LANGAGE C

AVANT-PROPOS

Le langage C est un langage de programmation dont la popularité, déjà acquise aux États-Unis, ne cesse de s'étendre en Europe et notamment en France. Cet engouement s'explique dans la mesure où l'existence du langage C est directement liée à celle d'un autre phénomène de l'informatique des années 80 : le système UNIX. En effet, C est le langage natif d'UNIX et constitue son axe central.*

La large diffusion du système UNIX en 1974 par les Laboratoires de la Bell Telephone auprès des Universités Américaines qui l'avaient choisi comme l'outil le mieux adapté à l'enseignement de l'informatique, a été, pour lui et par voie de conséquence pour le langage C, un véritable tremplin dans le monde de l'industrie informatique. Pendant cette période de validation, UNIX et C ont profité d'une constante auto-amélioration qui leur a permis de se bonifier sur le plan de la souplesse, de la fiabilité et de gagner en puissance.

La nouvelle vague d'informaticiens ainsi formée a contribué au succès et à l'expansion d'UNIX et de son langage.

En 1976, UNIX devient un produit fini digne de ce nom disponible sur le marché.

L'adoption récente d'UNIX par de nombreux constructeurs permet de dire aujourd'hui qu'UNIX est devenu un véritable standard pour les micros et mini-ordinateurs à mots de 16 ou 32 bits. Tout comme CP/M est devenu le système standard de fait pour les micro-ordinateurs à mots de 8 bits.

Cependant, si C est indispensable à UNIX, il reste un langage indépendant de l'environnement système. On assiste en effet, depuis ces dernières années à une poussée d'indépendance de C par rapport à UNIX. L'expérience concluante du couple UNIX-C et la portabilité du langage C, ont amené de nombreuses Sociétés, constructeurs et producteurs de logiciels, à le choisir comme principal outil de développement. Les implémentations de compilateurs C sur d'autres systèmes d'exploitation tels que CP/M-80, CP/M-86, MS-DOS, OASIS, VMS, etc., apportent la preuve de la vitalité du langage C.

---

* UNIX est une marque déposée par les Laboratoires BELL.

Le succès du langage C est certainement lié à celui d'UNIX, mais aussi à lui-même. Le langage C, par la richesse de son jeu d'opérateurs, par la souplesse d'utilisation des pointeurs et des fonctions, ont fait de lui un langage parfaitement adapté pour le développement de logiciels de base et d'application.

Le paradoxe de C consiste dans le fait qu'il soit à la fois un langage structuré de haut niveau et un langage relativement proche de l'architecture actuelle des machines, mais non d'une machine particulière.

L'étude du langage C comme pour tout autre langage peut se faire à partir des règles syntaxiques qui définissent le langage, sous la forme de notations BNF ou de graphes. Cette approche est souvent rébarbative et s'adresse à des spécialistes de langages. Dans le cadre de notre étude nous ne choisirons pas cette optique et nous tenterons d'avancer progressivement, non sans quelques petits problèmes de références en avant ou de renvois en arrière.

Cet ouvrage est découpé en six parties principales, dont nous donnons ici une brève présentation.

La première partie, qui sert d'introduction, nous permet de mieux situer le langage C dans son contexte historique et de définir succinctement les principales caractéristiques du langage. Elle nous apporte également quelques notions générales sur C et une description de quelques outils de la bibliothèque C qui s'avèreront indispensables dans l'illustration de nombreux exemples.

La seconde partie aborde les éléments de base du langage les unités syntaxiques, les types de données fondamentaux et les classes d'allocation des variables en mémoire.

La troisième partie concerne l'étude des opérateurs et des expressions.

La quatrième partie développe les instructions de contrôle traditionnelles et les instructions simples et composées.

Dans la cinquième partie sont abordés plus en détail les types d'objets complexes tels que les tableaux, les pointeurs, les structures, les unions, les énumérations et les fonctions.

Enfin, la sixième et dernière partie est consacrée à l'environnement de programmation C avec une vue d'ensemble des principales fonctions offertes par la librairie standard de C et des commandes nécessaires à la production de programmes.

Pour clore et compléter cette étude sur le langage C, quelques annexes utiles sont fournies à la fin de cet ouvrage. En particulier, le lecteur y trouvera, des règles et des conventions d'écriture pour programmer dans un style correct, une étude comparative de programmation dans sept langages différents sur un même algorithme (le crible d'Eratosthène), une liste des principaux appels système d'UNIX et un aperçu sur les compilateurs C disponibles sur le marché. Pour terminer, une bibliographie non exhaustive donne une idée de ce qui a pu être écrit sur C et la programmation sous UNIX.

Même si certaines références ou certains exemples sont liés à un environnement UNIX, cet ouvrage doit permettre à un utilisateur de programmer en C sur un tout autre système.

Ce livre s'adresse principalement à des informaticiens ayant déjà quelques notions de programmation en langage évolué. La lecture sera plus aisée pour un public déjà confronté aux langages structurés tels que PASCAL ou PL/1, mais le caractère universel du langage C permet de penser que les programmeurs ayant pratiqué les langages d'assemblage, le FORTRAN, le BASIC, le COBOL, y attiseront leur curiosité et y trouveront des éléments complémentaires à leur univers d'informaticien.

TABLE DES MATIÈRES

I

INTRODUCTION

1. HISTORIQUE

1.1. Origine de C

Le langage C est né il y a plus de dix ans au sein des Laboratoires de la Bell Telephone au moment où le système UNIX*, son aîné de trois ans, était encore dans sa phase d'expérimentation.

Le système UNIX a été effectivement créé en 1969 lorsque apparaissaient les premiers mini-ordinateurs sur le marché, et où les coûts de production devenaient de moins en moins élevés grâce à la technologie avancée des circuits intégrés. UNIX, sa philosophie, ses spécifications, sa réalisation, émanent de deux hommes, deux ingénieurs des Bell-Laboratories: Ken Thompson et Dennis Ritchie. Ces derniers ne visaient aucun objectif particulier, sinon de "se faire plaisir" en réalisant un système agréable à utiliser.

Dans sa phase de démarrage, le système UNIX est initialement écrit en langage d'assemblage sur le mini-ordinateur PDP-7 de Digital Equipment (DEC). Les contraintes de maintenance et les nécessités de portabilité sur d'autres ordinateurs (PDP-9, PDP-11), amènent alors Ken Thompson à développer un langage indépendant de la machine, un langage évolué possédant des structures de contrôle puissantes.

En 1970, Ken Thompson définit le langage B en s'inspirant du langage BCPL conçu par Martin Richard en 1967 bien avant la vogue actuelle des langages structurés. Thompson développe un compilateur et réécrit le système UNIX en langage B sur un PDP-11.

---

* UNIX est une marque déposée par les Laboratoires BELL.

En 1972, Dennis Ritchie profitant des travaux de Thompson, définit à son tour le langage C qui s'inspire très fortement des langages B et BCPL, ses prédécesseurs.

Les langages BCPL et B possédaient déjà des instructions de contrôle : if, while, do, case, etc., et incluaient même le concept de pointeur, la notion de variable dynamique et la récursivité des fonctions. Le langage C a introduit en plus la notion de type de données, mais moins renforcée qu'en ALGOL 68 ou en PASCAL.

En 1973, le premier système UNIX (version 5) est écrit à 90 % en C sur un PDP-11. C'est ainsi que naquit le langage C.

1.2. Évolution de C

Ce langage a bien sûr suivi des évolutions parallèles à celles du système UNIX qui, lui, est passé par plusieurs versions (V5 en 1973, V6 en 1974, V7 en 1979 et System III en 1981). La version System V qui sera maintenue par Western Electric est annoncée pour début 1984.

A l'heure actuelle, le langage C est devenu assez populaire et répandu pour qu'il vole de ses propres ailes en se démarquant de plus en plus d'UNIX. C est maintenant présent sur la plupart des systèmes modernes : C sous le système VMS du VAX, C sous CP/M, sous MS/DOS et OASIS. La firme Digital Research, qui en 1973 a développé le système CP/M, écrit en langage PL/M puis optimisé en assembleur pour les microprocesseurs 8080, 8085, Z80 et 8086, vient d'adopter le langage C pour réécrire son système afin de pouvoir le transporter sur d'autres familles de microprocesseurs (Z8000, MC68000).

Actuellement un standard du langage C est en cours de définition. Des efforts de standardisation de C ont en effet été entrepris conjointement par les Laboratoires de la Bell, les groupes d'utilisateurs et le comité de normalisation de l'ANSI.

2. PRÉSENTATION GÉNÉRALE DU LANGAGE

2.1. Les domaines d'utilisation de C

Comme nous l'avons vu, les évolutions de C et d'UNIX sont intimement mêlées. En ce sens, on peut dire à priori que C est un langage de programmation de système ou un langage adapté au développement de logiciels de base. Cette affirmation se vérifie par l'existence même d'UNIX et de toute la gamme des produits qui constituent le "package UNIX". Mais aussi, le langage C se pratique de plus en plus pour développer des applications scientifiques de haut

niveau, des logiciels de traitement de texte (éditeurs pleine page, formateurs de documents, correcteurs d'orthographe, etc), des bases de données relationnelles, des logiciels de communication inter-ordinateurs, etc.

La puissance du langage C permet de dire aujourd'hui que c'est un langage de programmation à usage général. A titre d'exemple, citons les principaux logiciels de l'environnement d'UNIX écrits en C :

Produits standard (Bell Laboratories) :

```
- C, FORTRAN-77, RATFOR      (compilateurs et traducteurs)
- LEX, YACC                  (générateurs de compilateur)
- ED,SED                     (éditeurs de texte)
- NROFF,TROFF,TBL,EQN,REFER (traitement de documents)
- SPELL                      (détecteur de fautes d'orthographe)
- LEARN                      (apprentissage du système UNIX)
- CU, UUCP                   (communications inter-ordinateurs)
```

Autres produits développés :

```
- PASCAL UCSD, PASCAL ISO, PASCAL UBD
- BASIC BB2 BB3, CBASIC-16, BASIC M
- RMCOBOL, CIS-COBOL
- VI, EX, EMACS                    (éditeurs plein écran)
- INGRES,INFORMIX,MISTRESS,UNIFY (bases de données)
```

2.2. Comparaison sommaire à d'autres langages

Si on tente une comparaison avec d'autres langages évolués, C se place à un excellent niveau, surtout sur le plan de l'efficacité (puissance d'un assembleur). Pourtant, selon ses auteurs, C n'est pas considéré comme un langage de "très haut niveau" du type PL/1 ou ADA. C est un langage d'utilisation générale (general purpose). Il n'est pas spécialisé dans des domaines particuliers d'application, comme l'est FORTRAN, langage à caractère scientifique, ou COBOL, langage adapté à la gestion.

Une des principales insuffisances du langage C, est qu'il est un langage "atypique", comme l'affirment ses auteurs. Les données qu'il manipule sont encore trop proches de l'architecture des machines. Il traite les mêmes objets que ceux des ordinateurs : les mots, les registres, les octets, les nombres représentés en virgule fixe et flottante, les adresses (pointeurs) et d'autres objets combinés à partir de ces derniers (tableaux, structures, unions), mais ne va pas au-delà dans l'abstraction des types.

De même, le langage C ne fournit pas d'opérations pour manipuler directement les chaînes de caractères et les tableaux comme un tout, comme le font PL/1 et APL, ou pour manipuler les listes comme le fait le langage LISP.

En outre, rien n'a été prévu en C pour résoudre la multiprogrammation, les opérations parallèles, le contrôle de processus ou la synchronisation, comme on le rencontre dans le langage PL/1.

Les autres insuffisances de C, par rapport aux autres langages structurés, se manifestent par l'absence de gestion mémoire intégrée (allocation, libération, retassement, etc), sauf en ce qui concerne la gestion de la pile pour les variables automatiques, et par l'inexistence d'instructions d'entrées-sorties (read, write, get, put) et de méthodes d'accès aux fichiers (séquentiel, direct, séquentiel indexé).

Malgré l'absence de quelques-unes de ces caractéristiques, au niveau interne du langage, qui peuvent paraître comme des carences, l'environnement de C permet de résoudre la plus grande partie de ces problèmes.

En effet, non traités directement par le langage, tous les mécanismes de haut-niveau, sont en fait fournis par des fonctions explicitement appelées et regroupées dans une librairie appelée la librairie standard, elle même écrite en C. On y trouve les fonctions de gestion mémoire d'allocation (alloc), de libération (free), les fonctions d'accès aux fichiers (open, close), les fonctions d'entrées-sorties (read, write, getc, putc, printf, etc.). Les fonctions de gestion des processus parallèles et la synchronisation (fork, exec, wait, exit, pipe, signal, etc.), ne sont possibles que dans un environnement de type UNIX ou "UNIX-like". Cette librairie offre en outre tous les outils logiciels nécessaires au programmeur: tests et manipulations de caractères, de chaînes de caractères, de fonctions mathématiques, etc.

Le langage C fournit les instructions usuelles de contrôle (if, else, while, do, for, switch, break) qu'on retrouve dans la plupart des langages de programmation structurés.

Il offre également des types de données de base comme PASCAL. De nouveaux types d'objets peuvent être créés à partir des types fondamentaux (parfois de manière récursive). C comprend donc une hiérarchie conceptuellement infinie de types dérivés.

Par contre le contrôle des types est loin d'être aussi sévère qu'en PASCAL, voire même relaché. En ce sens, C est dit un langage "faiblement typé". C ne masque pas ses objectifs de souplesse et d'efficacité qui se caractérisent par sa permissivité, et diront certains, par son insécurité. Conscients de ce fait, les auteurs de C ont adjoint un programme de contrôle sémantique, appelé "lint", indépendant du compilateur et tournant sous UNIX.

Deux autres caractéristiques importantes de C permettent de résoudre avec élégance les problèmes de taille mémoire grâce à la réentrance des programmes et la récursivité des fonctions. La réentrance permet le partage du code exécutable entre plusieurs utilisateurs. De même les fonctions en C peuvent être récursives, c'est-à-dire s'appeler elles-mêmes.

Un des atouts majeurs de ce langage c'est qu'il permet d'utiliser naturellement les primitives du système d'exploitation (UNIX ou un autre) par des appels de fonctions. Ces fonctions qui permettent d'accéder aux points d'entrée du système sont regroupées dans une librairie système appelée la "C Run Time Library", dépendante du système d'exploitation.

C remplace avantageusement la programmation en assembleur. Par exemple, les "drivers" ou "handlers" de périphériques sous UNIX sont intégralement écrits en C. Ils font appel à des routines spécifiques du noyau pour effectuer les transferts physiques ou pour traiter les interruptions externes.

Finalement, les principaux points qui font l'originalité du langage C sont: son jeu très riche d'opérateurs, sa permissivité sur la manipulation des types de données et son efficacité sur le plan de la production du code objet qui est très fortement optimisé. Le compilateur, lui-même, n'occupe que très peu d'espace mémoire et est facilement transportable d'une machine à une autre ou d'un système à un autre.

Dépouillé de certains outils très sophistiqués, rencontrés dans la plupart des langages de très haut niveau, le langage C, par sa concision, est d'une grande facilité d'apprentissage.

3. ASPECT GÉNÉRAL D'UN PROGRAMME

3.1. Les fonctions

Une fonction est en fait un sous-programme, au sens "subroutine" de FORTRAN ou "function" de PASCAL, à laquelle on transmet une liste d'arguments (qui peut être vide), et qui retourne une valeur à la fonction appelante.

La notion de "procédure" en PASCAL, ou séquence d'instructions constituant un tout, sans retourner de résultat, n'existe pas en C, mais est équivalente à une fonction qui ne retourne pas de valeur ou à un appel de fonction qui n'affecte pas de variable.

De manière générale, un programme C est constitué de séquences d'objets de données et de corps de fonctions dans de multiples fichiers.

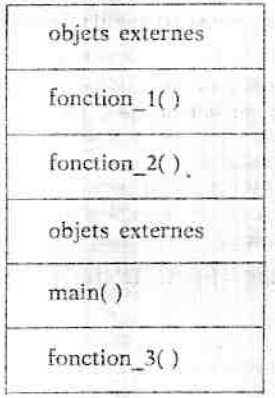
Les séquences d'objets sont situées à l'extérieur des corps des fonctions. Les objets externes aux fonctions sont en fait des variables globales utilisables par les diverses fonctions du programme, elles-mêmes considérées comme externes. Tous ces éléments, variables externes et fonctions, sont visibles entre eux au niveau de l'exécution du programme.

Si les objets de données ou variables définies à l'extérieur des fonctions sont globaux à tous les fichiers du programme, les fonctions possèdent aussi leurs propres objets, dits variables locales. Les variables locales à une fonction ne sont pas visibles de l'extérieur de celle-ci.

Les variables doivent impérativement être déclarées avant leur utilisation. Le plus souvent les déclarations de variables sont situées en tête du corps de la fonction. La seconde partie du corps de la fonction constituant la séquence des instructions exécutables. Une fonction peut à nouveau être organisée en plusieurs blocs d'instructions composées, chacune pouvant avoir ses propres variables privées, invisibles des blocs extérieurs.

Un programme C n'est donc constitué, mises à part les variables externes, que de fonctions. Celles-ci peuvent apparaître dans n'importe quel ordre dans le fichier compilable.

