main ()
{
 long somme;
 /*la valeur de la somme*/
 int k,bs,n;
 /*
   k : variable de boucle
   bs : la valeur limite
   n : le nombre de termes additionnes
 */
```

```
  printf("entrez une valeur limite : ");
  scanf("%d", &bs);

  for(k=1, somme=0, n=0 ;  k<=bs  ; k+=2, n++)
    somme += k;

  printf("la somme des nombres impairs compris entre ");
  printf("1 et %d vaut %ld \n",bs,somme);
  printf("il y a %d nombres impairs compris entre 1 et %d \n",n,bs);

  return 0;
 }
```

Remarquons que si la condition `k<=bs` est fausse dès le début, i.e. si `bs` est associé à une valeur strictement inférieure à 1, les instructions de la boucle ne seront jamais exécutées (mais les variables `k`, `somme` et `n` seront bien initialisées). Il peut aussi y avoir un débordement : il est possible, si la valeur associée à `bs` est trop grande, que la somme ne soit pas représentable avec le type `long` (cf. Table 2.1). En fait, si on remarque que la somme S des nombres impairs compris entre 1 et K vaut

$$S = \sum_{\substack{k\in 2\mathbb{N}+1 \\ 1\leq k\leq K}} k = \frac{1}{2}\lfloor\frac{K+1}{2}\rfloor\big((2\lfloor\frac{K+1}{2}\rfloor - 1) + 1\big), \tag{4.1}$$

on a, si K est impair, $S = (K+1)^2/4$. De là, on obtient aisément la valeur à laquelle doit être strictement inférieure la variable `bs`, à savoir 92681. Dès lors, il peut sembler avantageux de modifier, dans le programme précédent, les lignes

```
  long somme;
  int k,bs,n;

  printf("entrez une valeur limite : ");
  scanf("%d", &bs);
```

par les instructions qui suivent.

```
  long somme,k,bs,n;

  do {
      printf("entrez une valeur limite comprise entre 1 et 92680 : ");
      scanf("%d", &bs);
     } while (bs<1 || bs>92680);
```

Remarquons que l'existence de la formule explicite (4.1) pour calculer la valeur S implique que le programme précédent n'est pas optimum.

Chapitre 5

Pointeurs

5.1 Définition de variables pointeurs

Un *pointeur* est une variable dont la valeur associée est une adresse mémoire. En *C*, les pointeurs sont dits « typés », autrement dit, un pointeur contient une adresse mémoire et cette adresse mémoire est associée à un type. Nous savons déjà que pour une variable quelconque `x`, l'opérateur `&` permet de former l'expression `&x`, qui représente l'adresse de la variable `x`. À l'exception des champs de bits et des variables de type `register`, les pointeurs peuvent référencer des donnée de chaque type. On définit une variable pointeur de la manière suivante.

```
<type> *<nom_pointeur>
```

Ainsi, `<nom_pointeur>` est une variable dont la valeur associée sera une adresse et le type associé à cette adresse sera le type `<type>`. Bien qu'il existe un pointeur pour chaque type, les variables pointeurs occupent toutes la même place en mémoire (2 ou 4 octets selon la machine). Les variables pointeur sont initialisées comme les autres variables. Si `x` est une variable de type `int` et `p_x` un pointeur associé au type `int`, l'expression `p_x=&x` a pour effet d'associer à `p_x` la valeur correspondant à l'adresse de `x`. Le programme suivant affiche trois fois l'adresse mémoire de la variable `x`.

```
#include <stdio.h>

int main ()
{
 int x, *p1, *p2;

 p1=&x;
 p2=p1;

 printf("l'adresse de x est %X \n",&x);
 printf("l'adresse de x est %X \n",p1);
 printf("l'adresse de x est %X \n",p2);

 return 0;
}
```

5.2 Accès direct aux variables

Pour accéder au contenu d'une adresse via un pointeur, on a besoin de l'opérateur d'indirection « * ». Ainsi, si `p` est un pointeur associé au type `t` et dont la valeur est `x`, `*p` est une expression de type `t` dont la valeur est la valeur mémorisée à l'adresse `x`. Cela est illustré par le programme suivant.

```
#include <stdio.h>

int main ()
{
 float x, *p;

 printf("entrez un nombre : ");
 scanf("%f",&x);

 x *= x;

 p=&x;

 printf("le carre de ce nombre vaut %f \n",*p);

 return 0;
}
```

Le programme suivant est équivalent au précédent. Ici, on associe d'abord `p` à l'adresse de `x`, puis on élève le contenu de l'adresse au carré. Puisque c'est la valeur de `x` qui est placée à cette adresse, cette opération affecte directement la valeur de `x`.

```
#include <stdio.h>

int main ()
{
 float x,*p;

 printf("entrez un nombre : ");
 scanf("%f",&x);

 p=&x;
 *p *= *p;

 printf("le carre de ce nombre vaut %f \n",x);

 return 0;
}
```

Enfin dans ce programme, au associe à `p` l'adresse de `x`, puis on mémorise la valeur entrée par l'utilisateur à l'adresse pointée par `p`, i.e. dans `x`. Remarquons que si on n'a pas préalablement associé à `p` l'adresse de `x`, une erreur peut se produire durant l'exécution : `p`

n'ayant pas été initialisé, le programme tentera de mémoriser la valeur entrée par l'utilisateur à un endroit indéfini. Ce genre de négligences (qui n'est pas une erreur sémantique), est source d'erreurs (dans le sens où le programme ne produira pas le résultat escompté) souvent difficiles à détecter.

```
#include <stdio.h>

int main ()
{
 float x,*p;

 p=&x;
 printf("entrez un nombre : ");
 scanf("%f",p);

 x *= x;
 printf("le carre de ce nombre vaut %f \n",*p);

 return 0;
}
```

La conversion de type pour les pointeurs peut être utilisée de la même manière que pour les autres variables. Dans le programme suivant, pour l'instruction `*p_q= (double) *p_n / *p_d`, il est nécessaire de spécifier qu'il faut considérer le numérateur comme une valeur de type `double`, sinon le résultat de la division sera de type `int`.

```
#include <stdio.h>

int main ()
{
  int n,d,*p_n,*p_d;
  double q,*p_q;

 printf("entrez le numerateur : ");
 scanf("%d",&n);

 printf("entrez le denominateur : ");
 scanf("%d",&d);

 p_n=&n;
 p_d=&d;
 p_q=&q;

 *p_q= (double) *p_n / *p_d;

 printf("%d/%d= %lf \n",n,d,*p_q);

 return 0;
}
```

5.3 Arithmétique des pointeurs

Pour manipuler les pointeurs, on dispose outre des opérations d'affectation, d'une série d'opérations que l'on regroupe sous l'appellation « *arithmétique des pointeurs* ».

Addition

Supposons que `p` soit un pointeur associé au type `t` et que ce type soit codé, en *C*, sur k octets. Si la valeur associée à `p` est l'adresse x, alors la valeur associée à `p+1` est l'adresse située k octets plus loin que l'octet situé à l'adresse x, i.e. l'adresse $x + k$. Ainsi, `p+1` ne désigne pas l'octet situé juste après celui désigné par la valeur de `p`. De manière générale, `p+n` représente l'adresse située $n \cdot k$ octets après l'adresse associée à `p`, $x + nk$ (et non $x + n$).

Seul l'addition d'un nombre entier à un pointeur est autorisée. En *C*, l'addition de deux pointeurs n'est pas définie.

Soustraction

Supposons que `p` soit un pointeur associé au type `t` et que ce type soit codé, en *C*, sur k octets. Si la valeur associée à `p` est l'adresse x, alors la valeur associée à `p-n` est l'adresse située $n \cdot k$ octets avant l'octet situé à l'adresse x, i.e. l'adresse $x - nk$.

Il est possible de soustraire un pointeur à un pointeur. Si `p1` et `p2` sont des pointeurs associé au type `t` et que ce type est codé, en *C*, sur k octets, `p2-p1` donne le nombre d'octets séparant les adresses associées aux pointeurs `p2` et `p1` divisé par k.

Comparaison

Des pointeurs de même type peuvent être comparés. En *C*, l'adresse 0 ne contient pas de donnée. On utilise en général cette adresse comme un indicateur (il arrive souvent qu'un pointeur associé à la valeur 0 signifie qu'une erreur est survenue). La valeur 0 n'est en général pas utilisée explicitement pour de telles comparaison ; on préfère utiliser la constante symbolique `NULL`, définie dans le fichier d'en-tête `stdio.h`. On peut donc initialiser un pointeur (appelé par exemple `p` et de type `int`) à zéro comme suit, `int *p = NULL`, ou affecter à tout moment dans le programme cette valeur au pointeur, `p=NULL`.

Chapitre 6

Types de données complexes

6.1 Tableaux

Un tableau permet de définir un vecteur dont les éléments sont des tableaux ou des variables de même type. Un *tableau* est la donnée d'un pointeur (typé) et d'un nombre naturel, correspondant au nombre d'éléments du tableau.

Tableaux unidimensionnels

Nous commencerons par traiter le cas où les éléments du tableau ne sont pas eux-mêmes des tableaux[1]. La définition d'un tel tableau admet la syntaxe suivante.

```
<type> <nom> [nombre];
```

le type des éléments dont est constitué le tableau est spécifié par `<type>` ; `nombre` est le nombre d'éléments du tableau. Les crochets ne signifie pas ici que cette valeur est optionnelle, il font partie de la syntaxe. Après la définitions, si le type `<type>` est mémorisé sur n octets, un bloc contigu de `nombre`$\cdot n$ octets sera réservé en mémoire et l'adresse de ce bloc sera placé dans le pointeur `<nom>`, associé au type `<type>`. Par exemple,

```
float x [5];
```

a pour effet de définir un tableau pouvant contenir cinq éléments de type `float`. En pratique, un bloc contigu de $5 \cdot 4 = 20$ octets (4 étant le nombre d'octets nécessaire pour coder une valeur de type `float`) est réservé en mémoire. L'adresse du début de ce bloc est affecté à la variable `x`, qui est un pointeur associé au type `float`. Remarquons que l'adresse associée à `x` est constante : elle ne pourra être modifiée dans le programme ; autrement dit `x` est un pointeur constant et n'est donc pas une lvalue. Nous verrons que l'allocation dynamique permet de modifier le nombre d'éléments d'un tableau et que, dans ce cas, le pointeur associé n'est pas constant.

À un tableau, on associe bien sûr son `nom`. Cependant, et contrairement aux variables, cela n'est pas suffisant pour accéder aux valeurs d'un tableau, puisqu'un tableau peut contenir plusieurs éléments. Ces éléments sont différenciés entre eux par leur indice. Le premier élément du tableau est associé à l'indice 0, le deuxième à l'indice 1 et le n-ième à l'indice $n-1$. Si `nom` est le nom d'un tableau contenant n éléments, `nom[0]` représente

[1] On parle de tableau unidimensionnel.

la valeur associée au premier élément de `nom`, `nom[1]` représente la valeur associée au deuxième élément de `nom` et `nom[n-1]` représente la valeur associée au dernier élément de `nom`. Voici un exemple, demandant à l'utilisateur d'entrer 5 nombres réels et calculant leur somme.

```
#include <stdio.h>

int main ()
{
  int j;
  float x [5];
  double somme;

  for( j=0 ; j<5 ; j++ )
  {
    printf("entrez le nombre numero %d : ",j+1);
    scanf("%f",&x[j]);
  }

  for( j=0 , somme=0 ; j<5 ; j++ )
    somme += x[j];

  printf("la somme des nombres que vous venez ");
  printf("d'entrer vaut %lf \n ",somme);

  return 0;
}
```

Il est important de noter que le nombre d'éléments d'un tableau doit être constant ; le compilateur doit connaître la taille du bloc mémoire qu'il doit allouer au tableau. Comme nous le verrons, il est cependant possible d'allouer de la mémoire de manière dynamique.

Il est possible d'initialiser les tableaux dès leur définition. Les valeurs doivent être des constantes séparées par des virgules et placées entre accolades,

```
<type> <nom> <nombre> = { x1, x2, ..., xnombre-1};
```

Si le nombre de constantes est inférieur au nombre d'éléments du tableau, les éléments non pris en considération sont initialisés avec la valeur 0. Si le nombre de constantes est supérieur au nombre d'éléments, le compilateur signalera une erreur. On peut renoncer à spécifier le nombre d'élément d'un tableau initialisé ; en effet, le compilateur supposera alors que le nombre d'élément du tableau est égal au nombre de constantes d'initialisation. Ainsi les expressions suivantes sont équivalentes.

```
int tab [5]={0, 0, 0, 0, 0};
int tab []={0, 0, 0, 0, 0};
int tab [5]={0};
```

Remarquons enfin que certains compilateurs anciens n'admettent pas ce type d'initialisation[2].

[2]En fait, pour ces compilateurs, on ne peut initialiser de cette manière les variables de classe `auto`, définies dans les sous-programmes.

Il est intéressant de noter que, puisque le nom d'un tableau représente un pointeur, on peut utiliser l'arithmétique des pointeurs avantageusement pour indexer les tableaux. Ainsi, le programme précédent peut être ré-écrit comme suit.