Exemple :

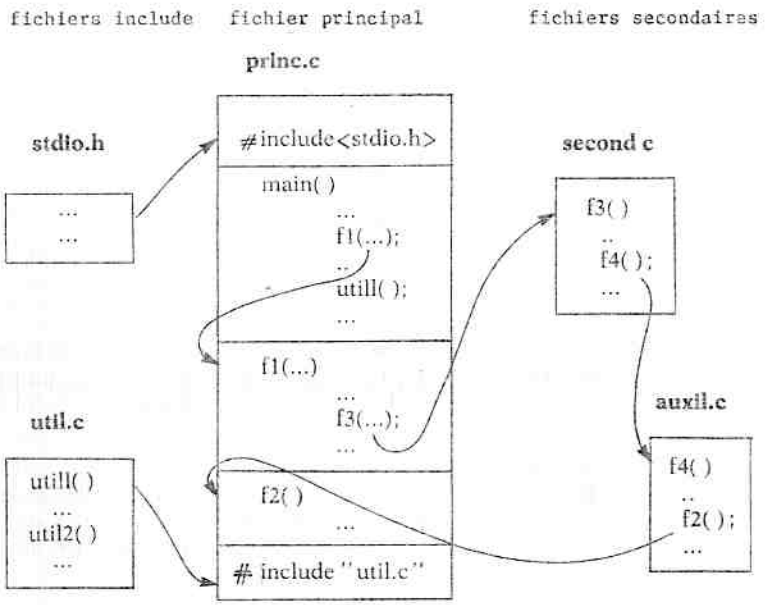


3.2. Les fichiers

Les fonctions peuvent être physiquement regroupées dans un même fichier source, mais aussi être éparpillées dans différents fichiers. Cette organisation implique nécessairement que le langage C offre la compilation séparée des différents modules constituant un programme.

Une directive d'inclusion de fichier source **#include**, permet d'inclure à tout endroit dans le fichier principal des fichiers source secondaires. En fait, cette directive n'appartient pas au langage C, mais est traitée par le macro-préprocesseur C dont le rôle est de préparer le fichier compilable définitif.

La liaison entre ces fonctions, et des objets externes par extension, est réalisée par l'éditeur de liens. Celui-ci, produit à partir des modules objets issus des compilations séparées, et des binaires objets de la librairie standard, un module exécutable de nom **a.out** par défaut.



Les fichiers source en C

La figure ci-dessus montre que plusieurs fichiers source peuvent contribuer à construire un fichier exécutable. Pour cela, il est nécessaire d'effectuer trois compilations séparées : celle du fichier **princ.c** auquel sont inclus les fichiers **stdio.h** et **util.c**, et celles des fichiers **second.c** et **auxil.c.** Les modules objets **princ.o**, **second.o** et **auxil.o** devront alors être édités ensemble avec la librairie standard pour produire le fichier exécutable **a.out**.

3.3. Le programme principal

Que le programme C soit monolithique ou morcelé en plusieurs fichiers, une fonction particulière joue le rôle de fonction principale, encore appelée communément "programme principal".

Cette fonction a pour nom "main". Ce nom n'est pas connu du langage, mais est reconnu par l'éditeur de liens, qui peut alors facilement identifier quelle est la fonction initiale en vue de lui donner le contrôle au niveau de l'exécution après la phase de chargement.

Exemple:

```
main()
{
        ...
        fonction(arg1,arg2);
        ...
}
fonction(param1,param2)
{
        ...
}
```

Les appels de fonctions en C, comme **main( )**, sont identifiés par un nom symbolique, le nom de la fonction, suivi d'une liste d'arguments mise entre parenthèses. Dans le cas ci-dessus **main( )** ne comporte pas d'arguments, mais les parenthèses sont néanmoins obligatoires.

Dans l'environnement UNIX, existe un programme, appelé le **Shell**, qui permet de réaliser l'interface entre l'homme et la machine. Le **Shell** est à la fois un interpréteur de commandes et un langage de programmation. Toute commande utilisateur est interprétée par le **Shell**, lequel lance l'exécution de la commande.

Les arguments propres à la commande, s'ils sont présents, pourront être pris en compte et analysés par le programme utilisateur, c'est-à-dire le programme identifié par la commande. En effet, le **Shell** lui transmet le nombre d'arguments ainsi qu'une adresse à partir de laquelle il lui sera possible de retrouver chacun des champs de la commande, sous forme de chaînes de caractères.

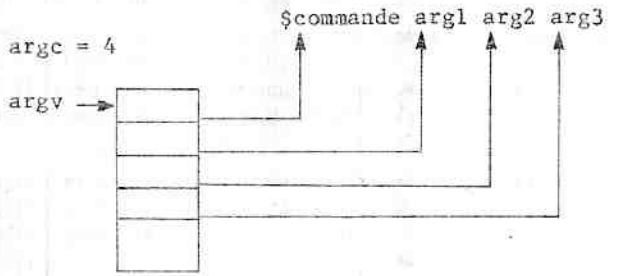
Pour prendre en compte les paramètres transmis par le **Shell**, la déclaration de la fonction du programme principal s'écrit:

```
main(argc,argv)
```

où **argc** = compte d'arguments (nom de commande inclus)

et **argv** = pointeur d'une table de pointeurs, chacun pointant vers les arguments enregistrés sous forme de chaînes de caractères.

Exemple de commande sous UNIX:



Les arguments transmis par le "shell"

4. OUTILS PRÉLIMINAIRES

Avant d'aborder en détail les différents éléments du langage C, étude qui ne peut se faire que séquentiellement, il s'avère nécessaire de présenter, en premier lieu, quelques outils standard de grande utilité qui vont nous servir à introduire ou reproduire des données (caractères, nombres entiers et flottants, chaînes de caractères, etc.), comme si l'on se trouvait en face d'un terminal conversationnel "écran-clavier".

Ces outils sont en fait des fonctions (que nous étudierons par la suite) qui ne font pas partie du langage, mais appartiennent à la librairie des fonctions standard, associée au compilateur C.

4.1. Les fonctions: getchar et putchar

La fonction **getchar( )** permet de lire un caractère sur le fichier standard d'entrée "stdin". Par défaut c'est le clavier du terminal qui est assigné au fichier standard d'entrée. Cette fonction retourne la valeur du caractère lu. Si **carac** représente une variable de l'utilisateur, on peut écrire l'affectation suivante:

```
carac = getchar();
```

carac contient alors la valeur du caractère tapé au clavier.

Pour afficher un caractère sur le fichier standard de sortie "stdout", par défaut l'écran, on utilise la fonction **putchar(c)** où **c** est le nom de la variable donnée par l'utilisateur contenant la valeur du caractère à imprimer:

```
putchar(c);
```

On peut remarquer au passage que **getchar( )** se comporte comme une fonction au sens PASCAL, et **putchar(c)** comme une procédure (absence de valeur retournée).

La variable **c** mise entre parenthèses peut aussi être remplacée par une constante, ce qui conduit aux écritures suivantes:

```
putchar('a');
putchar('Z');
putchar('8');
putchar('+');
```

Puisque **getchar( )** est une fonction qui délivre un caractère, **getchar( )** seul, représente un caractère, et on peut par exemple écrire:

```
putchar(getchar());
```

cette construction ayant pour effet de doubler la frappe du caractère tapé par son écho.

4.2. Fin de ligne, fin de fichier

La frappe sur la touche "return" ou sur la touche "line-feed" provoque une fin de ligne. La représentation en C d'un caractère de fin de ligne s'exprime par "\n". Le caractère "\" est un caractère dit d'échappement qui permet de donner une signification au symbole qui le suit; ici "n" pour "newline".

De même une fin de fichier sur le clavier (fin d'introduction des données), est obtenue, sous UNIX, par la frappe du caractère de fonction <**Control D**>. La fin de fichier est représentée par la constante symbolique **EOF** dont la valeur est (— 1) ou (0) selon les compilateurs. Si le programmeur ne connaît pas la valeur de **EOF**, celle-ci est définie dans le fichier en-tête **stdio.h** qui contient les déclarations de variables et de fonctions ainsi que les définitions de constantes les plus usuelles.

L'utilisation des fonctions **getchar( )**, **putchar( )** et du symbole **EOF** nécessite l'inclusion préliminaire de ce fichier au début du programme, par la directive **#include**. Exemple:

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

Le fichier **stdio.h** regroupe la plupart des déclarations des principales fonctions intégrées dans la librairie standard.

4.3. La fonction printf

Une des fonctions les plus utilisées en C est la fonction **printf** qui permet une impression avec format. Cette fonction fait bien sûr partie de l'environnement de C et se trouve dans la librairie standard associée au compilateur.

Printf accepte différents types possibles d'arguments: caractère, valeur numérique entière ou réelle, chaîne de caractères, et les imprime, selon le format spécifié, sur le fichier de sortie standard (**stdout**), normalement l'imprimante ou l'écran de la console de l'utilisateur.

La forme la plus générale de la fonction **printf** est la suivante:

```
printf("format",arg1,arg2,...,argn);
```

où **format** est une chaîne de caractères dans laquelle on peut trouver deux

types de données: du texte à imprimer littéralement et des spécificateurs de format pour imprimer les variables données en argument. Une des formes les plus simples de **printf** est:

```
printf("message de bienvenue");
```

Si l'on désire que le message se termine par un retour à la ligne on doit rajouter **\n** à la fin du texte, soit:

```
printf("message de bienvenue\n");
```

Chaque spécificateur de format commence par le caractère **"%"** suivi par un caractère indiquant le format à utiliser pour l'impression de l'argument correspondant (caractère, nombre, chaîne). Le premier spécificateur de format rencontré correspond au premier argument, et ainsi de suite. Il doit y avoir autant de spécificateurs de format que d'arguments présents dans la liste.

Les caractères de conversion standard sont:

- d : notation décimale
- o : notation octale
- x : notation hexadécimale
- u : notation décimale non signée
- f : représentation flottante
- c : caractère imprimable en ASCII par exemple
- s : chaîne de caractères

Exemple: si a = 75

```
printf("a_dec= %d , a_oct= %o , a_hex= %x , a_car= %c",a,a,a,a);
```

donnera comme résultat:

```
a_dec= 75 , a_oct= 0113 , a_hex= 4b , a_car= K
```

Un certain nombre de caractères optionnels peuvent être insérés entre le symbole "%" et le caractère spécifiant la conversion. Par exemple:

- le signe "—" pour demander un cadrage à gauche, au lieu du cadrage à droite pris par défaut;
- un nombre indiquant la taille minimale en caractères du champ à imprimer. Les "espaces" jouant le rôle de caractères de remplissage;
- un point décimal (virgule flottante), suivi d'un nombre donnant la précision de la partie fractionnaire, c'est-à-dire le nombre de chiffres significatifs après le point. Si la donnée n'est pas du type flottant, ce nombre représente la taille maximale du champ à imprimer.

Exemples:

%8d imprime un nombre en décimal cadré à droite dont la longueur du champ imprimable est de huit caractères. Des espaces de remplissage précèdent le nombre.

%—25s imprime une chaîne de caractères cadrée à gauche assurant une longueur minimum de 25 caractères.

%.6f imprime un nombre flottant avec un maximum de six chiffres significatifs.

L'utilisation de la fonction **printf** est double.

— D'une part **printf** est normalement utilisé pour imprimer des messages, poser des questions ou éditer des états ('mise en forme de résultats).

— D'autre part **printf** est pratiquement le seul outil de mise au point dynamique d'un programme. En effet il est parfois utile de rajouter des **printf** à des endroits judicieux du programme (entrée et retour de fonction, à l'intérieur des boucles, avant ou après les **if**, etc), pour visualiser l'état des variables ou des expressions. Dès que la phase de mise ou point s'avère correcte, ces **printf** de "debugging" doivent être soustraits du programme.

4.4. La fonction scanf

Scanf est à **printf** ce que **getchar** est à **putchar.** La fonction **scanf** permet en effet de lire sur le fichier standard d'entrée (**stdin**), normalement le clavier, des données selon un certain formatage. Comme pour **printf** les données peuvent être converties sous forme de caractère, de nombre entier ou réel et de chaîne de caractères, selon les spécificateurs de format utilisés.

La syntaxe est la suivante:

```
scanf("format",liste d'arguments);
```

où **format** est une suite de spécificateurs de conversion du même type que pour **printf**. Comme pour **printf** il doit y avoir autant d'arguments (noms de variables) dans la liste que de spécificateurs de conversion.

Exemple: si **r** est une variable réelle, on peut écrire:

```
printf("donner le nombre PI:");
scanf("%f",&r);
```


Le signe **&** spécifie qu'il s'agit de l'adresse de **r**. L'utilisateur peut alors introduire au clavier:

```
donner le nombre PI : 0.314159e+1
```

et la variable **r** contiendra alors la valeur de PI (en flottant) avec cette précision.

Autre exemple: lecture d'une chaîne de caractères sur le clavier si **name** est défini comme une chaîne. L'opérateur d'adressage & n'a plus de raison d'être car **name** représente un pointeur sur la chaîne réceptrice:

```
nom ?= Durand
```

soit à la console:

```
char name[20];

printf("nom ?= ");
scanf("%s",name);
```

Le caractère * précédé de % spécifie que la valeur lue sera sautée, donc non affectée à la variable suivante.

Exemple:

```
scanf("%d %*s %d %*s %d %*s",&heures,&minutes,&secondes);
```

L'entrée des données au clavier pourrait se présenter sous la forme:

```
17 H 35 min 30 secondes
```

Les chaînes de caractères **H**, **min**, **secondes** seront alors ignorées.

II

LES ÉLÉMENTS DE BASE DU LANGAGE

Le langage C comporte six classes d'unités syntaxiques:

- les indicateurs
- les mots réservés
- les constantes
- les opérateurs
- les séparateurs

Les séparateurs peuvent être constitués d'un ou plusieurs caractères "espace" ou "blanc", d'un ou plusieurs caractères de tabulation "TAB" et du caractère de passage à la ligne suivante "newline". Ces caractères servent à délimiter les unités syntaxiques.

Avant d'aborder en détail chacune de ces unités syntaxiques, nous commencerons par définir les commentaires qui ne rentrent pas dans cette classification, car ignorés par le compilateur.

1. LES COMMENTAIRES

Par souci de présentation et de clarification, il est souvent judicieux d'agrémenter son programme de commentaires. Ceux-ci sont en effet utiles pour la compréhension du programme, lorsqu'il doit être relu par une tierce personne. Il n'est pas indispensable d'être un expert en C, pour suivre un programme bien commenté, ce qui est important pour assurer un bon suivi du produit; c'est le cas de sa maintenance.

En C, un commentaire est constitué par une suite de caractères compris entre deux symboles "/*" et "*/". Les deux caractères "/*" juxtaposés, identifient le début du commentaire. Celui-ci peut être constitué d'une ou plusieurs lignes, et doit se terminer obligatoirement par "*/".

Exemples:

```
/* ceci est un commentaire sur une ligne */

/*

   ce commentaire indique que l'accentuation ne sera pas
   representee dans tous les commentaires qui suivront
*/

/***************************/
/*  exemple de commentaire */
/*       esthetique        */
/***************************/
```

Une restriction cependant: les commentaires ne doivent pas être imbriqués, car c'est le premier "*/" rencontré qui fermera le commentaire.

Exemple:

```
/* ceci est un commentaire
         /* imbrique */
   qui est incorrect     */
```

dans cet exemple: "qui est incorrect" ne fait plus partie du commentaire et sera analysé par le compilateur comme une instruction.

Remarque importante: lorsqu'on ouvre un commentaire par "/*", ne jamais oublier de le fermer par "*/", car une séquence ou la totalité de la suite du programme serait considérée comme un commentaire et donc ignorée par le compilateur.

Exemple:

```
/* commentaire non clos
        . . .
   séquence de programme
        . . .
/* autre commentaire qui ferme le premier */
```

2. LES IDENTIFICATEURS

Les identificateurs sont des noms qui permettent de référencer les différents objets manipulés par le programme. En particulier:

- les constantes symboliques,
- les variables,
- les fonctions.

Un identificateur est constitué d'une séquence de lettres et de chiffres, le premier caractère devant être obligatoirement un caractère alphabétique.

Les lettres peuvent être indifféremment en minuscules ou en majuscules, mais il n'y a pas d'équivalence entre minuscules et majuscules.

alpha et ALPHA sont deux identificateurs différents

Le caractère souligné "_" est également autorisé et compte comme une lettre, il sert souvent à aérer des identificateurs assez longs.

Exemple:

pointeur_fichier
process_id

Seuls les huits premiers caractères de l'identificateur sont significatifs, bien que ce dernier puisse avoir une longueur quelconque, donc supérieure à huit. Le caractère espace, étant un séparateur, ne peut faire partie d'un identificateur car il en détermine sa fin.

identificateurs corrects	identificateurs incorrects
i	8080
TAMPON	1ere
Adresse_Memoire	Bus de donnee
cas_01	cas:01
Z 80	CP/M

de même:

treslongidentificateur et treslongcourrier

sont équivalents à:

treslong (huit caractères significatifs).

Notons aussi que certains identificateurs sont interdits car déjà réservés par le langage. Il s'agit des mots réservés du langage, appelés aussi mots clés.