```
#include <stdio.h>

int main ()
{
  int j;
  float x [5];
  double somme;

  for( j=0 ; j<5 ; j++ )
  {
    printf("entrez le nombre numero %d : ",j+1);
    scanf("%f",x+j);
  }

  for( j=0 , somme=0 ; j<5 ; j++ )
    somme += *(x+j);

  printf("la somme des nombres que vous venez ");
  printf("d'entrer vaut %lf \n",somme);

  return 0;
}
```

ou encore sous la forme suivante, où le tableau `x` n'est pas explicitement utilisé dans l'affectation ou le calcul de la somme.

```
#include <stdio.h>

int main ()
{
  int j;
  float x [5], *px;
  double somme;

  for( j=0 , px=x ; j<5 ; j++, px++ )
  {
    printf("entrez le nombre numero %d : ",j+1);
    scanf("%f",px);
  }

  for( j=0 , px=x, somme=0 ; j<5 ; j++ , px++ )
    somme += *px;

  printf("la somme des nombres que vous venez ");
  printf("d'entrer vaut %lf \n",somme);
```

```
  return 0;
}
```

Enfin, remarquons que les effets de bords peuvent être source d'erreur. Dans le programme qui suit,

```
#include <stdio.h>

int main ()
{
  int x [5] = {0};
  int n = 2;

  x[n] = n--;

  return 0;
}
```

il est impossible de savoir si c'est le deuxième élément ou le premier qui va se voir attribuer la valeur 2. Cela dépend en fait du compilateur.

Tableaux à plusieurs dimensions

Si l'on souhaite mémoriser une matrice de type $m \times n$ (dont les éléments sont par exemple de type `float`) en mémoire, il peut paraître avantageux de pouvoir disposer de tableaux à double entrée, i.e. indexés par deux indices. De tels tableaux sont des tableaux dont les éléments sont des tableaux. En pratique, il suffit bien sûr de définir un tableau de $m \cdot n$ éléments et de placer les valeurs les unes à la suite des autres. Si c'est ainsi que sont mémorisés tous les tableaux en *C*, ce dernier offre néanmoins la possibilité d'avoir recours à plusieurs indices pour accéder aux éléments d'un tableau.

On peut définir un tableau à n dimensions en ayant recours à la syntaxe suivante.

```
<type> <nom> [e1] [e2] ... [en];
```

On accède, comme précédemment, aux éléments du tableau en mettant les indices entre crochets. Le programme suivant calcule, si possible, la matrice inverse d'une matrice de type 2×2.

```
#include <stdio.h>

int main ()
{
  double det, mat [2][2];
  int j,k;

  for (j=0; j<2; j++)
    for (k=0; k<2; k++)
    {
      printf("entrez l'element %d %d : ",k+1,j+1);
```

```
      scanf("%lf", &mat[k][j]);
    }

  det=mat[0][0]*mat[1][1]-mat[1][0]*mat[0][1];

  if (!det) printf("la matrice n'admet pas d'inverse \n");
  else
  {
    printf("l'inverse de la matrice est \n");
    for (j=0; j<2; j++)
      for (k=0; k<2; k++)
        mat[k][j] /= det;

    printf("%lf %lf \n",mat[1][1],-mat[0][1]);
    printf("%lf %lf \n",-mat[1][0],mat[0][0]);
  }

  return 0;
}
```

Comme nous l'avons déjà fait remarquer, les tableaux multidimensionnels n'existent pas physiquement : la mémoire correspondant à ces tableaux est réservée linéairement, comme pour un tableau unidimensionnel. Supposons que `tab` soit un tableau défini comme suit.

```
<type> tab [e1] [e2] ... [en];
```

Supposons en outre qu'une variable de type `<type>` soit codée sur m octets en mémoire. À l'initialisation, un bloc de $\texttt{e1}\cdot\texttt{e2}\cdot\ldots\cdot\texttt{en}\cdot m$ octets est réservé pour le tableau `tab`. L'élément `tab[i1][i2]...[in]` (où `ik` est compris entre 0 et $\texttt{ek}-1$) se trouve à l'emplacement mémoire

$$\texttt{x} + \texttt{i1}\cdot\texttt{e2}\cdot\ldots\cdot\texttt{en}\cdot m + \texttt{i2}\cdot\texttt{e3}\cdot\ldots\cdot\texttt{en}\cdot m + \ldots + \texttt{in}\cdot m.$$

Un tableau multidimensionnel peut être initialisé lors de sa définition de la même manière qu'un tableau unidimensionnel,

```
int k [2] [3] = {1, 2, 3, 4, 5, 6};
```

mais on peut aussi utiliser des accolades supplémentaires afin de rendre l'affectation précédente plus explicite ou ne pas initialiser explicitement certains éléments.

```
int k [2] [3] = {
                 {1, 2, 3},
                 {4, 5, 6}
                };
```

On peut, dans de telles initialisations, renoncer à la spécification du premier indice (et seulement du premier indice). Le compilateur se charge de déterminer la constante à utiliser.

```
int k [] [3] = {1, 2, 3, 4, 5, 6};
```

Chaînes de caractères

En *C*, une *chaîne de caractères* est un tableau associé au type `char` dont au moins un des éléments est le caractère nul `\0`, matérialisant la fin de la chaîne. Dans d'autres langages, la chaîne de caractères est un type à part entière. Le nombre entier associé à un élément de la chaîne représente un caractère, codé par la *table ASCII*.

A priori, il faut initialiser une chaîne de caractères élément par élément comme suit.

```
char s [8] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

ou

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

On peut cependant utiliser une syntaxe plus commode.

```
char s [8] = "bonjour";
```

ou

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

Ainsi, un tableau de trois chaînes de caractères peut se définir comme suit.

```
char s [3][6] = {"alpha", "beta", "gamma"};
```

Nous avons déjà vu que les fonctions `printf` et `scanf` prennent en charge les chaînes de caractère. Le programme suivant compte le nombre de caractères `a` ou `A` présents dans le nom entré par l'utilisateur.

```
#include <stdio.h>

int main ()
{
  char nom [128];
  int j,nb_a;

  printf ("entrez votre nom : ");
  scanf ("%s",nom);

  for(j=0, nb_a=0; nom[j]!='\0'; j++)
    if (nom[j]=='a' || nom[j]=='A') nb_a++;

  printf ("votre nom comporte %d lettre(s) 'a' \n",nb_a);

  return 0;
}
```

Allocation dynamique de tableaux

La fonction `malloc` alloue un bloc de la mémoire à un pointeur, de façon à ce que ce dernier puisse être utilisé comme un tableau. Plus précisément,

```
void * malloc ( size_t <size> );
```

alloue un bloc de `<size>` octets de la mémoire. En cas de succès, `malloc` retourne un pointeur vers le début du bloc nouvellement alloué. Le type de ce pointeur est toujours `void*` et peut être converti au type désiré. Si un bloc de la taille `size` ne peut être alloué, un pointeur nul est retourné. La fonction `malloc` se trouve dans la librairie standard `stdlib`. Le programme suivant détermine le plus grand élément d'une suite d'entiers entrés par l'utilisateur ; le nombre d'entiers est aussi demandé à l'utilisateur.