3. LES MOTS RÉSERVÉS

Comme pour la plupart des langages évolués, C possède un jeu de mots clés nécessaires à la sémantique du langage:

- spécificateur de type d'objet,
- spécificateur de classe d'allocation d'un objet,
- opérateur symbolique,
- instruction de contrôle,
- étiquette de contrôle.

En aucun cas ces mots clés ne peuvent être utilisés comme des identificateurs classiques.

Liste des mots réservés:

type	classe	instruction	opérateur	étiquette
int char short long unsigned float double struct union enum	auto extern static register typedef	if else while do for switch break continue goto return	sizeof	case default

4. LES CONSTANTES

Il existe plusieurs types de constantes liées à l'architecture de la machine (taille du mot, code interne des caractères, représentation en virgule fixe et flottante, etc).

- nombre entier,
- nombre réel,
- caractère,
- chaîne de caractères,
- expressions de constantes.

4.1. Nombres entiers

Un nombre entier est généralement lié à la taille du mot physique de la machine. Pour les microprocesseurs 8086 d'Intel, Z8000 de Zilog et MC68000 de Motorola, un entier est codé sur 16 bits.

Du point de la formulation, un nombre entier peut être représenté selon trois systèmes de notation (décimal, octal et hexadécimal).

— en *décimal* c'est la notation usuelle qui est employée.

Exemple:

14 , 213 , 65535 , 1

— en *octal*, chaque chiffre est codé sur trois bits, donc en base 8. Pour différencier un nombre octal d'un nombre décimal, le nombre doit être précédé du chiffre '0'. Les nombres précédents, écrits en décimal, s'écrivent en octal:

016 , 0325 , 0177777 , 01

— en *hexadécimal* ou notation en base 16, les chiffres 10 à 15 sont codés de 'a' ou 'A' à 'f' ou 'F'. Pour les représenter, ils doivent être précédés des deux caractères juxtaposés "0x" ou "0X". Exemple des mêmes nombres déjà cités:

0xe , 0Xd5 , 0xFFFF , 0x01

Un nombre entier qui excède la capacité d'un mot machine sera considéré comme un entier long, et donc codé sur deux mots mémoire consécutifs.

Exemple:

128000

Dans certains cas, même si le nombre n'excède pas la taille physique du mot machine, le programmeur peut demander explicitement que ce nombre soit codé sur un entier long. Pour cela, le nombre entier, qu'il soit représenté en décimal, octal ou héxadécimal devra être suivi du caractère "l" ou "L" spécifiant qu'il s'agit d'un entier long.

Exemple:

14l , 0325L , 0xffffL , 01l

4.2. Nombres réels

Un nombre réel est associé en machine à un nombre en virgule flottante, appelé aussi "flottant", par opposition à un nombre entier, dit aussi "virgule fixe". En machine, un *flottant* est constitué d'un exposant, de son signe, d'une mantisse et de son signe. L'exposant exprime la puissance et la mantisse les chiffres significatifs, c'est-à-dire la précision.

Au niveau de la formulation, un nombre réel comporte une partie entière suivie d'un point et d'une partie fractionnaire, elle-même peut être suivie d'un caractère "e" ou "E", signé ou non, et terminé par un entier qui exprime le facteur d'échelle (puissance de 10).

Syntaxes:

[—] ddd.ddddd	notation décimale
[—] d.ddddde[±]dd	notation scientifique

où **d** est un chiffre décimal. Les crochets indiquent que les signes sont facultatifs.

Exemples:

3.14159	—518.14E17
0.314e+1	—4.2812e—4

4.3. Caractères

Les principaux codes internes de représentation des caractères en machine sont les codes normalisés: ASCII, EBCDIC et BCD. Sur la plupart des machines modernes, les caractères sont codés en ASCII (code à 7 bits), mais occupent un octet (8 bits) en mémoire physique par souci de simplicité de traitement et de compatibilité avec d'autres machines. Le huitième bit ou bit de poids fort pouvant être utilisé comme bit de parité verticale, ou avoir une autre signification selon l'application.

La représentation d'un caractère au niveau de la programmation s'exprime en encadrant le caractère par deux simples quotes "'".

Exemple:

'A' , 'a' , 'z' , '4' , '$' , ':'

Certains caractères de fonction, non imprimables, peuvent être représentés par le couplage d'un caractère dit "d'échappement", le caractère "\" suivi du caractère symbolisant la fonction.

Table des caractères de fonction

newline	NL	ou LF	\n
tabulation	TAB	ou HT	\t
backspace	BS		\b
retour chariot	CR		\r
form-feed	FF		\f
antislash	\		\\
simple quote	'		\'
octet binaire	ddd		\ddd

les trois chiffres "ddd" sont représentés en notation octale, et permettent de définir n'importe quelle configuration de caractère sur 8 bits.

Exemple:

```
BELL    \007
XON     \021
XOFF    \023
rubout  \177
NUL     \0
```

Si le caractère qui suit immédiatement l'antislash n'est pas un de ceux spécifiés ci-dessus, l'antislash est alors ignoré.

\ x donne x (pour x non octal)

Attention:

\7 donne 7 (binaire)

mais:

\8 donne 8 (caractère ASCII), soit 0x38 (binaire)

4.4. Chaînes de caractères

Une constante chaîne de caractères est constituée d'une séquence de caractères délimitée par des doubles quotes "...".

Exemple:

```
"suite de caracteres"
```

Une chaîne de caractères est en fait un vecteur ou un tableau unidimensionnel de caractères, du type (*array of*).

```
chaine[]
```

Toutes les chaînes de caractères sont obligatoirement terminées par le caractère NUL 0. Celui-ci est engendré de manière automatique par le compilateur.

Exemple:

nom =? sera représenté en machine par:

n	o	m	=	?	\0

avec:

```
nom[0] = 'n'
nom[1] = 'o'
nom[2] = 'm'
nom[3] = '='
nom[4] = '?'
nom[5] = '\0'
```


Si deux chaînes sont identiques, leurs emplacements sont distincts en mémoire et sont alloués en mode "statique", c'est-à-dire en mémoire permanente et non dans des piles (stockage automatique) ou des registres.

Si le caractère double quote " doit figurer dans une chaîne de caractères, il doit être précédé du caractère d'échappement \ (antislash).

Exemples:

```
"double quote = \""
"antislash = \"
"chaine = \"suite de caractère\""
```

Ne pas confondre:

"a" qui donne en mémoire:

a	\0

et

'a' qui donne en mémoire:

a

Notons qu'une chaîne peut être vide: "", soit 0 en mémoire.

Au même titre que les autres constantes une chaîne de caractères peut être considérée comme une expression, et on peut par exemple écrire l'affectation suivante:

```
nom = "nom_du_fichier"
```

où **nom** est une variable qui représente l'adresse de cette chaîne de caractères.

4.5. Expressions constantes

Une expression constante est une expression qui n'est composée que de constantes reliées entre elles par des opérateurs.

Exemples:

```
4 + 12 * 7
'A' - 'a'
1<< 8
```

5. LES OPÉRATEURS

Les opérateurs peuvent être représentés par un ou deux caractères spéciaux, par un mot réservé (cas du "sizeof"), ou par un spécificateur de type mis entre parenthèses.

Il existe trois classes d'opérateurs:

— les *opérateurs unaires* qui précèdent un identificateur, une expression ou une constante.

— les *opérateurs binaires* qui mettent en relation deux termes ou expressions.

— Un *opérateur ternaire* qui met en relation trois termes ou expressions.

Table des opérateurs

unaires	binaires			ternaires
-		==	=	? :
*	+	!=	+=	
&	-	>	-=	
!	*	>=	*=	
~	/	<	/=	
++	%	<=	%=	
--	|	||	|=	
(type)	&	&&	&=	
sizeof	^		^=	
	>>		>>=	
	<<		<<=	
	.	->		

6. LES DÉLIMITEURS

Les délimiteurs sont des caractères spéciaux qui permettent au compilateur de reconnaître les différentes unités syntaxiques du langage. Les principaux délimiteurs sont les suivants:

```
;   : termine une déclaration de variable ou une instruction
,   : sépare deux éléments dans une liste
( ) : encadre une liste d'arguments ou de paramètres
[ ] : encadre la dimension ou l'indice d'un tableau
{ } : encadre un bloc d'instructions ou une liste de valeurs
      d'initialisation
```


7. NOTION DE VARIABLE

Comme dans la plupart des langages, C manipule des objets de différentes natures. les objets de base sont les variables et les constantes.

Les variables peuvent avoir des tailles ou des structures différentes selon le type d'objet auquel on fait référence (caractère, entier, flottant, pointeur, tableau, structure, etc) et selon les caractéristiques internes du processeur (octet, mot, virgule fixe, virgule flottante, codage des caractères). A la différence d'une constante, une variable doit être localisée quelque part en mémoire pour y accéder selon certains dispositifs d'adressage tels que:

— adressage direct d'un objet statique en mémoire,
— adressage indirect d'un objet repéré par un pointeur,
— adressage indexé d'un élément de tableau indicé,
— adressage au sommet d'une pile de travail,
— adressage dans les registres programmables.

En C, une variable est définie par trois attributs:

— son nom ou identificateur,
— son type,
— sa classe d'allocation en mémoire.

La formalisation d'une variable répond à la syntaxe suivante:

classe type nom;

ou

type nom; si classe est implicite

ou

classe nom; si type est implicite (int)

Il existe d'autres variables plus complexes tels que tableaux, pointeurs, structures, unions et fonctions, que nous étudierons par la suite.

8. LES TYPES DE BASE

Comme nous l'avons déjà vu pour les constantes, le langage C offre pour les variables trois types fondamentaux directement liés aux données manipulées par la machine. Les types de base s'expriment à l'aide de mots clés réservés au langage. Pour les trois types fondamentaux les mots sont les suivants:

— *char* pour caractère
— *int* pour entier
— *float* pour flottant

Exemples:

```
char c;     /* c est une variable de type caractere */
int i;      /* i est une variable de type entier    */
float f;    /* f est une variable de type flottant  */
```

Les types fondamentaux se sont affinés pour s'adapter à diverses tailles de mots afin d'améliorer la précision ou de réduire l'encombrement en mémoire. Les trois types de base **char**, **int** et **float** peuvent ainsi être modulés en taille à l'aide des spécificateurs:

— *short* pour entier court
— *long* pour entier long
— *unsigned* pour entier non signé
— *double* pour flottant long

Ces mots clés sont des spécificateurs de type, comme **char**, **int**, **float**, mais sont en fait les adjectifs des spécificateurs des trois types fondamentaux.

Les types **short**, **long**, **unsigned** définissant des entiers, peuvent se formuler avec ou sans le spécificateur **int**, qui lui, est pris par défaut.

Exemples:

```
long int l; long l;          /* sont equivalents */
short int s; short s;        /* sont equivalents */
unsigned int u; unsigned u;  /* sont equivalents */
```

De la même manière que le type **int**, le type **long** considère les entiers signés. Si **long** est précédé de l'adjectif **unsigned**, le signe du nombre est alors ignoré et le nombre entier long doit être pris comme un nombre absolu.

Exemples:

```
unsigned long int ul;     /* ul et lu sont des nombres */
unsigned long lu;         /* entiers longs absolus */
```

Dans certains compilateurs il existe des spécificateurs de type pour les variables non signées tels que:

— *uchar* pour unsigned char
— *ushort* pour unsigned short
— *ulong* pour unsigned long

Le tableau ci-contre nous donne un aperçu de la taille des objets de base pour des ordinateurs à mots de 16 et 32 bits.

type	taille du mot	
	16 bits	32 bits
char	8 bits	8 bits
int	16 bits	32 bits
short	16 bits	16 bits
long	32 bits	32 bits
unsigned	16 bits	32 bits
float	32 bits	32 bits
double	64 bits	64 bits

8.1. Type char

Une variable de type **char** est généralement stockée sur un octet, offrant ainsi toutes les valeurs de 0 à 255. Le codage d'un caractère peut être soit en ASCII soit en EBCDIC. C'est le code ASCII à 7 bits qui est généralement le plus utilisé, en particulier sur les machines modernes et les micro-ordinateurs. Ce code cependant laisse libre le bit de poids fort, et celui-ci est habituellement forcé à zéro pour éviter que les caractères soient signés. Par contre, en transmission, il sert de parité paire ou impaire, calculée sur l'ensemble des huit bits de l'octet.

$$0 \leqslant \textit{caractère} \leqslant +255$$

Certaines machines considèrent des octets signés (cas du PDP-11), dans ce cas le bit de poids fort doit être pris en compte.

$$-128 \leqslant \textit{caractère} \leqslant +127$$

Dans les deux cas, un caractère ASCII ne peut évoluer qu'entre 0 et + 127.

8.2. Type int

Le type **int** est associé à la taille normale du mot physique du processeur. Il permet de représenter les nombres entiers relatifs en virgule fixe. C'est le bit de poids fort qui détermine le signe du nombre. Exemple pour un mot de 16 bits:

$$-2^{15} \leqslant \textit{nombre entier} \leqslant +2^{15}-1$$

$$-32768 \leqslant \textit{nombre entier} \leqslant +32767$$

8.3. Type float

Le type **float** définit un nombre en virgule flottante en simple précision. Un nombre *flottant* est représenté en machine par un exposant et une mantisse dont la taille et la forme sont liées à l'architecture interne du processeur. Pour un nombre *flottant* codé sur 32 bits on a :

10^{-38} *magnitude* 10^{+38}

mantisse = 7 chiffres significatifs

8.4. Type short

Le type **short** définit un entier de taille inférieure ou égale à un entier de type **int**. Par exemple sur une machine 32 bits, **int** est codé sur 32 bits et **short** sur 16 bits. Sur une machine 16 bits, **int** et **short** devraient être synonymes car si le **short** était codé sur 8 bits il ne pourrait évoluer qu'entre – 128 et + 127, ce qui paraît insuffisant, et est normalement réservé au type **char**.

8.5. Type long

Le type **long** définit un entier de taille double d'un entier de type **short**. Pour un entier long codé sur 32 bits, on a :

$$-2^{31} \leqslant l \leqslant +2^{31}-1$$

soit :

$$-2147483648 \leqslant l \leqslant +2147483647$$

8.6. Type unsigned

Le type **unsigned** définit un entier absolu c'est-à-dire toujours positif. Le bit de poids fort n'est plus considéré comme bit de signe, mais fait partie intégrante du nombre.

$0 \leqslant n \leqslant 65535$ pour un mot de 16 bits

$0 \leqslant L \leqslant 4294967295$ pour un mot de 32 bits

Exemple :

```
main()
{
   unsigned char uc = -1;
   unsigned int  ui = -1;
   unsigned long ul = -1;
   printf("uc = %u ; ui = %u ; ul = %lu",uc,ui,ul);
}
```

donne

```
uc = 255 ; ui = 65565 ; ul = 4294967295
```

Le spécificateur de format **%u** convertit la valeur décimale en une valeur absolue non signée.

8.7. Type double

Le type **double** ou **long float** définit un nombre en virgule flottante en double précision. Ce type permet, si un simple *flottant* est codé sur 32 bits, d'étendre la mantisse de 7 à 14 ou 15 chiffres significatifs selon les machines. Si le type **double** n'est pas implémenté, il devient alors synonyme du type **float**.

Notons que toutes les manipulations de nombres réels de type float se réalisent par défaut en double précision (remplissage de zéros dans les poids forts si le nombre originel est un *flottant* simple) ; le résultat est ensuite tronqué et arrondi pour reconstituer un **float**.

9. LES TYPES DÉRIVÉS

De la même manière qu'en PASCAL où il existe des types de base (**char, integer, real, boolean**) et des types (**array, record, pointer**), en C, de nouveaux types plus complexes, peuvent également être construits à partir des types de base (**char, int, float**).

Certains d'entre-eux sont déjà prédéfinis par le langage, d'autres se déduisent des types de base et de certains opérateurs. Il s'agit des objets suivants :

— *array* tableau d'objets de même type
— *pointer* pointeur d'un objet de type donné
— *struct* structure contenant une suite d'objets de types différents
— *union* alternative entre un et plusieurs objets de types différents
— *enum* énumération de constantes symboliques associées à un objet
— *function* fonction qui retourne un objet d'un type donné.

La construction de ces objets peut se faire de manière récursive, et permettre ainsi de multiplier les types d'objets manipulables. Les objets structurés (**array, pointer, struct, union, function**) seront étudiés dans le détail dans le chapitre qui leur est consacré.

10. LES CLASSES D'ALLOCATION DES OBJETS

Nous avons déjà vu qu'une variable était caractérisée par son identificateur (nom de la variable) et son type (taille et essence de la variable) ; c'est ce que l'on rencontre dans la plupart des langages structurés modernes.

En C, apparaît une caractéristique supplémentaire, c'est la classe d'allocation de la variable qui permet de préciser dans quelle catégorie d'espace mémoire elle va être utilisée, (mémoire permanente, pile, registre), et de lui associer des règles de visibilité (variable globale ou locale). Le choix d'une classe d'allocation influe sur le mode de mémorisation mais aussi sur la rémanence ou la durée de vie de la variable. Il est certain qu'une variable stockée dans un registre du CPU aura une existence très brève par rapport à une variable stockée en mémoire vive classique.

Il existe quatre classes d'allocation pour les variables :

— externe,
— automatique,
— statique,
— registre.

Les mots clés qui définissent les attributs d'allocation mémoire sont les suivants :

— *extern*,
— *auto*,
— *static*,
— *register*.