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

int main ()
{
  int *liste;
  int taille,j,p_gr;

  printf("nombre d'elements : ");
  scanf("%d",&taille);

  liste = malloc (taille*sizeof(int));

  if (liste==NULL) printf ("allocation impossible \n");
  else
  {
    for(j=0 ; j<taille ; j++)
    {
      printf ("element %d : ",j+1);
      scanf("%d",&liste[j]);
    }

    for(j=1, p_gr=0 ; j<taille ; j++)
      if (liste[j]>liste[p_gr]) p_gr=j;

    printf("le plus grand element est le ");
    printf("numero %d et vaut %d \n",p_gr+1,liste[p_gr]);
  }

  return 0;
}
```

Remarquons que la première instruction `if` teste si la mémoire a bien été allouée. Les instructions

```
  liste = malloc (taille*sizeof(int));
```

```
if (liste==NULL) printf ("allocation impossible \n");
```

peuvent être remplacée de manière à raccourcir le programme comme suit,

```
if ( (liste= malloc (taille*sizeof(int))) == NULL)
  printf ("allocation impossible \n");
```

Dans le programme précédent, la fonction `malloc` réserve un bloc mémoire de taille `taille*sizeof(int)`, soit la taille nécessaire pour stocker un nombre de variables de type `int` égal à `taille`. L'adresse du début du bloc est placé dans la variable `liste`. On vient donc de définir un tableau d'entiers. Il est cependant important de remarquer que cette manière de procéder est différente de celle que nous avons vue précédemment. D'abord, l'instruction

```
int liste [taille];
```

n'est pas correcte si `taille` n'est pas une constante. On pourrait imaginer de remplacer la variable `taille` par une constante suffisamment grande.

```
int liste [1024];
```

Outre le fait que cette manière de procéder introduit une limitation dans le nombre d'entiers que peut entrer l'utilisateur, rappelons que le pointeur `liste` est ici une constante. Une instruction telle que `liste++` produira une erreur. Il n'en va pas de même pour le tableau dynamique défini à l'aide de la fonction `malloc`.

Un bloc mémoire alloué peut être libéré grâce à la fonction `free`, qui sera décrite par la suite.

6.2 Structures

Une *variable structurée* est une variable composée de variables de types différents, appelées *champs*. Pour créer une variable structurée, il convient d'abord de déclarer les champs la constituant ; on parle de déclaration de la *structure*. La syntaxe est la suivante.

```
struct <nom_struct>
{
  <type_champ1> <nom_champ1>;
  <type_champ2> <nom_champ2>;
                .
                .
                .
  <type_champn> <nom_champn>;
};
```

Après cette déclaration, des variables de type `nom_struct` pourront être définies. Les champs peuvent être de n'importe quel type, à une exception près : le type d'un champ ne peut pas être celui de la structure contenant le champ. Ici, `<nom_champk>` ne peut être de type `<nom_struct>`. Le type d'un champ peut en revanche être un pointeur associé au type de la structure le contenant. Par exemple, `<nom_champk>` peut être de type

`*<nom_struct>`. Lors d'une déclaration de structure aucun bloc mémoire n'est alloué ; la mémoire est allouée lors de la déclaration d'une variable du type défini par la structure.

Supposons que les caractéristiques types d'un étudiant que l'on souhaite traiter soient les suivantes : le nom, le prénom, l'âge, la taille et l'année d'étude. La structure associée pourrait être celle-ci.

```
struct etudiant
{
  char nom [64];
  char prenom [64];
  short age;
  short annee;
};
```

On peut, après cette déclaration, définir des variables de type `etudiant`. Par exemple, l'instruction

```
struct etudiant e1,e2,e3;
```

définit trois variables, `e1`, `e2` et `e3`, de type `etudiant`. C'est à ce moment que de la mémoire est allouée pour chacune variable. Une variable de type `short` est codée sur deux octets et les variables chaîne de caractères définies dans la structure occupent chacune 64 octets. Une variable de type `etudiant` occupera donc $2 \cdot 2 + 2 \cdot 64 = 132$ octets en mémoire.

On peut aussi définir des variables structurées en les définissant directement après l'accolade fermante de la déclaration de la structure. Ainsi les variables de type `etudiant` introduites plus haut auraient pu être déclarée comme suit.

```
struct etudiant
{
  char nom [64];
  char prenom [64];
  short age;
  short annee;
} e1, e2, e3;
```

Cette déclaration est bien-sûr plus compacte. Le nom de la structure peut être omis dans une telle déclaration. Il est cependant bien évident que dans ce cas, il ne pourra y avoir aucune autre déclaration de variable structurée de ce type dans le programme (puisqu'aucun nom ne sera affecté au type). La déclaration précédente peut donc s'écrire

```
struct
{
  char nom [64];
  char prenom [64];
  short age;
  short annee;
} e1, e2, e3;
```

Comme pour les tableau, il nous faut encore décrire la manière dont on peut accéder aux champs d'une variable structurée. Ici, puisque les champs peuvent être de types différents, le crochet n'est pas de mise. On accède aux champs par l'opérateur de champ « . ». Cet opérateur se place entre le nom de la variable structurée et celui du champ.

```
<nom_variable>.<nom_champ>
```

Ainsi, les opérations suivantes sont licites sur les variables `e1`, `e2` et `e3`.

```
printf("le nom du second etudiant est %s \n",e2.nom);
e1.age=19;
e3.age=e1.age+1;
```

L'affectation lapidaire, faisant intervenir l'opérateur « = » est permise entre structures. Donc `e1=e2 ;` est permis. Par contre on ne peut comparer directement deux variables structurées ; il faut comparer chaque champ séparément. L'opérateur de moulage `(cast)` ne peut non plus être utilisé sur une variable structurée. Les opérateurs `&` et `sizeof` sont en revanche permis.

Les variables structurées, comme les autres variables, peuvent être initialisées lors de leur définition. L'initialisation de variables structurées est semblable à celle des tableaux, avec les mêmes restrictions. Par exemple, en reprenant l'exemple de la structure `etudiant`,

```
struct etudiant e1 = {"Dupont", "Jean", 19, 2};
```

initialisera les champs de la variable `e1`, en suivant l'ordre donné dans la structure. Remarquons qu'il est bien entendu possible de définir un tableau structuré. Par exemple on peut définir un tableau d'étudiants comme suit.

```
struct etudiant e [25];
```

Une *union* permet de définir un type de la même manière qu'une structure (on remplace simplement le mot clef `struct` par `union`). La différence entre une union et une structure est que, dans une union, un seul des champs est accessible à la fois. Si une union possède deux champs et que, dans une variable définie à partir de cette union, une valeur est affectée au deuxième champ, la valeur attribuée au premier sera perdue. En fait dans une union, les champs partagent le même espace mémoire (un bloc correspondant au type le plus volumineux en mémoire est réservé lors de la déclaration d'une variable correspondante). Il en résulte un gain de place en mémoire.

Il est aussi possible de définir des structures spéciales, où les champs correspondent à un nombre de bits. La définition d'une telle structure est fort similaire à la définition traditionnelle.

```
struct <nom_struct>
{
  <type_champ1> <nom_champ1> : <nombre_bits1>;
  <type_champ2> <nom_champ2> : <nombre_bits2>;
                .
                .
                .
  <type_champn> <nom_champn> : <nombre_bitsn>;
};
```

Seuls les types `int` et `unsigned int` sont autorisés. La donnée entière `<nombre_bitsk>` correspond aux nombres de bits qu'il faut réserver au champ correspondant.

Chapitre 7

Fonctions

7.1 Sous-programmes

Un sous-programme est une programme dont l'exécution peut être commandée par un autre programme. Dans la plupart des langages de programmation, deux ordres sont, au moins implicitement, présents dans la rédaction d'un sous-programme : l'*appel* et le *retour*. L'appel suspend l'exécution du programme appelant le sous-programme, afin de donner la main à ce dernier. Lorsque le sous-programme se termine, il provoque un ordre de retour à l'instruction du programme appelant qui suit immédiatement l'ordre d'appel. Dans certains langages, l'appel à un sous-programme se fait via un mot réservé ; en *C*, c'est le nom du sous-programme qui provoque l'appel à celui-ci. De même, le retour est souvent commandé par un mot réservé (`return` en *C*). Si un type est associé au sous-programme, ce sous-programme est appelé une *fonction*, sinon, on parle de *procédure*. Le *C* ne fait pas directement la distinction entre fonction et procédure : une procédure est une fonction dont le type est `void`. Il en résulte que la manière de rédiger une procédure en *C* est strictement analogue à la rédaction d'une fonction.

7.2 Définition d'une fonction

Pour définir une fonction en *C*, il faut coder les instructions qu'elle doit exécuter en respectant certaines règles syntaxiques. Il convient de spécifier les informations suivantes.

- La classe de mémorisation de la fonction,
- le type de valeur renvoyé par la fonction, i.e. le type associé à la fonction,
- le nom de la fonction,
- les paramètres (arguments) qui doivent être passés à la fonction,
- les variables locales et externes utilisées par la fonction,
- les autres fonctions appelée par la fonction,
- les instructions que doit exécuter la fonction.

Il existe deux syntaxes permettant de définir une fonction ; nous présentons ici[1] celle correspondant au standard *ANSI*. La définition se fait comme suit.

```
[<classe de memoire>] [<type>] nom ([<type1> <var1>, <type2> <var2>,...])
{
```

[1] L'autre méthode, plus ancienne, est à proscrire, mais peut parfois être rencontrée dans les programmes les plus anciens.

```
  [<definition des variables locales>]
  [<definition des variables externes>]
  [<declaration des fonctions pouvant etre appelees>]

  [instructions]
 }
```

La première ligne de la définition est appelée l'*en-tête*. Le reste est appelé le *bloc* de la fonction. Une définition de fonction ne peut contenir une définition de fonction.

Il existe deux classes de mémoire : la classe `extern` et la classe `static`. Si la classe n'est pas précisée la fonction est associée à la classe `extern`. Si la fonction est de classe `static`, elle ne pourra être utilisée dans un autre module que celui où elle est définie ; nous n'utiliserons ici que la classe `extern`. L'instruction[2] `return` termine la fonction en mettant fin à l'exécution des instructions d'une fonction et rend le contrôle au programme ayant fait appel à cette fonction. Cette instruction peut renvoyer une valeur au programme appelant via la syntaxe

```
 return (<expression>);
```

ou

```
 return <expression>;
```

La fonction peut alors être considérée comme une opérande et intervenir dans une expression. Le type de cet opérande est celui donné en en-tête. Le compilateur convertit éventuellement la valeur de l'expression présente dans l'instruction `return` pour qu'elle corresponde au type de la fonction. Si aucun type n'est donné dans l'en-tête de la fonction, elle est considérée comme étant de type `int`. Si aucune expression n'est stipulée après l'instruction `return` ou si une accolade « `}` » de fin de sous-programme est rencontrée, c'est-à-dire si l'instruction `return` est omise, la valeur de la fonction est indéfinie.

Un *paramètre* est une variable dont la valeur associée est définie par le programme appelant. La valeur est appelée *paramètre effectif*, par opposition à la variable elle-même, qui est appelée *paramètre formel*. Les paramètres sont déclarés entre parenthèses dans l'en-tête. Si la fonction ne nécessite aucun paramètre, on peut, dans l'en-tête, ne rien mettre entre les parenthèses en écrivant « `()` », ou préciser explicitement l'absence de paramètre en utilisant le mot clef `void`, i.e. en écrivant « `(void)` ». L'appel à une fonction sans paramètre se fait toujours en utilisant les parenthèses vides « `()` », et non avec l'instruction `void`.

Considérons un premier exemple de fonction.

```
double cube (double x)
{
 return (x*x*x);
}
```

La fonction nommée `cube` est de type `double` et possède ici un paramètre, `x`, de type `double`. Supposons que le programme appelle la fonction `cube` avec, comme paramètre effectif, la valeur $2,1$. La fonction `cube` va calculer la valeur de $2,1^3$ via l'expression `x*x*x` et la retourner au programme appelant grâce à l'instruction `return`. Ainsi le programme

[2] Il s'agit bien d'une instruction et non d'une fonction ; les parenthèses accompagnant généralement l'instruction `return` ne sont que des conventions syntaxiques.

```
#include <stdio.h>

double cube (double x)
{
 return (x*x*x);
}

int main ()
{

 printf("2.1 au cube vaut %lf \n",cube(2.1));

 return 0;
}
```

affichera

```
2.1 au cube vaut 9.261
```

à la sortie standard.

On comprends maintenant que le mot `main`, apparaissant dans les programmes *C* est en fait une fonction particulière : c'est la fonction qui est exécutée lorsque le programme est lancé par l'utilisateur. Elle est ici de type `int` et ne prend pas de paramètre. Après l'affichage, la fonction `main` et donc le programme en entier, se termine en renvoyant la valeur 0 à l'environnement ayant lancé le programme. Le programme précédent peut être modifié pour calculer le cube de n'importe quel valeur (de type `double`) entrée par l'utilisateur.