Les spécificateurs de classe se placent devant les spécificateurs de type, qui eux-mêmes précèdent la ou les variables. Le spécificateur de type **int** peut être omis à condition de spécifier la classe de la variable dans la déclaration.

Exemples :

```
static char car;
auto int index;
register char c;
static entier;          /* entier est un int */
register i;             /* i est un int */
```

11. LES VARIABLES GLOBALES

Par définition un programme C est constitué d'objets externes qui peuvent être soit des variables externes, soit des fonctions. Ces objets existent et conservent leur valeur pendant la durée totale d'exécution du programme. Même si les fonctions sont séparément compilées, chaque objet externe est globalement visible et accessible par n'importe quelle autre fonction.

Les fonctions étant aussi des objets externes, les variables externes devront être définies à l'extérieur du corps de ces fonctions, et ainsi deviendront potentiellement disponibles pour ces dernières.

Les variables externes en C sont analogues au COMMON de FORTRAN. Du fait de leur accessibilité à partir de toutes les fonctions, même celles compilées séparément dans des fichiers source différents, les variables externes sont globales par opposition aux variables locales internes aux fonctions.

On définit généralement les variables globales avant le programme principal représenté par la fonction spéciale (connue de l'éditeur de liens) : **main( )**.

Exemple:

```
int v1,v2;      /* variables */
char c1,c2;     /* externes  */

main()          /* programme principal */
{
int i;
        ...
        i = v1 + v2;
        ...
}
f1()            /* fonction f1 */
{
char c;
        ...
        c = c1;
        ...
}
...
fn()            /* fonction fn */
{
        ...
        printf ("caract = %c",c2);
        ...
}
```

Dans le cas où une fonction, située dans un autre fichier source, doit utiliser une variable globale définie ailleurs, cette variable doit à nouveau être déclarée dans le corps de la fonction qui l'utilise. Cette déclaration permet de faire référence à la variable externe, mais ne la définit pas (puisqu'elle l'est déjà à l'extérieur de la fonction). Le spécificateur qui permet de réaliser cette référence est le mot réservé : **extern**.

Exemple:

```
extern int commun;
```

Il ne faut pas confondre définition et déclaration de variable externe. Une définition spécifie toutes les caractéristiques de la variable (type, classe) et alloue la variable en mémoire, tandis qu'une déclaration n'annonce que le type et ne provoque aucune allocation.

Le spécificateur **extern** ne définit pas une variable externe, au contraire, il permet de déclarer une variable externe qui existe déjà à l'extérieur de la fonction, pour que celle-ci puisse l'utiliser.

Notons qu'il n'y a pas de mot réservé pour définir une variable externe.

Exemple:

fichier 1

```
int data;       /* definit et declare
                   la variable externe data de type int */
main()          /* programme principal */
{
    ...
    fonc();     /* appel de la fonction fonc() */
    ...
}
```

fichier 2

```
fonc()              /* fonction fonc() */
{
extern int data;    /* declare la variable externe data */
    ...
    ...
}
```

12. LES VARIABLES LOCALES

Le sens de localité est très vaste en C. En effet une variable peut être locale à :

- un fichier source,
- une fonction,
- une instruction composée.

Il existe trois catégories de variables locales :

- les variables statiques,
- les variables automatiques,
- les variables dans les registres.

12.1. Variables statiques

Les variables statiques sont réservées au niveau de la compilation. L'allocation d'une variable statique étant donc permanente en mémoire, son contenu, qui peut évoluer, est garanti tout au long de la durée d'exécution du programme. En fait les variables statiques se comportent comme des cellules mémoires classiques directement adressables.

La déclaration d'une variable statique est faite explicitement par le spécificateur de classe **static** qui précède le spécificateur de type.

Exemple:

```
static float degre; /* le flottant degre est static */
```

Les variables globales peuvent aussi bénéficier de l'attribut **static**. Ce qui implique qu'elles ne sont visibles que du fichier source où elles ont été définies et non des autres fichiers extérieurs.

Exemple:

```
static long int local;
main()
{
    ...
}
...
```

12.2. Variables automatiques

Les variables automatiques, propres à une fonction, sont allouées dynamiquement dès l'appel à cette fonction, puis libérées au retour de la fonction. Leur contenu est donc définitivement perdu entre deux appels consécutifs à la fonction. De même, si deux exécutions différentes partagent le code de la fonction dans le temps (fonction réentrante), la variable de la première exécution est différente de la variable de la seconde exécution. Les variables automatiques sont des variables temporaires pendant le temps d'exécution de la fonction. Elles sont, soit initialisées dans le corps de la fonction pour compter les itérations de boucle (par exemple: **for**), soit initialisées par copie des arguments transmis par la fonction, ou par une copie d'une autre variable (globale ou locale).

L'implémentation des variables automatiques se fait dans une pile de travail (*stack* en anglais) allouée au programme pendant son exécution.

La classe par défaut au sein d'une fonction étant automatique, le spécificateur **auto** ne devient plus indispensable dans une déclaration.

Les deux déclarations suivantes sont par exemple équivalentes:

```
auto int k;    /* k = entier , classe auto */
    int k;     /* k = entier , classe auto */
```

L'utilisation de la pile pour les variables automatiques permet d'assurer la récursivité de la fonction dans laquelle celles-ci sont déclarées.

Exemple:

```
recursif(n)
int n;
{
    ...
    recursif(n);    /* appel recursif de recursif(n) */
    ...
}
```

12.3. Variables dans les registres

Si une variable locale à une fonction doit être intensément utilisée, le programmeur a la possibilité, à l'aide du spécificateur de classe **register**, de placer sa variable dans un registre programmable de la machine.

Les types d'objets autorisés dans les registres sont ceux dont leur taille est inférieure ou égale à la taille d'un mot physique; en effet les registres rapides ont des tailles identiques aux mots de la machine. Dans un registre, on peut donc mettre des **char**, des **int** et des pointeurs (adresses). Les objets de type **long**, **float** ou **double** ne sont pas permis. De même, il est hors de question de mettre un tableau ou une structure dans un registre.

Exemples:

```
register int i;      /* i : entier dans un registre */
register i;          /* identique a la ligne ci-dessus */
register char c;     /* caractere */
register char *ptc;  /* pointeur de caractere */
register short sh;   /* entier court */
register i,j,k;      /* trois variables i,j,k */
```

Le programmeur ne sait pas à priori dans quel registre il va placer sa variable, et il n'a aucune raison de le savoir. C'est pour cela qu'une nouvelle restriction interdit d'utiliser l'opérateur d'adressage **&** sur une variable registre.

Exemple:

```
int *adr;          /* adr = pointeur d'entier */
int i;
register j;
    ...
    adr = &i;      /* est correct */
    adr = &j;      /* est incorrect */
```

Le compilateur essaie de satisfaire au mieux les demandes d'allocation sur les registres dans la mesure où il en reste de disponibles. De manière générale, pas plus de trois registres sont réservés pour cette classe de stockage. Si le nombre de variables à placer dans les registres est supérieur à cette limitation, le compilateur continue d'allouer les variables suivantes en mode automatique, c'est-à-dire dans la pile de travail.

Le choix judicieux de l'utilisation des registres peut réduire d'une manière appréciable l'espace et surtout le temps d'exécution d'un programme.

13. DÉCLARATION DES VARIABLES

Dans un programme C toutes les variables doivent être définies et déclarées avant leur utilisation. Une déclaration de variable peut avoir plusieurs significations selon le contexte ou selon que la variable est ou non définie, et n'alloue pas nécessairement de la mémoire à la variable. Une déclaration permet de :

- définir des variables avec leur classe d'allocation,
- déclarer des variables ou des fonctions définies ailleurs,
- définir des types de données structurées,
- combiner ces possibilités.

Les déclarations peuvent apparaître dans trois endroits différents d'un programme C :

- à l'extérieur des fonctions (déclaration globale),
- entre la définition d'une fonction et le corps de la fonction (déclaration des paramètres),
- au début d'une instruction composée (déclaration locale).

Dans tous les cas, la formulation, qu'elle soit complète ou non, doit se terminer par le caractère délimiteur ";".

Nous avons déjà mentionné le distingo entre définition et déclaration d'une variable externe :

définition = spécificateur de classe + déclaration

Dans la plupart des cas, l'allocation est implicite et il n'est pas nécessaire de spécifier l'identificateur de classe. Ce qui revient, sur le plan formel, à ce que déclarations et définitions aient le même aspect. Les définitions implicites dépendent généralement du contexte du programme. C'est le cas :

- des *variables externes* qui n'ont pas de spécificateur de classe et sont définies à l'extérieur des fonctions, les fonctions elles-mêmes étant des objets externes ;

- des *variables automatiques*, internes aux fonctions, qui peuvent avoir le spécificateur **auto**, mais celui-ci est pratiquement inutilisé.

Tableau récapitulatif des définitions de variables

classe	syntaxe	exemple
externe	type identificateur static type identificateur	int common; static double sinus;
automatique	type identificateur auto type identificateur	char c; auto long temporaire;
statique	static type identificateur	static float pi=3.1416;
registre	register type identificateur register identificateur	register char *p; register int x;

13.1. Liste de variables

Chaque identificateur de variable est associé à un type et une classe, même si celle-ci est implicite. Dans le cas où on a plusieurs objets de même type et de même classe à déclarer, il est possible de définir de manière unique la classe et le type de ces objets. Dans ce cas, les identificateurs des variables se présentent sous la forme d'une liste où chaque identificateur est séparé par le caractère de ponctuation ",".

Exemples :

```
register int i,j,k; /* i j et k sont des entiers */

char c,*s,tab[];    /* c est un char
                     * s est un pointeur de caractere
                     * tab est un tableau de char
                     */
```

Cette formulation condensée a ses avantages, mais parfois il est préférable de bien détacher les variables et de leur associer un commentaire en vis-à-vis.

Exemples :

```
char c;             /* c = caractere */
char *s;            /* s = pointeur sur un caractere */
char tab[];         /* tab = tableau de caracteres */
```

Les deux formulations précédentes peuvent néanmoins se concilier comme suit :

```
char c,          /* c = caractere */
     *s,         /* s = pointeur sur un caractere */
     tab[];      /* tab = tableau de caracteres */
```

13.2. Taille des variables

Les objets de base tels que caractères, entiers, flottants, pointeurs ont des tailles imposées par l'architecture de la machine. De même pour les objets de type **long** et **double** qui dépendent des types de base et de l'implémentation qui est faite dans le compilateur.

Les tailles des objets de base pour une machine à mots de 16 bits sont par exemple :

caractère	: 8 bits
entier	: 16 bits
flottant	: 32 bits
pointeur	: 16 bits
entier long	: 32 bits
flottant long	: 64 bits

D'autres objets dérivés des premiers (tableau, structure, union) peuvent avoir des tailles quelconques. Dans le cas d'un tableau, on formule la taille du tableau par une constante entière mise entre crochets, immédiatement placée derrière l'identificateur.

Exemple :

```
int tab[10];          /* tableau de 10 entiers */
float result[12];     /* tableau de 12 flottants */
```

Notons que la déclaration d'un texte de caractères est équivalente à celle d'un tableau de caractères (*array of char*). Pour qu'un texte devienne une chaîne de caractères au sens C, ce texte doit être terminé par le caractère nul 0 binaire.

Exemple :

```
char texte[16];       /* texte de 16 caracteres */
```

Dans le cas des structures ou des unions, que nous étudierons en détail par la suite, elles sont constituées d'une collection d'objets de base ou d'objets déjà structurés. Leur taille est égale à la somme des tailles de chaque objet présent dans la structure.

Exemple :

```
struct {
     int entier;
     short court;
     float flottant;
     int *pointeur;
     char carac;
     int tab[32];
} ;
```

13.3. Initialisation des variables

Les variables simples peuvent être initialisées à l'aide de l'opérateur d'affectation "=". L'initialisation d'une variable se comporte différemment selon la classe d'allocation qui lui est attribuée.

Si la classe de la variable est externe ou statique, l'initialisation est faite une fois pour toute par le compilateur. L'expression à droite du signe d'affectation doit être évaluable, c'est-à-dire qu'elle peut être une constante, une expression de constantes ou une chaîne de caractères.

Exemples :

```
int a = 0;              /* l'entier a vaut zero */
char c = 'a';           /* le caractere 'a' est mis dans c *
double e = 2.71828;     /* nombre e */
int heure = 60 * 60;    /* heure = 3600 secondes */

main()
{
    static char mess[] = "message";
    static float pi = 3.14159;
    ...
}
```

Par contre, si la variable est automatique ou dans un registre, l'initialisation a lieu à chaque exécution de la fonction. Ne pas oublier que les données de classe automatique ou registre sont perdues à la fin de l'exécution d'une fonction. La valeur à initialiser peut être une expression constante comme dans le cas des classes externes et statiques, mais aussi une autre variable automatique visible à cet endroit.

Exemple :

```
fonction(i)
int i;
{
    j = -1;
    register char c = 'A';
    char k = i + c;
    ...
}
```

En l'absence d'initialisation, seules les variables externes et statiques sont initialisées par défaut à zéro une fois pour toute par le compilateur. Les variables automatiques et registre restent indéfinies avant leur utilisation.

L'initialisation des objets plus complexes tels que les tableaux, les structures, les énumérations, sera étudiée dans les chapitres consacrés à ces types d'objets.

III

LES OPÉRATEURS

1. GÉNÉRALITÉS

Le langage C possède un jeu très riche d'opérateurs, ce qui lui permet de rivaliser avec la plupart des autres langages. L'utilisation de l'assembleur, sous Unix, est quasiment devenue obsolète face à C qui apporte une grande diversité d'opérateurs.

En première approximation, on peut dire qu'il existe plusieurs catégories d'opérateurs que l'on peut classifier de la manière suivante :

- les opérateurs arithmétiques,
- les opérateurs de manipulation de bits,
- les opérateurs d'affectation,
- les opérateurs d'incrémentation et de décrémentation,
- les opérateurs relationnels,
- les opérateurs d'adressage,
- les autres opérateurs.

D'autre part, les opérateurs peuvent agir sur une, deux ou même trois expressions, ce qui respectivement correspond aux opérateurs :

- unaires,
- binaires,
- ternaires.

Généralement l'étude des opérateurs commence par l'opérateur d'affectation. Mais dans le cas du langage C, où l'opération d'affectation, déjà abondamment utilisée dans de nombreux exemples (signe =), est intuitive comme dans la plupart des langages, nous ne l'étudierons pas en premier lieu. L'étude préalable des opérateurs arithmétiques et de manipulation de bits nous conduira à une plus grande généralisation de l'opérateur d'affectation.

2. OPÉRATEURS ARITHMÉTIQUES

C dispose de cinq opérateurs arithmétiques classiques.

+	l'addition
−	la soustraction et le moins unaire
*	la multiplication
/	la division
%	le modulo ou reste de la division

On remarque que l'exponentiation ou élévation à la puissance ne fait pas partie du jeu des opérateurs en C. Cette lacune peut être résolue de deux manières.

Soit en multipliant autant de fois la variable par elle-même, ce qui est admissible lorsque l'indice de puissance n'est pas trop élevé.

```
y = a*a*a;          /* a au cube */
```

Soit par l'appel à la fonction puissance, définie dans la librairie mathématique:

```
y = power(a,n);     /* y = a puissance n */
```

Dans la formulation d'une expression arithmétique, il peut y avoir une ambiguité dans l'ordre d'évaluation des opérations si l'expression comporte plusieurs opérateurs.

Par exemple: `a + b * x`

pourrait s'écrire, dans l'esprit du programmeur:

`a + (b * x)` ou `(a + b) * x`

En C, l'évaluation des expressions arithmétiques se fait de la gauche vers la droite. Des priorités, appelées précédences d'opérateur, sont données aux opérateurs pour résoudre cette ambiguité.

Les opérateurs *, / et % ont la même précédence entre eux, mais une précédence supérieure aux opérateurs additifs + et −, eux-mêmes d'égale précédence entre eux.

Ainsi: `a + b * x` sera évalué comme `a + (b * x)`

de même: `c * d % y` sera évalué comme `(c * d) % y`

Dans le doute, pour éviter certaines ambiguïtés de lecture, il est préférable d'agrémenter les expressions de parenthèses. Les parenthèses permettent de s'affranchir des règles de priorité, car leur contenu est évalué en premier. En fait le programmeur pouvait écrire:

`c * (d % y)` ce qui est différent de `c * d % y`

Exemple:

```
main()      /* essais de precedence */
{
int c=2,d=3,y=4;
        printf("c * d %%y =%d\n",c*d%y);
        printf("(c * d) %%y =%d\n",(c*d)%y);
        printf("c * (d %%y) =%d\n",c*(d%y));
}
```

donnera:

```
c * d % y = 2
(c * d) % y = 2
c * (d % y) = 6
```

On remarque que dans la zone format de **printf** nous avons mis **%%** devant **y**. Le caractère **%** a en effet été doublé pour éviter qu'il soit pris comme un spécificateur de conversion. L'impression d'un **%** se fait donc en doublant celui-ci, soit **%%** pour l'opérateur *modulo*.

3. OPÉRATEURS DE MANIPULATION DE BITS

Une des originalités du langage C est d'offrir des opérateurs sophistiqués pour manipuler les informations les plus fines de la machine, à savoir les bits.