```
#include <stdio.h>

double cube (double x)
{
 return (x*x*x);
}

int main ()
{
 double x;

 printf("entrez une valeur : ");
 scanf("%lf",&x);

 printf("%lf au cube vaut %lf \n",x,cube(x));

 return 0;
}
```

Ce programme pourrait a priori poser problème. En effet, la fonction `main` utilise une variable appelée `x`, tout comme la fonction `cube`. En fait, il n'en est rien. Les fonctions

ne partagent pas les variables. En d'autres termes, les variables, même si elles portent le même nom, sont associées à des adresses mémoires distinctes et une fonction ne peut avoir accès aux variables d'une autre fonction. On dit que les variables sont *locales*. Ainsi, le compilateur ne peut se méprendre et confondre les variables. Voici un autre exemple où la somme des entiers compris entre 1 et un nombre entré par l'utilisateur est calculée et affichée.

```
#include <stdio.h>

long sommeS (int bs)
{
 long somme
 int j;

 for(j=1 , somme=0 ; j<=bs ; j++) somme +=j;

 return somme;
}

int main ()
{
 int x;

 printf("entrez une valeur : ");
 scanf("%d",&x);

 printf("la somme des entiers compris entre 1 et %d",x);
 printf(" vaut %ld \n",sommeS(x));

 return 0;
}
```

Après l'exécution de la fonction `sommeS`, i.e. lorsque l'instruction `return` est rencontrée dans cette fonction, la main est rendue à la fonction `main`. Les variables `bs`, `j` et `somme` n'existent plus. Les valeurs associées à ces variables sont d'ailleurs irrémédiablement perdues : si la fonction `sommeS` était appelée une seconde fois, les valeurs associées aux variables après leur déclaration ne seraient pas nécessairement les mêmes que celles à la fin du premier appel.

7.3 Déclaration de fonctions

Nous avons déjà précisé que la définition d'une fonction doit être globale, c'est-à-dire ne pas se faire dans une fonction. En outre, un programme ne peut appeler une fonction si elle n'a pas encore été définie, sans occasionner des problèmes. Dans les programmes précédents, les fonctions `cube` et `sommeS` sont définies avant la fonction `main`. Cette dernière peut donc faire appel aux deux premières sans problème. Si, par exemple, la fonction `cube` avait été définie après la fonction `main`, une erreur à la compilation se serait produite. En effet, dans ce cas, la fonction `main` fait appel à une fonction `cube` non encore définie.

Le compilateur extrapole alors le type de cette fonction : il suppose que `cube` est de type `int`. La définition de la fonction `cube`, de type `double`, à la suite de la fonction `main` est incompatible avec les informations que leur compilateur possède déjà, à savoir que la fonction `cube` est de type `int`. Le compilateur déclarera donc qu'une erreur de redéfinition s'est produite. Les erreurs de déclaration peuvent être difficiles à détecter.

Il est possible de faire appel à une fonction avant qu'elle ne soit définie ; il est cependant nécessaire de fournir au compilateur les informations nécessaires via une déclaration syntaxique particulière, la *déclaration de fonction*. Une déclaration de fonction fournit les informations sur la classe, le type et le nom de la fonction ainsi que sur le type des éventuels paramètres, mais ne crée pas de nouvelle entité. Il existe plusieurs syntaxes pour la déclaration de fonctions (parfois appelé un *prototype*), dont deux selon le standard *ANSI*.

```
[<classe>] [<type>] nom ([<type1> <var1>, <type2> <var2>,...]);
```

ou

```
[<classe>] [<type>] nom ([<type1>, <type2>,...]);
```

Si fonction n'admet pas de paramètre, il convient d'utiliser le mot clef `void` et non les parenthèses vides. Dans ce dernier cas, le compilateur ne procéderait pas à une vérification syntaxique. Le programme calculant le cube d'un nombre peut être ré-écrit comme suit.

```
#include <stdio.h>

double cube (double);

int main ()
{
 double x;

 printf("entrez une valeur : ");
 scanf("%lf",&x);

 printf("%lf au cube vaut %lf \n",x,cube(x));

 return 0;
}

double cube (double x)
{
 return (x*x*x);
}
```

La fonction `cube` a été déclarée globalement : toutes les fonctions (dont `main`) qui suivent la déclaration de la fonction `cube` peuvent y faire appel. On peut aussi déclarer une fonction dans une autre fonction ; on parle de déclaration locale. Par exemple, le programme suivant est équivalent au précédent.

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

```
int main ()
{
 double cube (double);

 double x;

 printf("entrez une valeur : ");
 scanf("%lf",&x);

 printf("%lf au cube vaut %lf \n",x,cube(x));

 return 0;
}

double cube (double x)
{
 return (x*x*x);
}
```

Cependant, dans ce dernier cas, une autre fonction que `main` ne pourra faire appel à `cube` (en tant que fonction de type `double`) : si une fonction `f1` est déclarée localement dans une fonction `f2`, seule cette dernière pourra faire appel à `f1`. Il est bien sûr permis de faire plusieurs déclarations locales (c'est-à-dire de déclarer `f1` dans plusieurs fonctions).

Il est possible de partitionner un programme ; ainsi on pourrait imaginer rédiger les fonctions `cube` et `main` du programme précédent dans des fichiers séparés. Il est nécessaire dans ce cas de préciser au compilateur qu'il faut prendre en compte deux fichiers et non un seul. Si une fonction `f1` est déclarée dans un fichier et qu'une fonction `f2`, déclarée dans un autre fichier, doit faire appel à `f1`, cette fonction doit être déclarée dans le même fichier que `f2`, soit globalement, soit localement dans `f2`. La déclaration d'une fonction ne doit pas nécessairement être incluse explicitement dans le texte source ; la déclaration peut être placée dans un fichier d'en-tête, se terminant généralement avec l'extension « .h ». On fait appel à ce fichier d'en-tête grâce à la directive `include`. Ainsi, le fichier d'en-tête `stdio.h` contient les déclarations des fonctions `printf` et `scanf` notamment. Les corps des sous-programmes `printf` et `scanf` sont logés dans des *bibliothèques* auxquelles l'utilisateur peut faire appel. Nous sommes maintenant en mesure de comprendre chaque lignes apparaissant dans les programmes rencontrés jusqu'ici.

7.4 Passage de paramètres

Un programme peut communiquer avec un de ses sous-programmes. D'une part le programme peut transmettre des paramètres au sous-programme, mais il peut aussi recevoir une valeur du sous-programme, si celui-ci est une fonction. En fait cette valeur est aussi associée à un type et la fonction peut être utilisée comme opérande. Il n'est donc pas envisageable de définir une fonction qui pourrait retourner plusieurs valeurs, associées, ou non, à différents types. Ainsi, si l'on peut passer plusieurs paramètres à une fonction, celle-ci ne peut, en retour, communiquer qu'une seule valeur associée à un type. En revanche, la plupart des langages de haut niveau peuvent permettre à un sous-programme, via une syntaxe particulière, de modifier la valeur des variables passées en paramètre.

Ces variables sont donc modifiées après l'appel au sous-programme, ce qui lui permet de communiquer d'une autre manière avec le programme appelant.

En général, par défaut, les valeurs qui peuvent être passées à un sous-programme comme paramètres peuvent l'être via des opérandes (notamment des constantes ou des variables). Si ce sont des variables qui sont passées en paramètre, elles ne sont pas modifiées après l'appel au sous programme : c'est la valeur associée à l'opérande qui est passée en paramètre et qui est affectée à une variable locale du sous-programme (le compilateur se charge de vérifier la concordance des types). On parle de passage par *valeur*. Mais il est possible de définir le sous-programme de telle manière que ce soit la variable qui soit passée en paramètre et non la valeur de la variable. La valeur associée à cette variable peut ainsi être modifiée par le sous-programme. On parle de passage par *variable*.

Contrairement à d'autre langages de programmation (*Cobol*, *Pascal*,...), le passage par valeur ou par variable n'est pas précisé de manière syntaxique en *C*. En *C*, il n'y a pas, à proprement parler, de passage par variable ; pour obtenir un résultat analogue, on utilise les pointeurs ; on parle de passage par *adresse*. C'est d'ailleurs selon ce mécanisme que s'effectue la passage par variable dans d'autres langages. Si l'on souhaite qu'une fonction puisse modifier la valeur d'une variable du programme appelant, il suffit de passer en paramètre non pas la variable, mais l'adresse de la variable. La fonction connaîtra ainsi l'adresse mémoire où est placée la valeur de la variable. Si la fonction modifie la valeur placée à cette adresse, c'est, par définition, la valeur de la variable qu'elle modifie. Cette modification persistera donc après l'exécution de la fonction. À titre d'exemple, considérons le programme suivant, qui fait appel à la fonction `swap`. Cette fonction prend comme argument l'adresse de deux variables de type `double` et échange le contenu de ces deux variable.