Les opérateurs sont au nombre de six. Ils s'appliquent aux objets de types fondamentaux tels que les entiers (**int, short, long, unsigned**) et les caractères (**char**), mais non aux objets de type **float, double** et autres objets de types dérivés.

&	ET logique (AND)
|	OU logique inclusif (OR)
^	OU logique exclusif (EOR)
~	complémentation à un unaire
<<	décalage vers la gauche
>>	décalage vers la droite

L'opérateur binaire **&** est utilisé conjointement avec un masque de bits.

Exemple:

```
long n = 0x12345678;     /* entier sur 32 bits */
printf("%lx & 0X0F0F0F0F = %lx",n,n & 0x0f0f0f0f);
```

donne:

```
12345678 & 0X0F0F0F0F = 02040608
```

On remarque que dans l'utilisation de **printf** nous avons utilisé le spécificateur de format **%lx**. Le caractère **l** devant le **x** indique qu'il s'agit d'un entier long à imprimer, et **x** pour une notation en hexadécimal.

Une des utilisations les plus courantes de **&** est de supprimer le bit de parité d'un octet en transmission.

Exemple la fonction suivante:

```
noparity(c)
char c;
{
        return (c & 0x7f);
}
```

L'opérateur | ou union logique permet de "monter" un ou plusieurs bits dans un objet de type caractère ou entier.

Exemple:

```
carneg(c) char c;  /* fonction caractere negatif */
{
        return (c | 0x80);
}
```

Le OU exclusif s'exprime à l'aide de l'opérateur ^. Si les bits de même rang de la variable et du masque ont les mêmes valeurs, le bit résultat prendra la valeur 0, sinon s'il y a disjonction le bit résultat sera égal à 1.

L'opérateur ~, ou complémentation à un, convertit chaque bit en son inverse: 1 → 0 et 0 → 1. Du fait que cet opérateur soit unaire, on peut le trouver à la suite d'un opérateur binaire.

Exemple:

```
k |~ 0xf0     est équivalent à     k | 0x0f
```

Les opérateurs de décalage << et >> opèrent respectivement une translation binaire à gauche ou à droite de l'expression située à gauche de l'opérateur et dont l'amplitude, en bits, est définie par l'expression située à sa droite.

Exemple:

```
x >> 4   /* decalage a droite de 4 positions binaires */
x << 8   /* decalage a gauche de 8 positions binaires */
```

Les bits sortants sont perdus. Par contre, dans le cas du décalage à gauche <<, les positions binaires laissées libres sont remplies par des bits à zéro; c'est le cas du décalage logique.

Le décalage à droite est généralement arithmétique, si l'objet est du type **int**, **short** ou **long**. Le bit de poids fort se propage à chaque décalage, conservant ainsi le signe initial de l'objet.

Si le type de l'objet est **unsigned**, le décalage à droite est alors logique et les bits de poids fort sont remplis par des zéros.

Le résultat est indéfini si la seconde expression est négative ou si le nombre de décalages binaires est supérieur à la taille de l'objet. Il n'existe pas d'opérateur de décalage circulaire.

4. OPÉRATEUR D'AFFECTATION

L'affectation est l'opération la plus répandue en C comme dans les autres langages. En C, elle est symbolisée par le signe =, comme en FORTRAN et en BASIC, alors qu'en PASCAL et en ADA, elle est représentée par le signe :=.

L'affectation a pour but de mettre une valeur dans une variable. La valeur à introduire est située à droite du signe =, et peut être, soit le résultat d'une expression arithmétique ou logique, soit le résultat retourné par une fonction.

La partie à gauche du signe = est l'identificateur de la variable; on l'appelle aussi une *g_expression* (comme gauche ou *lvalue* (en anglais).

Exemples:

```
i = 8;
j = -1;
k = (j >> i) & 0377;
c = getchar();
vecteur[i] = 0;
matrice[i][j] = 1;
```

Notons qu'en C une affectation est une expression à part entière. Une affectation peut donc à nouveau être affectée. Par exemple l'affectation multiple suivante est évaluée de la droite vers la gauche:

```
i = j = k = 0;      /* k=0 puis j=0 puis i=0 */
```

Il est alors possible d'imbriquer une affectation dans une expression. Dans l'exemple qui suit, le caractère lu est mis dans la variable **c** qui est ensuite masquée avec la constante **0x5f** (suppression du bit minuscule), pour finalement être affectée à la variable **car**:

```
car = 0x5f & (c = getchar());   /* en majuscule */
```

Des simplifications dans la syntaxe peuvent survenir lorsque la partie à gauche du signe = (identificateur) se répète à sa droite. Une affectation de la forme:

variable = variable opérateur autre variable

ou

expr_1 = (expr_1) opérateur (expr_2)

s'écrit:

expr_1 opérateur =expr_2

Opérateur= constitue un opérateur d'affectation composé. Les opérateurs permis sont les opérateurs binaires arithmétiques et de manipulation de bits:

```
+  -  *  /  %  &  |  ^  ~  <<  >>
```

Exemple:

```
a = a & 0x0F      et      b = b / a
```

s'écrivent:

```
a &= 0x0F         et      b /= a
```

attention:

```
x = x * (a + b)  donne    x *= a + b
```

mais:

```
x = x * a + b  est évalué comme : x = (x * a) + b
               si on applique les règles de
               précédence.
```

5. INCRÉMENTATION ET DÉCRÉMENTATION

La simplification de la syntaxe peut aller jusqu'à une condensation extrême de l'écriture avec les deux nouveaux opérateurs ++ et --. L'opérateur d'incrémentation ++ ajoute 1 à la variable qui lui est associée, tandis que l'opérateur de décrémentation -- lui soustrait la valeur 1.

```
x = x + 1    peut s'écrire    x += 1
```

mais aussi:

```
++x  ou  x++
```

De même **--x** *ou* **x--** sont équivalents à:

```
x = x - 1
```

La constante 1 disparaît car elle est implicite tout comme le signe =.

Ces opérateurs peuvent être utilisés soit en préfixe, soit en suffixe de la

variable. En préfixe, la variable est incrémentée ou décrémentée avant son utilisation, alors qu'en suffixe, la variable est incrémentée ou décrémentée après son utilisation. Le résultat final est le même dans les deux cas, mais le comportement intermédiaire de la variable est tout à fait différent selon le contexte dans lequel elle est utilisée.

Exemple:

```
pile[s++] = val;      /* empilement */
```

place la valeur **val** dans la pile indexée par **s** (sommet courant), puis incrémente le sommet en vue d'une prochaine opération sur la pile.

Autre écriture:

```
pile[s] = val;
s++;

val = pile[--s];     /* depilement  */
```

recalcule la valeur du sommet de la pile **(--s)**, puis extrait la valeur dans la pile indexée par **(s-1)**.

Autre écriture:

```
s--;
val = pile[s];
```

6. OPÉRATEURS RELATIONNELS

6.1. Opérateurs de comparaison

Les opérateurs relationnels de comparaison sont au nombre de six:

>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal
==	égalité
!=	inégalité

Ils mettent en relation deux expressions, et le résultat est une expression booléenne vraie ou fausse.

expr_booléenne := expr_1 opérateur relationnel expr_2

Exemple:

si a < b alors l'expression a < b est vraie
sinon l'expression a < b est fausse

Le type *booléen* n'existe pas explicitement en C, à la différence de PASCAL qui supporte le type **boolean**, mais implicitement le résultat d'une expression relationnelle donne toujours en entier 0 ou 1, soit un *booléen*.

Une expression booléenne vraie a pour valeur 1.
Une expression booléenne fausse a pour valeur 0.

Les opérateurs relationnels ont une précédence plus faible que celle des opérateurs arithmétiques et de manipulation de bits, car les expressions de part et d'autre de l'opérateur relationnel, doivent être effectuées avant d'évaluer la condition.

Exemple:

x < MAX − 1 sera évalué comme: x < (MAX − 1)

De plus la précédence des opérateurs == et != est plus faible que celle des opérateurs < <= > >=. Une expression relationnelle de la forme:

x< y == x< z

sera vraie (valeur 1) si x<y et x<z sont vrais tous les deux.

6.2. Opérateurs logiques

Les opérateurs relationnels logiques sont au nombre de trois:

!	négation unaire (NOT)
&&	ET logique (AND)
||	OU logique (OR)

L'opérateur unaire ! a pour effet d'inverser la valeur du résultat de l'expression qui le suit. Si l'expression est vraie (valeur 1), le résultat de l'opération négation sera faux (valeur 0) et vice-versa. Par exemple, si **trouve** est une expression qui donne un résultat vrai, alors:

(trouve) et (trouve != 0) sont équivalents et sont vrais

de même:

(!trouve) et (trouve == 0) sont équivalents et sont faux

vrai vrai
faux faux

Les opérateurs && et || permettent de faire une ou plusieurs opérations logiques entre les expressions qui leur sont associées.

Les expressions reliées par ces opérateurs sont évaluées de la gauche vers la droite. L'évaluation prend fin dès que le résultat d'une seule expression entraîne un résultat définitif (vrai ou faux) pour l'expression globale.

expression globale := (*expr_1 oper expr_2 oper* ...)

Le résultat des expressions globales suivantes est vrai si on a respectivement:

(expr1 && expr2) != 0 l'expression **expr2** sera évaluée seulement si **expr1** est vraie

(expr1 || expr2) != 0 l'expression **expr2** sera évaluée seulement si **expr1** est fausse

Exemple:

(index < MAX−1 && (c=getchar( )) != '\n') && c != EOF)

est vrai si les trois expressions sont vraies; est faux si une des expressions est fausse, l'évaluation est alors arrêtée. Si **index** atteint MAX − 1 (condition fausse) alors les expressions suivantes ne sont pas analysées et l'expression globale a un résultat faux (valeur 0).

L'affectation d'une variable étant elle-même considérée comme une expression, il faut faire attention à l'ordre d'évaluation de l'expression lorsque celle-ci comporte plusieurs opérateurs.

Par exemple:

```
if (c = getchar() != EOF)    /* cas 1 : incorrect */

if ((c = getchar()) != EOF) /* cas 2 : correct   */
```

Dans le cas 1 la valeur retournée par **getchar( )** est immédiatement comparée à la constante EOF (− 1), car la priorité de l'opérateur **!=** est plus élevée que celle de l'affectation. Si **getchar** retourne une valeur différente de EOF alors l'expression **getchar( ) !=** EOF sera vraie (valeur 1) et la variable **c** sera donc toujours égale à 1 sauf dans le cas EOF où l'expression sera fausse, soit **c = 0**.

Dans le second cas, l'écriture est correcte car les parenthèses (opérateurs de plus forte priorité), permettent de réaliser l'affectation de la variable **c** avant de tester la condition de fin de fichier.

6.3. Opérateur conditionnel

L'opérateur conditionnel est un opérateur ternaire mettant en relation trois expressions. L'expression résultante est alors du type booléenne, soit vraie, soit fausse. Cet opérateur est composé de deux signes : le signe ? (*alors*) et le signe : (*sinon*).

On peut exprimer cette opération à l'aide de l'algorithme suivant :

si (*expression_1*)
alors *expression_2*
sinon *expression_3*

ce qui s'écrit en C de la manière suivante :

(expression_1) ? expression_2 : expression_3

le tout étant à son tour considéré comme une expression à part entière.

L'expression 1 est évaluée en premier ; si elle est vraie alors c'est l'expression 2 qui est évaluée et son résultat est celui de l'expression globale, sinon c'est l'expression 3 qui est évaluée pour donner sa valeur à l'expression globale.

Exemple de la fonction abs(n) donnant la valeur absolue d'un nombre n :

```
abs(n)
int n;
{
        return ((n > 0) ? n : -n);
}
```

7. OPÉRATEURS D'ACCÈS AUX OBJETS

7.1. Adresse d'un objet

Une variable est caractérisée par son nom, son type et sa classe d'allocation en mémoire. Dans l'affectation suivante :

a = b

le contenu de la variable b est rangé dans la variable **a**. Les variables **a** et **b** ont chacune une adresse en mémoire.

Le langage C fournit un opérateur unaire **&** qui s'applique sur la variable qui le suit. Ne pas le confondre avec l'opérateur **&** binaire qui réalise un ET logique

entre deux variables de type entier ou caractère. L'adresse de l'objet **obj** s'écrit : **&obj**, et on peut poser l'affectation :

```
adr_obj = &obj;     /* adresse de l'objet obj */
```

adr_obj est dit aussi "pointeur" vers l'objet **obj**, soit :

```
pointeur_obj = &obj;
```

L'opérateur **&** ne s'applique qu'aux variables et aux éléments de tableau. Il ne s'applique pas aux constantes et aux expressions, car l'adresse d'une constante ou d'une expression n'a pas de sens.

&i6 &(a*b + c — 8) sont invalides

De plus, il est incorrect de prendre l'adresse d'un registre ; le programme aurait des comportements différents selon le type de machine utilisée.

7.2. Adressage indirect

L'opérateur inverse de l'opérateur adresse **&** est l'opérateur * (*contenu de*) qui s'applique à l'expression qui le suit. Comme &, * est un opérateur unaire. Il ne faut pas le confondre avec l'opérateur de multiplication * qui, lui, s'applique à deux expressions.

L'opérateur indirection * permet d'atteindre le contenu de l'objet pointé par l'expression.

si `adr_obj = &obj`

alors `obj = *adr_obj`

soit

```
obj = *&obj
```

qui signifie :

obj = contenu de l'adresse de **obj**.

7.3. Adressage indexé

Cet adressage est tel qu'il permet, à l'aide d'une valeur appelée *déplacement* ou *index* par rapport à une adresse de base (en l'occurrence le nom de l'objet), d'accéder à l'élément choisi.

Les structures de données qui s'adaptent le mieux à ce mode d'adressage, sont en C les tableaux (*array of*).

Un tableau peut être monodimensionnel (*vecteur*) ou multidimensionnel (*matrice*). Les éléments du tableau sont tous du même type.

Exemple:

```
int tab[100] ;
float mat[20][20];
```

int tab[100] crée un tableau de 100 entiers (type **int**). L'adresse du tableau est donnée par: **tab** ou **&tab**[0], c'est-à-dire l'adresse du premier élément (l'indiçage commence à partir de 0).

Si **i** est un index de type **int**, l'adresse du **i** *ième* élément du tableau, indexé à partir de zéro, s'écrit:

```
tab + i
```

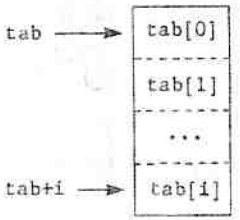
et l'accès à l'élément correspondant s'obtient, en utilisant l'opérateur d'indirection:

```
*(tab + i)
```

L'écriture étant un peut trop lourde, C propose un nouvel opérateur, l'opérateur [ ] d'indexation. L'accès à l'élément s'écrit alors:

```
tab[i]
```

Représentation des éléments d'un tableau



Notons que, quelque soit les types de données constituant un tableau, l'alignement automatique des index est réalisé. Par exemple, pour un mot machine de 16 bits:

```
char   tab[10];    tab[i+1] est l'octet qui suit l'octet tab[i]
int    tab[10];    tab[i+1] est le mot qui suit le mot tab[i]
float  tab[10];    tab[i+1] est le double mot qui suit le double mot tab[i]
```


8. L'OPÉRATEUR SIZEOF

Un autre opérateur, non encore répertorié dans notre classification, est l'opérateur symbolique **sizeof**. Celui-ci donne la taille en octets de l'opérande qui lui est associé.

Il trouve généralement son utilisation dans la détermination de la taille d'un tableau ou d'une structure, en vue d'une demande d'allocation mémoire (fonction **malloc**). Le résultat est toujours une constante entière.

Exemples:

```
char  tab[10];     sizeof (tab)    donne 10
int   tab[10];     sizeof (tab)    donne 20
float tab[10];     sizeof (tab)    donne 40
```

La forme **sizeof(type)** permet en particulier de connaître la taille des types de base. Son utilisation est conseillée pour améliorer la portabilité des programmes. La connaissance de la taille réelle d'un mot machine n'étant pas indispensable au programmeur.

Exemple:

```
printf("sizeof(char)  = %d\n",sizeof(char));
printf("sizeof(int)   = %d\n",sizeof(int));
printf("sizeof(short) = %d\n",sizeof(short));
printf("sizeof(long)  = %d\n",sizeof(long));
printf("sizeof(float) = %d\n",sizeof(float));
printf("sizeof(double)= %d\n",sizeof(double));
```

donne à l'exécution:

```
sizeof(char)  = 1
sizeof(int)   = 2
sizeof(short) = 2
sizeof(long)  = 4
sizeof(float) = 4
sizeof(double)= 8
```

9. CONVERSION DE TYPE

Les expressions qui font intervenir des objets ayant des types différents sont largement permises en C, contrairement à PASCAL et à ADA, qui eux sont intransigeants sur les opérations qui mélangent les types. Ainsi, en C, toute conversion de type au sein d'une expression ayant un sens, est possible.

Les changements de type qui ont un sens, se font en général de manière automatique par l'application de quelques règles de conversion. Sinon, C offre un opérateur explicite qui permet de forcer le type d'une expression. Il s'agit de l'opérateur **(*type*)** appelé aussi **"cast"** en anglais.

9.1. Conversions implicites

Le type **char** peut être considéré comme un entier court codé sur 8 bits. Les types **char** et **int** peuvent ainsi coexister au sein d'une même expression.

De manière générale, C convertit systématiquement les **char** en **int** dans une expression, ce qui permet une grande souplesse pour la transformation des caractères.

Ainsi :

```
'A' + 0x20    donne 0x61    soit   'a'
'1' + 1       donne 0x32    soit   '2'
'5' - '0'      donne 5
```

De même, une expression relationnelle qui fait intervenir des tests logiques sur des caractères donne un résultat entier vrai 1 ou faux 0.

Ainsi :

```
minuscule = c >= 'a' && c <= 'z'
```

est vrai si le caractère est une minuscule, faux si c'est un autre caractère.

Dans une expression arithmétique, les règles observées par le compilateur C pour convertir les types sont les suivantes :

char et **short** sont convertis en **int**.

float est converti en **double**.

Si un des opérandes est **double**, l'autre est converti en **double** et le résultat est du type **double**.

Sinon, si l'un des opérandes est **long**, l'autre est converti en **long** et le résultat est du type **long**.

Sinon, si l'un des opérandes est **unsigned**, l'autre est converti en **unsigned** et le résultat est du type **unsigned**.

Sinon les deux opérandes sont nécessairement du type **int**.

Tableau de conversion des types

oper1 \ oper2	char int short	unsigned	long	float double
char int short	int	unsigned	long	double
unsigned	unsigned	unsigned	long	double
long	long	long	long	double
float double	double	double	double	double

L'ordre d'affectation modifie également les types de manière implicite. On peut ainsi revenir du type **int** au type **char**, les bits de poids forts étant alors ignorés. Les transformations de type par affectation s'appliquent à tous les types de base. Des troncations ou des remplissages par des zéros sont effectués selon les cas.

Par exemple, les instructions suivantes sont valides :

```
char c;      /* c caractere */
int i;       /* i entier */
long l;      /* l entier long */
double d;    /* d flottant double precision */
int *p;      /* p pointeur d'un entier */

c = i;       /* l'entier devient caractere */
i = l;       /* l'entier long devient entier simple */
l = d;       /* le double flottant devient entier long */
p = i;       /* l'entier devient pointeur */
```

9.2. Conversions explicites

Dans la mesure où le programmeur désire effectuer une conversion explicite, il a à sa disposition le **"cast"** qui lui permet de forcer un type. La syntaxe est la suivante :

(type) *expression*

où type est un type de base (**char, int, short, long, float, double**) ou un pointeur (**char ***, **int ***, etc.).

Par exemple, les fonctions trigonométriques attendent un argument en double précision (type **double**). Si le programmeur utilise un nombre entier dans son programme, il doit alors écrire :

```
double sin(),cos();     /* declaration des fonctions */
main()
{
        int x,z;        /* x et y sont des entiers   */
        double y;       /* y est un double flottant  */
        ...
        y = sin((double) x);
        z = (int) cos( (double) x);
        ...
}
```

x est converti en double avant d'être transmis à la fonction **sin**, et l'expression **cos((double) x)** est forcée à une valeur entière.

10. PRÉCÉDENCE D'OPÉRATEUR

Lorsqu'une expression comporte plusieurs opérateurs, on effectue les opérations par ordre de priorité décroissante. A chaque opérateur est associé une priorité appelée précédence d'opérateur. Bien entendu, une expression entre parenthèses est évaluée en priorité indépendamment des opérateurs qui l'entourent.

Si les règles de précédence d'opérateur sont souvent mal connues du programmeur, ce dernier a toujours le recours aux parenthèses pour construire son expression.

Dans le tableau ci-contre, les opérateurs placés sur une même ligne sont d'égale précédence les lignes étant disposées par priorité décroissante, de la première à la dernière :

Classification des opérateurs selon leur précédence

() [] -> .
! ~ ++ -- - * & (type) sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
? :
= += -= *= /= %= >>= <<= &= |= ^=
,

11. ÉVALUATION DES EXPRESSIONS

Lorsqu'une expression comporte plusieurs opérateurs de même priorité, celle-ci est évaluée de la gauche vers la droite, c'est-à-dire dans le sens normal de la lecture de l'expression. Dans d'autres cas et non des moindres, l'affectation par exemple, l'évaluation de l'expression à droite du signe = s'effectue bien de la gauche vers la droite, mais une fois cette évaluation partielle terminée il est nécessaire d'affecter la variable à gauche du signe =, ce qui revient à effectuer une évaluation de la droite vers la gauche pour l'expression globale.

Dans l'expression suivante :

```
y = a * b - c / d ;
```

l'expression partielle **a * b - c /d** est évaluée de la gauche vers la droite. Notons son résultat **expr_droite** et dans ce cas d'évaluation finale s'effectue bien de la droite vers la gauche pour réaliser l'affectation :

```
y = expr_droite ;
```

Le mécanisme d'évaluation de la droite vers la gauche s'illustre parfaitement dans le cas d'affectations multiples. En effet dans l'exemple ci-dessous :

```
i = j = k = 0;
```

la constante 0 est affectée à **k**, puis **j = k** et enfin **i = j**.

L'évaluation des expressions, lorsque la priorité des opérateurs ne joue pas, répond aux règles suivantes :

- évaluation de la droite vers la gauche pour les opérateurs unaires ou d'affectation.
- évaluation de la gauche vers la droite pour tous les autres opérateurs.

12. AUTRES OPÉRATEURS

Il existe d'autres opérateurs d'accès à certaines structures de données comme les structures ou les unions. Nous étudierons les opérateurs d'adressage "**.**" et "**->**" relatifs à ces structures dans le chapitre qui leur est consacré.

IV

LES INSTRUCTIONS

1. GÉNÉRALITÉS

Un programme C est constitué d'un ensemble d'objets externes, données et fonctions. Les fonctions elles-mêmes, comportent des données locales et des suites d'instructions exécutables. Le déroulement des instructions dans le corps d'une fonction se fait séquentiellement sauf dans le cas de rupture de séquence, de terminaison de boucle ou de retour de fonction.

On peut considérer qu'il existe deux catégories d'instructions :

- les instructions simples,
- les instructions de contrôle.

2. INSTRUCTIONS SIMPLES

La forme la plus générale pour représenter une instruction simple est la suivante :

expression;

où *expression* nous l'avons déjà dit, peut être :

- soit une affectation de la forme :

variable = expression;

variable étant une *g_expression* ou *lvalue*, et où *expression* peut représenter :

une constante,
une variable,
une expression arithmétique ou logique,
une expression relationnelle (résultat booléen),
une fonction retournant une valeur,

— soit un appel de fonction ne nécessitant pas d'affectation (fonction au sens procédure de PASCAL).

Exemples :

```
i = 0;
x = index;
y = a * (x * x) + b * x + c ;
n++;
i = j = k = 0;
c = getchar();
vrai = (faux != 0);
false= (!true);
putchar(c);
printf("valeur de z = %d",z);
```

Toute instruction d'affectation ou d'appel de fonction doit obligatoirement se terminer par le caractère "**;**". Ce caractère n'est pas un séparateur d'instruction comme en PASCAL, mais un terminateur d'instruction.

Une instruction simple peut être vide, dans ce cas elle n'engendre aucun code.

instruction vide = ;

3. INSTRUCTIONS DE CONTRÔLE

Les instructions de contrôle permettent de définir l'ordre dans lequel doit s'exécuter l'ensemble des autres instructions du programme. C possède un jeu d'instructions de contrôle réduit mais néanmoins largement suffisant :

if , else , else if	: instructions conditionnelles
switch , case , default	: instructions d'aiguillage
while , do while , for	: instructions répétitives
break , continue	: instructions associées aux boucles
goto	: instruction de branchement
return	: instruction de retour de fonction

4. INSTRUCTION COMPOSÉE

Une instruction composée représente généralement une séquence d'instructions simples (affectation, appel de fonction). On rencontre les instructions composées :

— soit à la suite d'une définition de fonction : c'est le corps de la fonction,
— soit à la suite de certaines instructions de contrôle qui attendent une seule instruction en séquence.

Dans les deux cas, une instruction composée, appelée aussi "*bloc*", est syntaxiquement équivalente à une instruction simple.

A la différence d'une instruction simple qui se termine par le caractère "**;**", une instruction composée doit toujours être délimitée par des accolades { et }. L'accolade ouvrante { s'apparente au "*begin*" de PASCAL, tout comme l'accolade fermante } au "*end*". Le choix des accolades par rapport au *begin end* rend le programme moins lourd et plus aéré en C qu'en PASCAL.

D'ailleurs, tout comme en PASCAL, la présentation d'un programme écrit en C est libre. Seuls, l'espace ou blanc et le caractère de tabulation jouent le rôle de séparateurs à l'intérieur d'une ligne *source*. Bien entendu, on n'a pas le droit de couper en deux un identificateur, un mot clé ou une expression, sauf dans le cas de la virgule qui est un séparateur de liste (liste d'arguments). Mais le programmeur a la possibilité de mettre autant d'espaces qu'il le désire entre deux expressions.

Cette liberté de présentation dans l'écriture d'un programme doit être utilisée à bon escient pour rendre le programme le plus lisible possible. Le programmeur doit en fait s'appliquer à faire ressortir la structure logique de son programme, c'est-à-dire son découpage en blocs.

Une des solutions est de recourir à l'indentation qui consiste à utiliser les tabulations pour décaler l'ensemble des instructions d'un même bloc par rapport à un autre bloc de niveau inférieur. Cette technique d'écriture permet ainsi de faire apparaître avec davantage de visibilité les différents niveaux d'imbrication des boucles ou des séquences alternatives (**if else**).

Le contenu de l'instruction composée peut comporter des déclarations non exécutables et des instructions simples ou d'autres instructions composées. L'exemple le plus courant d'instruction composée est celui d'une fonction.

```
fonction(...)        /* objet externe vu de l'exterieur */
{                    /* comme une seule instruction     */
        déclarations
            ...
        instructions
            ...
}
```

Ce schéma peut se généraliser comme suit:

```
fonction( )
{
        déclarations niveau 1
        instructions niveau 1
        {
                déclarations niveau 2
                instructions niveau 2
                {
                        instructions niveau 3
                }
                instructions niveau 2 (suite)
        }
}
```

Comme le montre le schéma ci-dessus, on remarque que la partie déclaration **niveau 3** est absente, ce qui implique que les déclarations sont facultatives. De même, la partie instructions est également facultative; dans ce cas il n'y a pas d'exécution. Par exemple, le fichier <**stdio.h**> ne comporte que des déclarations.

Si dans une instruction composée les parties déclarations et instructions sont absentes, on est alors en présence d'une instruction vide { } qui n'engendre donc aucun code.

```
instruction composée vide  { }
```

Exemple de fonction vide:

```
vide()
{
}
```

L'emploi de fonctions vides trouve généralement son utilisation dans les phases de mise au point de gros programmes où toutes les fonctions non vitales n'ont pas encore été programmées, mais sont néanmoins nécessaires à l'architecture globale du programme.

5. INSTRUCTIONS CONDITIONNELLES

5.1. Instruction if else

Dans un programme il est fréquent de vouloir effectuer une action qui dépend d'une condition; c'est ce que l'on appelle un test. L'instruction C qui permet d'effectuer ce test est l'instruction **if**. Celle-ci peut aussi être associée à l'instruction **else** qui est facultative. Par contre la clauqe **then**, qui est intuitive, n'existe pas en C contrairement à PASCAL (*if... then... else*).

On en déduit les deux formes suivantes:

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

Dans le cas général (**if else**), *expression* qui est une expression au sens large (expression relationnelle, arithmétique, logique ou conditionnelle), est évaluée. Si elle est vraie (valeur résultante différente de 0) alors il y a exécution de l'instruction qui suit le **if** et saut de l'instruction qui suit le **else**.

Si l'expression donne un résultat faux (valeur = 0) alors l'instruction qui suit le **if** est ignorée et c'est l'instruction qui suit le **else** qui est exécutée. Si le **else** est absent, l'exécution continue après l'instruction qui suit le **if**.

Exemple:

```
if (n > 0)
        printf("n = %d est positif",n);
else
        printf("n = %d est negatif ou nul",n);
```

Du fait que le **else** soit facultatif, une ambiguïté peut survenir lorsque l'instruction qui suit un **if** est elle-même un **if**; on dit que les **if** sont imbriqués.

```
if (e1) if (e2) action1 else action 2
```

Pour lever l'ambiguïté, on adopte la règle qui associe le premier **else** rencontré avec le plus proche **if** qui le précède. Dans le cas de l'exemple ci-dessus **action2** sera exécuté si **e1** est vrai et si **e2** est faux.

De même:

```
if (d != 0)                    /* test du denominateur *,
     if (r = n%d)              /* calcul du reste r    *
          printf("n%%d soit %d%%%d=%d",n,d,r);
     else
          printf("le reste est nul");
```

est différent de:

```
if (d != 0) {
     if (r = n%d)
          printf("n%%d soit %d%%%d=%d",n,d,r);
}
else
     printf("attention : le diviseur est nul");
```

Au passage, faisons une remarque concernant le test : **if (r = n%d)**. Cette condition se lit :

si **r** qui est égal à **n** modulo **d** n'est pas nul

ce qui pouvait s'écrire aussi :

if ((r = n%d) != 0)

mais ne se lit pas :

si **r** est égal à **n** modulo **d**

ce qui se serait écrit : if (r == n%d)

Ne pas confondre l'affectation = avec le test d'égalité ==.

5.2. Instruction else if

De la même manière qu'un **if** peut suivre immédiatement au autre **if**, un **if** peut également suivre un **else**. La construction d'une telle structure de contrôle répond à la syntaxe suivante :

```
if (expr_1)
        action_1
else if (expr_2)
        action_2
else if (expr_3)
        action_3
else
        action_4
```

Les expressions sont évaluées dans l'ordre. Si une expression est vraie, l'instruction associée est exécutée, et la reprise s'effectue à la fin de la chaîne des **else**. Le dernier **else** traite le cas où toutes les conditions ne seraient pas satisfaites. Dans le cas où il n'y a pas d'action finale, ce **else** peut être omis.

L'utilisation du **else if** peut s'illustrer par la fonction suivante dont le rôle est d'analyser des commandes utilisateur et de vérifier si elles existent.


```
command()
{
    int r;
    char cmd[16];          /* tampon : nom de la commande */

    while(1) {             /* boucle centrale */
        r = getcom(cmd);   /* lecture d'une ligne */
        if (r == EOF)      return (EOF); /* fin des commandes */
        else if (anacom(cmd,"creer"))     creer();
        else if (anacom(cmd,"supprimer")) supprimer();
        else if (anacom(cmd,"ajouter"))   ajouter();
        else if (anacom(cmd,"inserer"))   inserer();
        else if (anacom(cmd,"changer"))   changer();
        else if (anacom(cmd,"lister"))    lister();
        else
             printf("commande %s inconnue\n",cmd);
    }
}
anacom(ligne,com)
char *ligne;
char *com;
{
    while(*com) {          /* voir while et pointeur plus loin */
        if (*ligne++ != *com++)
            return (0);    /* ne correspond pas */
    }
    if ((*ligne == '\0')
     || (*ligne++ == ' ') || (*ligne == '\t'))
        return (1);        /* correspond */
    else
        return (0);
}
```

6. INSTRUCTION D'AIGUILLAGE

Les constructions de la forme **else if** ne sont pas très élégantes s'il y a beaucoup de conditions. D'autre part, elles sont pénalisantes au niveau de l'exécution proprement dite, car les tests sont séquentiels.

L'instruction **switch** est une instruction de choix multiple qui permet d'effectuer un aiguillage direct vers les actions (instructions) en fonction d'un cas matérialisé par la clause **case**.

Le *switch case* de C est le cousin du *case of* de PASCAL. Une amélioration cependant à l'actif du **switch** est dûe à la clause **default**. Cette instruction permet en effet de résoudre tous les autres cas non explicitement déclarés par les clauses **case**.

La syntaxe globale de l'instruction **switch** est la suivante :

```
switch (expression) {
case constante : instruction
case ...       :   ...
case ...       :   ...
default : instruction
}
```

Les arguments de **case** doivent être obligatoirement des constantes entières (**int**, **char**, **short**, **long**, **unsigned**), soit sous forme numérique, soit sous forme symbolique. C'est une des restrictions par rapport à l'instruction **else if** qui, elle, peut accepter toute sorte d'expression.

Plusieurs **case** peuvent référencer une même action ; dans ce cas ils sont disposés en séquence et c'est le dernier **case** qui référencera la séquence d'instructions associée.

Exemple :

```
/* fonction qui teste si c est une voyelle */
voyelle(c) char c;
{
    switch (c) {
        case 'a' :
        case 'e' :
        case 'i' :
        case 'o' :
        case 'u' :
        case 'y' : return(1);
        default  : return(0);
    }
}
```

Pour sortir d'une instruction **switch**, on doit utiliser l'instruction **break** qui est détaillée plus loin.