```
#include <stdio.h>

void swap (double *p_v1, double *p_v2)
{
 double val_swap;

 val_swap=*p_v1;
 *p_v1=*p_v2;
 *p_v2=val_swap;
}

int main ()
{
 double x,y;


 printf("entrez une valeur : ");
 scanf("%lf",&x);
 printf("entrez une autre valeur : ");
 scanf("%lf",&y);

 swap(&x,&y);
```

```
 printf("les valeurs echangees : %lf %lf \n",x,y);

 return 0;
}
```

Le passage d'un tableau comme paramètre mérite une attention particulière. Nous avons vu qu'en définissant un tableau, on définissait un pointeur et réservait une taille de bloc mémoire. Passer un tableau en paramètre consiste en fait à passer le pointeur en paramètre ; il s'agit donc d'un passage par adresse. Dans la fonction, c'est donc le tableau qui sera directement modifié et non une copie de celui-ci. La fonction `cp_tab` copie le tableau associé au type `int` passé comme premier paramètre dans le tableau passé en second paramètre. Le troisième paramètre est la taille du tableau.

```
#include <stdio.h>

void cp_tab (int *t1, int *t2, int taille)
{
  int j;
  for (j=0; j<taille; j++)
    t2[j] = t1[j];
}

int main ()
{
 int x[10] = {1,2,3,4,5,6,7,8,9,10}, y[10], j;

 cp_tab(x,y,10);

 for(j=0; j<10; j++)
   printf("%d \n",y[j]);

 return 0;
}
```

Remarquons que dans la fonction `cp_tab`, `t1` et `t2` ne sont pas des pointeurs constants, contrairement aux pointeurs `x` et `y` de la fonction `main`. Dans la déclaration des paramètres formels d'une fonction, `<type> *<nom>` peut être remplacé par `<type> <nom>[]`. Ainsi, dans la déclaration de la fonction `cp_tab`, on peut écrire

```
void cp_tab (int t1 [], int t2 [], int taille)
```

7.5 Fonctions récursives

D'une manière générale, la définition d'un objet est *récursive* lorsqu'elle contient une référence à cet objet. Dans les langages évolués, les sous-programmes peuvent s'appeler eux-mêmes, i.e. sont récursifs. Si on n'y prend pas garde, une fonction récursive risque de ne jamais s'arrêter. Il est des caractéristiques que doit respecter un algorithme récursif :

1. il doit dépendre de paramètres,

2. chaque appel récursif se fait avec des valeurs de paramètres différentes de celles qui ont été reçues,
3. pour certaines valeurs particulières des paramètres, il n'y a pas d'appel récursif.

Ces conditions ne sont pas suffisantes pour assurer qu'un algorithme récursif s'arrête. Il semble en général plus facile de démontrer la validité d'un algorithme s'il est écrit de manière récursive. Le principal inconvénient des programmes récursif est souvent la quantité de mémoire nécessaire.

Chapitre 8

Description de quelques fonctions

8.1 Allocation dynamique

La fonction `free`

La fonction `free` libère un bloc mémoire alloué grâce à la fonction `malloc`, `calloc` ou `realloc`,

```
void free ( void *<ptr> );
```

Le bloc mémoire désigné par le pointeur `<ptr>` (et alloué par une des fonctions `malloc`, `calloc` ou `realloc`) est libéré. La valeur de `<ptr>` n'est pas modifiée ; si `<ptr>` pointe sur l'adresse `NULL`, aucune action n'est effectuée.

Le programme suivante demande d'abord à l'utilisateur d'entrer un nombre entier n, puis de rentrer une suite de n nombres entiers avant de déterminer le plus grand de ces nombres. Ensuite le programme recommence, en demandant à nouveau à l'utilisateur d'entrer un nombre entier n, puis une suite de n nombres entiers. Le programme s'arrête lorsque l'allocation n'a pu se faire, notamment lorsque le nombre entré n est négatif.

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

int main ()
{
  int *liste;
  int taille,j,p_gr;

  do
  {
    printf("nombre d'elements (entrez un nombre ");
    printf("negatif pour terminer) : ");
    scanf("%d",&taille);

    liste = malloc (taille*sizeof(int));

    if (liste==NULL) printf ("allocation impossible \n");
    else
```

```
    {
      for(j=0 ; j<taille ; j++)
      {
        printf ("element %d : ",j+1);
        scanf("%d",&liste[j]);
      }

      for(j=1, p_gr=0 ; j<taille ; j++)
        if (liste[j]>liste[p_gr]) p_gr=j;

      printf("le plus grand element est le ");
      printf("numero %d et vaut %d \n",p_gr+1,liste[p_gr]);
    }
  } while(liste!=NULL);

    free (liste);
    return 0;
}
```

Avant de pouvoir réallouler de la mémoire pour le nouveau tableau dynamique `liste`, il faut d'abord libérer la mémoire occupée par ce tableau avec les anciennes valeurs, maintenant inutiles.