Exemple :

```
erreur(error)                   /* fonction erreur */
int error;
{
        switch (error) {
        case 0 : break;         /* pas d'erreur */
        case 1 :
        case 2 :
        case 3 :    printf("erreur numero %d",error); break;
        case -1:    printf("erreur catastrophique"); return;
        default:    printf("erreur %d inconnue",error), break;
        }
}
```


7. INSTRUCTIONS RÉPÉTITIVES

7.1. Instruction while

L'instruction **while** permet de répéter une suite d'actions tant qu'une condition est réalisée (condition vraie). Sa syntaxe est la suivante :

```
while (expression)
    instruction
```

où, bien entendu, *instruction* peut être une instruction composée.

On remarque que le test est effectué au début de la boucle, on dit aussi au sommet de la boucle. Ce qui revient à dire que si l'expression donne une valeur fausse dès le début, la boucle ne sera jamais exécutée. Si l'évaluation de l'expression est vraie, la boucle est exécutée jusqu'à ce que l'expression donne une valeur fausse.

Il est évident que l'une des actions internes de la boucle doit modifier à un moment donné une variable utilisée dans l'expression pour que la condition devienne fausse et permette ainsi de sortir proprement de la boucle. Pourtant, nous verrons plus loin qu'il existe d'autres moyens, plus autoritaires, pour quitter une boucle (instructions **break** et **return**).

Exemples :

```
while (n > 0) {          et      while (n--) {
    ...                                  ...
    n--;                                 ...
    ...                                  ...
}                                    }
```

sont des constructions équivalentes.

La partie instruction peut être vide, on a alors une boucle du type :

```
while (expression);
```

dans laquelle *expression* peut par exemple comporter une affectation.

```
while ((c=getchar()) == ' ' || c == '\t') ; /* saut des espaces */
```

7.2. Instruction do while

L'instruction *do ... while* est une instruction de boucle dont le test de la condition est effectué en fin de boucle. Elle s'apparente à l'instruction *repeat ... until* de PASCAL. Sa syntaxe est :

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

On constate que cette boucle est exécutée au moins une fois quelque soit la valeur de l'expression. La boucle est exécutée à nouveau tant que la condition reste vraie, et se termine dès que la condition devient fausse (valeur nulle). Cette instruction doit se terminer par un ";" qui indique la fin de l'instruction **do while**.

Exemple:

```
/* conversion d'un entier en une suite de chiffres binaires */

ltob(i,s)
int i, char s[];        /* declaration des arguments */
{
    int x=0;            /* x = index */
    int n = (sizeof(i) * 8) - 1; /* n = nombre de decalages */
    do {
        s[x++] = ((i<0) ? '1' : '0'); /* 1/0 selon le signe */
        i <<= 1;        /* decalage vers la gauche d'un bit */
    }
    while (n--);        /* jusqu'au dernier bit  */
    s[x] = '\0';        /* fin de chaine = NUL   */

}
```

7.3. Instruction for

Les instructions du type "for" telles que le *"for to step next"* de BASIC ou de *"for to do"* de PASCAL ou encore le *"do continue"* de FORTRAN, font généralement intervenir l'initialisation d'un compteur, le test de fin de boucle sur le compteur et aussi son pas de progression. Dans ces langages, l'instruction **for** permet de regrouper ces trois actions en une seule. C'est le cas également du **for** en C, mais qui est plus général.

Par exemple au lieu d'écrire:

```
n = 1;
while (n < 10) {
        printf("n=%d n**2=%d\n",n,n*n);
        n++ ;
}
```

ou:

```
n = 1;
do {
        printf("n=%d n**2=%d\n",n n*n);
        n++ ;
}
while (n <= 10) ;
```


on peut écrire en une seule instruction:

```
for (n = 1 ; n <= 10 ; n++)
        printf("n=%d n**2=%d\n",n,n*n);
```

La syntaxe de l'instruction **for**:

for (expr1; expr2; expr3)
instruction

est identique à la construction suivante:

expr1;
while (*expr2*)
instruction
expr3;

Les trois membres de l'instruction **for** sont des expressions au sens large. Dans la pratique:

expr1 est une expression d'initialisation de la boucle. Elle n'est exécutée qu'une seule fois; c'est généralement une affectation.

expr2 est une expression relationnelle dont le résultat vrai ou faux détermine la continuation ou la terminaison de la boucle.

expr3 est une expression qui agit la plupart du temps sur la valeur de la condition (progression d'un compteur par exemple).

L'instruction **for** peut alors s'exprimer sous une forme plus parlante de la façon suivante:

for (*initialisation*; *test de la condition*; *incrémentation*)

Ces trois composantes sont sur la même ligne, ce qui rend la lecture du programme plus claire. Chacune des composantes de l'instruction **for** est facultative ce qui permet d'avoir les configurations suivantes:

```
for ( ; test ; incr)      /* pas d'initialisation     */

for (init ; test ; )      /* pas de progression       */

for ( ; test ; )          /* identique a while(test) */

for (;;)                  /* boucle infinie while(1) */
```

De plus le **for** de C ne se contente pas de faire progresser un compteur et de tester une expression arithmétique du type:

```
do 10 i=1,10,1 (FORTRAN)   ou   for i:=1 to 10 do (PASCAL)
```

soit:

```
for (i = 1 ; i <= 10 ; i++ ) (C)
```

Le **for** de C est plus général que ceux des autres langages. Il n'existe aucune restriction sur les expressions qui composent une instruction **for**. Les trois champs de l'instruction peuvent à priori n'avoir aucun lien entre eux. Par exemple, le **for** qui suit est valide:

```
for (i=j=0 , k=1 , a=b ;
                        (c = getchar()) != EOF
                        && c != '\n' ;
                                        i++ , k<<=1 , p--)
```

Nous savons qu'un programme C peut être constitué d'instructions composées. Ces instructions sont délimitées par des accolades ouvrantes et fermantes. Dans des programmes relativement importants, comportant de nombreuses imbrications, le programmeur peut oublier de refermer une instruction composée. Le compilateur ne sachant pas où doit se terminer cette instruction continue la compilation sans mentionner d'erreur jusqu'au moment où il détectera inéluctablement une incohérence.

Pour éviter au programmeur de chercher une autre erreur, on peut écrire un petit programme qui comptabilise les accolades. En fait, le programme de l'utilisateur est nécessairement dans un fichier mais nous ne disposons pas encore des outils d'accès aux fichiers qui seront abordés dans le chapitre relatif à l'environnement de C.

Cependant la subtilité d'UNIX nous donne la possibilité de résoudre de manière élégante le problème des entrées-sorties. L'interpréteur de commande "Shell" permet de rediriger la sortie standard d'un programme vers l'entrée standard d'un autre programme. C'est ce que l'on appelle un "filtre". Dans ce cas, la fonction **getchar( )** nous suffit pour lire un fichier listé par la commande **cat** d'UNIX, de la manière suivante:

```
$cat fichier | accolade
```

Ce qui nous donne le programme "accolade" suivant:


```
/*  programme de verification
    du nombre d'accolades
    dans un programme C       */

#include <stdio.h>          /* pour la constante EOF */
main()
{
        char c;
        int begin,end;      /* compteurs d'accolades */
        for (begin=0 , end=0 ; (c=getchar()) != EOF ;){
                switch (c) {
                case '{' : ++begin; break;
                case '}' : ++end  ; break;
                default  : break;
                }
        }
        printf("nombre de { = %d\n",begin);
        printf("nombre de } = %d\n",end);
        if (begin - end) printf("erreur de {}\n");
}
```

8. INSTRUCTIONS ASSOCIÉES AUX BOUCLES

En plus des sorties normales effectuées sur des conditions, sortie en tête de boucle (**while, for**), sortie en fin de boucle (**do while**), C fournit des instructions de contrôle internes aux boucles qui permettent soit de sortir d'une boucle soit de s'y déplacer. Il s'agit des instructions suivantes:

```
break
continue
goto
;
```

8.1. Instruction break

L'instruction **break** provoque une sortie immédiate de la boucle en cours d'exécution. Le **break** n'est limité qu'à un seul niveau d'imbrication. La sortie de la boucle s'effectue sur l'instruction qui suit l'instruction de boucle. Comme nous l'avons vu précédemment, **break** est également la seule issue aux différents cas (clauses **case** et **default**) de l'instruction **switch**.

Exemple:

```
/* lecture d'un nom de longueur <= LGMAX caracteres */

lirenom(name,LGMAX)
int LGMAX;
char name[];
{
    register char c;
    register i;

    for (i = 1 ; i < LGMAX ; i++ ) {
        c = getchar();
        if (c == '\0')  break;      /* fin de texte */
        name[i] = c;                /* caractere normal */
    }
    name[0] = --i;                  /* taille du texte */
}
```

L'utilisation du **break** est également indispensable pour des boucles infinies de la forme:

```
while(1) {                ou        for (;;) {
    ...                                 ...
    if (exp) break;                     if (exp) break;
    ...                                 ...
}                                   }
```

8.2. Instruction continue

A la différence du **break**, l'instruction **continue** donne le contrôle à la fin de la boucle sans en sortir, en sautant ainsi la séquence d'instructions comprise entre le **continue** et la fin de la boucle. L'instruction **continue** a ainsi pour effet de provoquer une nouvelle itération. Ce qui revient à redonner le contrôle au niveau du test de la condition. Dans le cas du **for**, le troisième champ (par exemple l'itération d'un compteur) n'est pas exécuté.

Illustration du mécanisme du **continue**

```
while (expression) {          do {
    ...                           ...
    ...                           ...
    continue ;                    continue ;
    ...                           ...
    ...                           ...
}                             } while (expression) ;
```


Exemple d'un programme qui purge les caractères spéciaux non imprimables à l'exception du "newline" et de la tabulation. Ces caractères sont situés dans la tranche 0 à 31 de la table ASCII.

```
/*programme "clear" qui elimine les caracteres speciaux */

#include <stdio.h>
main()
{
    char c;
    int kspe;

    while ((c=getchar()) != EOF) {
        if (c < 32 && c != '\t' && c != '\n') {
            ++kspe;
            continue;       /* caractere special */
        }
        putchar(c);         /* caractere normal  */
    }
    if (kspe)
        printf("%d caracteres speciaux elimines",kspe);
}
```

Ce programme ne manipulant que des données en provenance du fichier standard d'entrée (fonction **getchar**), et ne restituant des données que sur le fichier standard de sortie (fonctions **putchar** et **printf**), peut être utilisé comme un *filtre* au niveau d'une commande **Shell**.

La commande utilisateur peut alors avoir l'allure suivante:

```
$cat fichier | clear
```

8.3. Instruction goto

L'instruction **goto**, telle qu'elle est pratiquée dans le langage BASIC est devenue une instruction contreversée avec l'apparition des langages structurés. Pourtant, que ce soit en C, en PASCAL ou en ADA, elle a été conservée, peut-être par souci du *Passé*, mais aussi pour donner au programmeur un moyen lui permettant de se sortir efficacement de situations délicates ou complexes (imbrication de plusieurs niveaux de boucles par exemple).

En fait, tout programme peut être écrit sans une seule ligne de **goto**, mais cela au prix d'une certaine difficulté de programmation, donc au détriment de la lisibilité du programme.

L'instruction **goto** paraît être l'outil le mieux approprié pour pallier l'insuffisance de l'instruction **break** qui, elle, ne se limite qu'à un seul niveau de boucle.

Du point de vue de la syntaxe, le **goto** est associé à un identificateur symbolique spécifiant l'endroit où doit être effectué le branchement inconditionnel.

La syntaxe du **goto** est la suivante :

```
            ...
            goto label ;
            ...
label:
```

où **label** est un identificateur qui doit être suivi du caractère ":" (comme **case** et **default**).

Exemple :

```
while ((c=getchar()) != EOF) {
        for (i = 0 ; c != '\n' ; i++) {
                if (c == 0x7f) goto erreur;
                ...
        }
        ...
}
erreur: ...
```

8.4. Instruction ;

Le caractère ; délimiteur d'instruction peut être considéré comme une instruction vide s'il est isolé.

Cette instruction permet en particulier de terminer une boucle **while** sans corps de boucle, c'est le cas d'un test répétitif sur la condition du **while**.

Exemple :

```
while ( (c = getchar()) == ' ' || c =='\t') ;
```

qui permet d'ignorer les caractères séparateurs successifs (espace et tabulation).

Une autre utilisation de l'instruction ; permet de placer un identificateur de **goto** juste avant la fin d'une instruction composée.

9. INSTRUCTION DE RETOUR DE FONCTION

L'instruction **return** permet à la fonction en cours de retourner une valeur à la fonction appelante. Un programme C, à l'exception des données externes, n'est constitué que de fonctions. Le programme principal **main** est lui-même une fonction.

L'instruction return peut prendre trois formes syntaxiques :

return ;	(1)
return *constante* ;	(2)
return (*expression*) ;	(3)

Dans le premier cas la valeur retournée à la fonction appelante est indéfinie, et la fonction peut être considérée comme une procédure au sens PASCAL. D'ailleurs, le compilateur se charge de rajouter automatiquement, à la fin d'une fonction, une instruction **return** de ce type, dans le cas où celle-ci n'y serait pas explicitement mise par le programmeur.

Dans le second cas la valeur retournée est une constante entière.

Dans le troisième cas, l'expression est évaluée et sa valeur est transmise à l'appelant après avoir subi une conversion de type si le type du résultat de l'expression est différent de celui de la fonction. Cette conversion est implicite pour les fonctions de type **int** et **char**. Pour les autres types, s'il n'y a pas concordance entre le type de la fonction et le type de la valeur à retourner, l'utilisation de l'opérateur **"cast"** s'avère nécessaire.

Exemple :

```
minuscule(c)      /* MAJUSCULE ==> minuscule */
char c;
{
        return ( (c >= 'A' && c <= 'Z') ? c + 32 : c);
}

majuscule(c)      /* minuscule ==> MAJUSCULE */
char c;
{
        return ( (c < 'a' || c > 'z') ? c : c - 32);
}

precision(n)
int n;
{
        ...
        ...
        return ( (double) n);
}
```

L'instruction **return** peut être également utilisée dans les boucles et provoquer ainsi une sortie directe vers la fonction appelante, évitant ainsi de programmer des **break** ou des **goto** vers la fin de la fonction.

Soit la fonction **readline(tampon,max)** qui remplit un tampon appartenant à la fonction appelante à partir des caractères lus sur le fichier standard d'entrée (**stdin**). L'argument **max** représente la taille maximale en caractères du tampon. On utilise pour cela une boucle infinie **while(1)**. Le contrôle est redonné à la fonction appelante par les instructions **return** dans les cas suivants: fin de fichier (EOF), fin de ligne (newline) ou débordemet de la taille du tampon hôte. La valeur retournée par **readline** représente le nombre de caractères utiles lus dans la mesure où la ligne se termine normalement.

```
readline(tampon,max)
char tampon[];
int max;
{
   register c , lg ;
   lg = 0;
   while (1) {
       c = getchar();          /* lecture du caractere */
       if (c == EOF)     return(EOF);   /* fin de fichier */
       if (c == '\n')    return(lg);    /* newline */
       if (lg >= max)    return(max);   /* debordement */
       tampon[lg++] = c;       /* cas normal */
   }
}
```

L'appel à la fonction **readline** pourrait par exemple se réaliser de la manière suivante:

```
#define EOF -1      /* voir #define au chapitre suivant */
#define MAX 256     /* taille du tampon = 256 */
char buffer[MAX];   /* declaration du tampon */

main()
{
    register size;
    while ( (size = readline(buffer,MAX)) != EOF ) {
        ...
        ...
    }
}
```


10. DIRECTIVES DE COMPILATION

A la différence des instructions, qui engendrent du code exécutable, les directives de compilation ne sont pas exécutables au niveau de la machine. Les directives du langage C sont interprétées par un module séparé du compilateur C appelé le préprocesseur C. Pour que le préprocesseur puisse reconnaître les directives dans le texte source, celles-ci commencent toujours par le symbole "#", ce qui permet de les différencier des autres identificateurs. Ce préprocesseur est lancé automatiquement avant la compilation pour préparer définitivement le fichier source à compiler.

On distingue trois catégories de directives:

- la directive de substitution symbolique: #define,
- la directive d'inclusion de fichier source: #include,
- les directives de compilation conditionnelle: #if, #else, #endif, #ifdef, #ifndef.

10.1. Substitution symbolique: #define

La substitution symbolique consiste à remplacer dans la suite du texte source toutes les occurrences d'un symbole défini par une suite de caractères qui lui est associée. C'est la directive **#define** qui permet de définir au préalable le symbole et son texte équivalent. La syntaxe de la directive de substitution peut prendre les deux formes suivantes:

#define *symbole suite_de_caractères* (1)

#define *symtole(liste_de_symboles) suite_de_caractères* (2)

Dans les deux cas, **symbole** et **symbole(liste_de_symboles)** sont des identificateurs qui devront être substitués par la suite de caractères, placée en vis-à-vis. Celle-ci constitue normalement le reste de la ligne compris entre le premier séparateur et la fin de la ligne. Pour une longue définition, il est possible de replier la ligne au moyen du caractère d'échappement \ suivi d'un retour à la ligne (newline).

Le domaine de visibilité du symbole défini par **#define** débute à partir de sa définition jusqu'à la fin du fichier source, ou jusqu'à sa prochaine définition (nouveau **#define** avec le même symbole), ou encore jusqu'à son invalidation par la directive:

#undef *symbole*

Afin d'assurer une meilleure lisibilité du programme les constantes symboliques sont le plus souvent mises en lettres majuscules pour les distinguer des autres variables. Il est conseillé d'utiliser la directive **#define** pour des constantes qui doivent se répéter souvent dans le programme source. Il est en effet préférable de modifier une seule ligne **#define** que toutes les lignes où apparaissent la constante en question.