8.2 Fonctions mathématiques

Les fonctions mathématiques se trouvent dans la librairie `math`. Sauf mention contraire, les fonctions et les arguments des fonctions seront associés au type `double`.

Valeur absolue et partie entière

La fonction `fabs` retourne la valeur absolue de l'argument, la fonction `floor` retourne la partie entière de l'argument et la fonction `ceil` le plus grand entier supérieur à l'argument. La fonction `modf` est plus complexe ; elle permet de séparer la partie entière de la partie fractionnaire. Le premier argument est le nombre dont on veut séparer les parties entière et fractionnaire. Le second argument est un pointeur associé au type `double` ; après l'appel à la fonction, cette variable pointera sur l'adresse de la partie entière du premier argument. La fonction retourne la partie fractionnaire du premier argument, avec le même signe.

```
double modf ( double x, double * intpart );
```

Remarquons que la fonction équivalente à la fonction `fabs`, mais pour le type `long`, se nomme `labs`.

Fonction circulaires et hyperboliques

Les fonctions `cos`, `sin`, `tan`, `cosh`, `sinh` et `tanh`, ainsi que les fonctions inverses `acos`, `asin` et `atan` sont définies en *C*. La fonction `atan2` prend deux arguments de type `double`, `y` et x et retourne la valeur de $\text{arctg}(y/x)$.

Fonctions racine, puissance, logarithme et exponentielle

La fonction `sqrt` retourne la racine carrée de l'argument. La fonction `pow`,

```
double pow ( double <base>, double <exponent> );
```

retourne le nombre $<$ `base` $>^{<\texttt{exponent}>}$. La fonction `log` retourne la valeur du logarithme naturel de l'argument ; la fonction `log10` retourne le logarithme en base 10 de l'argument. La fonction `exp` retourne l'exponentielle de l'argument.

La valeur HUGE_VAL

Une fonction (`cosh`, ...) retourne la valeur `HUGE_VAL` pour indiquer que la valeur que devrait prendre la fonction n'est pas représentable dans le type qui lui est associé. Une fonction peut retourne une valeur `HUGE_VAL` positive ou négative (`HUGE_VAL` ou `-HUGE_VAL`).

8.3 Gestion de fichiers

Ouverture de fichiers

Le type `FILE` est une structure définie dans l'en-tête `stdio.h` et permet de mémoriser les informations relatives à un fichier. Pour accéder concrètement à un fichier, il faut utiliser un pointeur vers une variable de type `FILE`.

Avant qu'un programme ne puisse manipuler un fichier, il faut commencer par l'ouvrir. Dans la gestion de fichiers de haut niveau, on utilise la fonction `fopen`,

```
FILE * fopen ( char *<nom>, char *<mode>);
```

pour ouvrir le fichier nommé `<nom>`. Cette fonction retourne un pointeur associé au type `FILE`, qui pointe sur les informations concernant le fichier `<nom>`. En cas d'insuccès, le pointeur `NULL` est retourné. Un fichier peut être ouvert de plusieurs manières : on peut par exemple ne souhaiter que lire les données qu'il contient, ou au contraire effacer toutes les données éventuelles pour en écrire de nouvelles. La manière dont on souhaite accéder au fichier est précisée par `<mode>`.

mode d'accès	effet
`"r"`	ouvre le fichier existant en lecture le pointeur `NULL` est retourné si le fichier n'existe pas
`"w"`	ouvre le fichier en écriture le fichier est créé s'il n'existe pas s'il existait, son contenu est effacé
`"a"`	ouvre le fichier en écriture afin d'ajouter des données si le fichier n'existe pas, il est créé
`"r+"`	ouvre le fichier en lecture et en écriture le pointeur `NULL` est retourné si le fichier n'existes pas
`"w+"`	ouvre le fichier en lecture et en écriture le fichier est créé s'il n'existe pas s'il existait, son contenu est effacé
`"a+"`	ouvre le fichier en lecture et en écriture afin d'ajouter des données le fichier est créé s'il n'existe pas

Fermeture de fichiers

Lorsque les opérations à effectuer sur un fichier sont terminées, il vaut mieux le fermer. Il est impossible d'accéder à un fichier précédemment fermé, sauf bien sûr en le ré-ouvrant. Le prototype de la fonction est le suivant.

```
int fclose ( FILE *<fichier>);
```

La valeur 0 est retournée si le fichier a pu être correctement fermé. Dans le cas contraire, c'est la valeur `EOF` qui est retournée.

Fichier et périphérique standard

En *C*, cinq fichiers de périphérique sont habituellement ouverts lorsqu'un programme est exécuté. À chacun de ces cinq fichiers spéciaux est associé un pointeur `FILE` ; ce sont des pointeurs constants.

pointeur	fichier de périphérique associé
`stdin`	entrée standard
`stdout`	sortie standard
`stderr`	sortie standard pour les erreurs
`stdaux`	fichier d'entrée/sortie auxiliaire
`stdprn`	imprimante standard

Nous avons déjà vu que la fonction `printf` est associée au pointeur `stdout` et que la fonction `scanf` est associée au pointeur `stdin`.

Écriture formatée dans un fichier

La fonction `fprintf` est analogue à la fonction `printf`, à ceci près que la sortie n'est plus nécessairement la sortie standard, mais est déterminée par un pointeur associé au type `FILE`. Le prototype est le suivant.

```
int fprintf ( FILE *<pointeur> , char *<format>, ...);
```

Par rapport à la fonction `printf`, la fonction `fprintf` admet un premier argument supplémentaire. La fonction `fprintf` écrit dans le fichier désigné par la variable `<pointeur>`. Si `<pointeur>` est la constante `stdout`, la fonction `fprintf` est équivalente à la fonction `printf`.

Lecture formatée dans un fichier

La fonction `fscanf` est analogue à la fonction `scanf`, à ceci près que l'entrée n'est plus nécessairement l'entrée standard, mais est déterminée par un pointeur associé au type `FILE`. Le prototype est le suivant.

```
int fscanf ( FILE *<pointeur> , char *<format>, ...);
```

Par rapport à la fonction `scanf`, la fonction `fscanf` admet un premier argument supplémentaire. La fonction `fscanf` lit dans un fichier désigné par la variable `<pointeur>`. Si `<pointeur>` est la constante `stdin`, la fonction `fscanf` est équivalente à la fonction `scanf`.

Test de fin de fichier

La fonction `feof` permet de savoir si la fin d'un fichier (après une lecture par exemple) a été atteinte. Le prototype est le suivant.

```
int feof ( FILE *<pointeur> );
```

Cette fonction renvoie une valeur non-nulle si la fin du fichier désigné par `<pointeur>` a été atteinte. Sinon, la fonction `feof` renvoie la valeur 0.