Exemples:

```
#define DIM  100                    /* dimension 100 */
#define LF   '\n'                   /* line-feed */
#define EOF  -1                     /* fin de fichier */
#define XON  0x11                   /* autorisation */
#define XOFF 0x13                   /* suspension   */
#define NOM_FICHIER "/usr/nom/projet1/fichier2"
#define NB_ENTREES  (sizeof(tab) / sizeof(tab[0]))
```

Dans la seconde forme syntaxique, la parenthèse ouvrante doit être accolée au premier symbole sinon ce serait tout le reste du texte qui suit le symbole qui alors, serait pris comme la suite de caractères à remplacer à chaque occurrence. Le premier espace ou caractère de tabulation rencontré après le symbole joue le rôle de caractère séparateur, tandis qu'il est non significatif dans la suite de caractères à laquelle il est associé.

Exemple:

```
#define abs(n) ((n > 0) ? n : -n)
```

définit la fonction **abs(n)** qui pourra être utilisée dans le programme sous la forme suivante:

```
abs(valeur);
```

La directive **#define** permet ainsi de définir des "macro-instructions" qui peuvent parfois remplacer des déclarations de fonctions. Les arguments d'une "macro-instruction", considérés comme des "macro-arguments" sont évalués chaque fois qu'ils apparaissent dans la définition. Ainsi, si on définit la "macro-instruction" qui prend le maximum de deux valeurs:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

l'appel:

```
max(x++, y++)
```

incrémentera soit **x** soit **y** deux fois.

Le parenthésage est fortement conseillé pour éviter les effets de bord. Puisque les "macros" sont évaluées par le préprocesseur C, des résultats imprévus pourraient survenir selon la précédence des opérateurs. Ainsi, si la définition

précédente de **mas** était non parenthésée, l'expression:

```
max(max(x,y), z)
```

produirait:

```
x > y ? x : y > z ? x > y ? x : y : z
```

qui serait mal interprétée par C. L'expression correcte produite par le préprocesseur avec la définition parenthésée de **max** produirait l'expression suivante, correcte pour C, mais illisible pour l'utilisateur:

```
((((x)>(y) ? (x) : (y)))>(z) ? (((x)>(y) ? (x) : (y))) : (z))
```

La directive **#define** permet également de remplacer n'importe quelle chaîne de caractères par un symbole, en particulier, les fonctions ou tout simplement les opérateurs et les mots clés du langage.

Exemples:

```
#define ADD +                   /* addition */
#define AND &                   /* ET binaire */
#define MOD %                   /* modulo */

#define forever while(1)        /* boucle infinie */
#define forever for(;;)         /*       ...      */
```

avec:

```
forever {
    ...
    if(exp) break;
    ...
}

#define forall for(i=0 ; i<MAX ; i++)
#define lirecar() getchar()
#define retour(e) return(e)
```

Je laisse au lecteur le soin d'imaginer une programmation en C avec de nouvelles définitions d'opérateurs, des mots réservés français ou appartenant à un autre langage structuré relativement voisin de C (PASCAL par exemple).

Si le symbole d'un **#define** n'est associé à aucun texte, chaque fois que ce symbole sera rencontré dans le fichier source il sera remplacé par rien.

```
# define PROGRAMME_ESSAI
# define MACHINE_ONYX_C8002
# define AUTEUR_Durand
```

10.2. Inclusion de fichiers source #include

La directive **#include** permet d'insérer, à la place de la ligne **#include** du fichier source courant à compiler, le contenu d'un autre fichier source dont le nom est donné en argument. La syntaxe de la directive **#include** peut prendre deux formes qui sont les suivantes:

include "nom de fichier" (1)

et:

include < fichier spécial > (2)

Dans la première forme, **"nom de fichier"** identifie un fichier appartenant à l'utilisateur. Sous le système UNIX ce fichier appartient au *directory* courant sous lequel travaille l'utilisateur.

Exemples:

```
#include "define"      /* inclusion de constantes */
#include "declaration" /* inclusion de declarations */
#include "utilitaires" /* inclusion de programmes */
```

Si ce fichier se trouve dans un autre directory, le programmeur doit alors spécifier le nom du chemin (*pathname*), pour y accéder.

```
# include "/usr/dupont/source/fich1"
# include "../prog_C/fich2"
```

où .. indique qu'il faut remonter au directory père pour redescendre dans le directory prog_C pour trouver le fichier **fich2**.

La seconde forme de **#include** est plus familièrement utilisée dans un contexte UNIX. Le nom du fichier est délimité par les signes < et >, ce qui indique que le préprocesseur doit aller chercher le fichier à inclure dans un directory spécialement affecté. Il s'agit du directory **/usr/include**, qui regroupe les fichiers où sont décrites les déclarations nécessaires aux fonctions de la Librairie Standard. Ces fichiers sont reconnaissables par leur suffixe **".h"**, qui spécifie qu'il s'agit de fichier en-tête (*head*). Parmi eux et le plus utilisé, le fichier **stdio.h** comporte les déclarations d'accès aux fonctions d'entrées-sortie (**getchar**, **putchar**, **stdin**, **stdout**...) et quelques **#define** tels que EOF, NULL, etc.

Exemple:

```
# include < stdio.h >
# include < ctype.h >      /* types de caractère */
```

Une des principales utilisations des fichiers **include** est de fournir des déclarations usuelles, des définitions de variables globales, et des constantes symboliques prédéfinies au programme appelant qui les utilisera. Un fichier appelé par **#include** peut également contenir une ou plusieurs fonctions C.

A la limite un programme C peut s'écrire sous la forme condensée suivante:

```
#include <stdio.h>    /* declarations de la librairie */
#include "define"     /* #define de constantes */
#include "globales"   /* declarations des variables globales */
#include "main"       /* programme principal */
#include "fonct1"
#include "fonct2"
#include   ...
```

La modification d'un fichier **include** doit nécessairement entraîner la recompilation du ou des fichiers (compilations séparées) qui lui font référence.

Les directives **#include** peuvent être imbriquées. En effet, un fichier appelé par **#include**, peut comporter lui aussi des lignes **#include** qui lui sont propres, et ainsi de suite.

10.3. Compilation conditionnelle

La compilation conditionnelle permet de compiler des lignes de C si une condition est vérifiée.

Syntaxe:

```
# if expression
...
séquence compilée si expression vraie (différente de 0)
...
# else
...
séquence compilée si expression fausse (égale à 0)
...
# endif
poursuite de la compilation
```

L'expression peut contenir des parenthèses, des opérateurs unaires –, !, ~, des opérateurs binaires +, –, *, /, %, | , <<, >>, <, <=, >, >=, !=, &&, || ainsi que l'opérateur ternaire ? :.

La directive **#else** peut être facultative, et dans ce cas si l'expression est fausse, la séquence entre le **#if** et **#endif** n'est pas compilée. Ces constructions peuvent également être imbriquées.

Deux autres directives, **#ifdef** et **#ifndef**, s'identifient au **#if**, à la différence que l'expression est un symbole défini par **#define**.

```
#ifdef  SYMBOLE      /* vrai si SYMBOLE est defini */
#ifndef SYMBOLE      /* vrai si SYMBOLE non defini */
```

Ces directives s'associent bien entendu aux directives **#else** et **#endif** de la même manière qu'avec le **#if**.

10.4. Autres directives

Sur certains compilateurs C et principalement sur ceux qui tournent sur des petits systèmes du type CP/M, existent les directives **#asm** et **#endasm**. La directive **#asm** indique que les lignes qui suivent sont des instructions en langage d'assemblage. La directive **#endasm** signale la fin de la séquence assembleur et la poursuite du programme en C.

```
#asm
    ...
    instructions en langage d'assemblage
    ...
#endasm
```


V

LES AUTRES OBJETS MANIPULÉS PAR C

Nous avons déjà vu que le langage C possède trois types de base : **char**, **int** et **float**.

char : représente un octet de huit bits.

int : représente un entier dont la taille est celle d'un mot machine.

float : représente un nombre *flottant* en simple précision, lui aussi dépendant de l'architecture machine (format de l'exposant et de la mantisse).

Les attributs **short**, **long**, **unsigned** et **double** sont venus s'ajouter aux types fondamentaux pour s'adapter aux différentes tailles et architectures de mots machine.

A partir de ces types de base et de leurs attributs le langage C permet de construire une infinité de type d'objets dérivés. Parmi eux, les tableaux, les pointeurs, les structures, les unions, les énumérations et les fonctions, sont connus du langage.

1. LES TABLEAUX

1.1. Déclaration d'un tableau

Un tableau est une structure de données de dimension finie dont les éléments consécutifs qui le constituent sont tous du même type. Dans un tableau on n'a pas le droit de mélanger des entiers, des caractères ou des réels. Par contre on peut avoir un tableau d'entiers, un tableau de caractères ou un tableau de réels.

Un tableau en C est caractérisé par quatre éléments:

- son nom,
- son type,
- sa dimension,
- sa classe d'allocation.

Ces éléments doivent figurer dans sa définition. Rappelons que l'identificateur de classe est implicite selon le contexte.

```
int tab[100];          /* classe externe ou automatique */
static char text[32];  /* classe statique */
```

Le nom du tableau est un identificateur qui permet de le localiser en mémoire.

Le type du tableau représente le type des éléments qui le constituent et non un type tableau. En C, il n'existe pas de type tableau comme on le rencontre en PASCAL avec "*array of*".

La dimension d'un tableau est une constante entière qui donne le nombre d'éléments. La dimension est mise entre crochets:

```
int tab[10];      /* tableau de 10 entiers */
```

Dans cet exemple, le compilateur alloue en mémoire dix entiers consécutifs. Un tableau peut ne pas être dimensionné dans une déclaration et dans ce cas, cette déclaration n'alloue aucun espace mémoire, mais permet de faire référence à ce tableau défini ailleurs avec sa dimension.

Exemple:

```
char tb[80];           /* definition d'un tableau */
main()
{
    ...
    f(tb);             /* appel de la fonction f */
    ...
}
f(t)                   /* fonction f */
char t[];              /* t est du type tableau */
{
    extern int tab[]; /* tab est un tableau externe */
    ...
}
```

L'opérateur [ ], au niveau de la déclaration signifie "*tableau de*"; ne pas le confondre avec l'opérateur d'indiçage ou d'indexation qui, lui, est utilisé pour accéder à un élément du tableau **tab**[i]. La taille physique en octets occupée par un tableau peut être obtenue à l'aide de l'opérateur **sizeof**.

Les classes d'allocation possibles sont les classes externes, statiques et automatiques. La classe registre n'est pas autorisée car la mémorisation d'un tableau dans un registre n'a pas de sens.

Un tableau, pris comme un tout, n'est pas considéré comme une variable. Un tableau est une succession de variables d'un type donné, dont chacune d'elles est destinée à recevoir une valeur. Si **t** est un tableau:

t[0] est le premier élément du tableau **t**
t[i] est le i+1 *ième* élément du tableau **t**
t[i] est une variable.

Par contre **t** n'est pas une variable mais une constante qui identifie le nom du tableau. Cet identificateur **t** permet en fait d'adresser le tableau par son nom. L'adresse **t** du tableau **t** peut également s'écrire, si on utilise l'opérateur unaire &:

&t[0] l'adresse du premier élément du tableau

Par contre, le nom d'un tableau étant une constante, les expressions suivantes sont invalides:

```
t = expression ;      /* faux */
t++;                  /* faux */
adr_t = &t ;          /* faux */
```

1.2. Initialisation d'un tableau

Un tableau peut être initialisé au moment de la compilation uniquement si la classe d'allocation est externe ou statique.

Le cas le plus simple est celui d'une chaîne de caractères qui est considérée comme un tableau de caractères (*array of char*), terminée par le caractère nul \0.

Exemple:

```
static char text[18] = "ceci est un texte" ;
```

Le tableau **text** est dimensionné à 18 caractères compte tenu du caractère nul de fin de chaîne. Pour éviter au programmeur de compter les caractères du texte, le compilateur calcule automatiquement la longueur de la chaîne de caractères. Il n'est donc pas nécessaire de spécifier explicitement la dimension du tableau, et on peut écrire:

```
static char merr[] = "message d'erreur" ;
```

Dans le cas général, les éléments du tableau doivent être spécifiés un à un sous la forme d'une liste. La liste des valeurs est délimitée par des accolades et chaque élément est séparé par une virgule ",". On aurait pu par exemple écrire :

```
static char mess[] = {'m','e','s','s','a','g','e','\0'};

static int chiffres[10] = { 0,1,2,3,4,5,6,7,8,9 } ;
```

Dans le cas où la dimension est explicitement spécifiée, celle-ci doit être supérieure ou égale au nombre d'éléments dans la liste d'initialisation, sinon une erreur est diagnostiquée. Soit le tableau suivant incomplètement initialisé :

```
short pairs[5] = { 2,4,6 };
```

Les deux derniers éléments non spécifiés dans la liste seront initialisés à zéro.

```
pairs[0]=2 , pairs[1]=4 , pairs[2]=6 , pairs[3]=pairs[4]=0
```

1.3. Tableaux multidimensionnels

L'opérateur [ ], signifiant *"tableau de"* au niveau d'une déclaration, peut être juxtaposé à un autre [ ]. Il est donc possible de définir ainsi des tableaux de tableaux.

Exemple :

```
int t2d[10][20] ;
```

où **t2d** est un tableau de 10 tableaux de 20 entiers chacuns.

En fait, un tableau de tableaux est un tableau bidimensionnel ou matrice possédant un tableau de lignes ; chaque ligne se comportant comme un tableau de colonnes.

```
int matrice[Y][X] ;      /* Y = nombre de lignes */
                         /* X = nombre de colonnes */
```

L'accès à un élément de la matrice s'obtient à l'aide d'indices de type entier, qui varient entre 0 (indice de début) et la dimension moins un (indice de fin).

```
cellule = matrice[y][x] ;
matrice[i][j] = 0 ;
```

L'application de la récursivité permet ainsi de définir des tableaux à n dimensions, dans la limite où l'espace disponible en mémoire le permet. Un tableau à n dimensions se déclare de la manière suivante :

```
tnd [D1][D2].....[Dn] ;
```


Soit le tableau à trois dimensions :

```
float t3d [10][20][30] ;
```

Les expressions **t3d**, **t3d[x]**, **t3d[x][y]** sont du type tableau ; l'expression **t3d[x][y][z]** est du type **float**.

L'initialisation d'un tableau multidimensionnel répond à la même syntaxe qu'un tableau monodimensionnel.

Exemples :

```
long t2d [5][4] = {
                    { 0,1,2,3 }    ,  /* t2d[0][] */
                    { 4,5,6,7 }    ,  /* t2d[1][] */
                    { 8,9,10,11 }     /* t2d[2][] */
                  } ;
```

avec :

```
t2d[0][0]=0 , t2d[0][1]=1 , t2d[0][2]=2 , t2d[0][3]=3
t2d[1][0]=4 , etc...
t2d[3][] et t2d[4][] sont bien entendu initialisés à zéro.
```

```
int t3d[2][2][2] = {
                     {          /* t3d[0][][] */
                       { 1,2 }
                       { 3,4 }
                     },
                     {          /* t3d[1][][] */
                       { 5,6 } ,
                       { 7,8 }
                     }
                   } ;
```

Les accolades peuvent clarifier ou embrouiller selon les cas ou selon le programmeur. L'initialisation du tableau précédent peut également s'écrire :

```
int t3d[2][2][2] = { 1,2,3,4,5,6,7,8 } ;
```

En effet, le compilateur procède de la manière suivante : en partant de l'accolade ouvrante, il garnit les cases du tableau le plus imbriqué en faisant progresser un indice. Dès que cet indice correspond à la dimension du tableau il le réinitialise à zéro et fait progresser l'indice du tableau de rang précédent et ainsi de suite. Cet algorithme de remplissage se perçoit mieux avec l'utilisation des boucles **for** suivantes :

```
int x,y,z,i=1;
for(x=0 ; x<2 ; ++x)
        for(y=0 ; y<2 ; ++y)
                for(z=0 ; z<2 ; ++z)
                        t3d[x][y][z] = i++ ;
```

Ne pas confondre:

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

qui initialise la première colonne à 1 et la seconde à 0

1	0
1	0
1	0

avec:

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

1	1
1	0
0	0

Pour illustrer l'utilisation des tableaux à plusieurs dimensions nous allons prendre l'exemple du carré magique. Un carré magique d'ordre n est un tableau à deux dimensions dont les cases qui sont remplies par les n*n premiers nombre entiers sont organisées de façon que les sommes des éléments de chaque ligne, de chaque colonne ou de chaque diagonale soient égales.

Il existe un algorithme simple de constitution de carré magique pour un ordre impair. Les nombres entiers progressent de 1 jusqu'à n*n. Le premier nombre doit se situer dans la case au-dessus de la case centrale, puis la progression normale consiste à suivre la diagonale vers la direction Nord-Est; soit i— et j++ si i et j sont respectivement les indices de ligne et de colonne. Dès qu'il y a un dépassement des limites $1 < 0$, $j < 0$, alors on repart à la ligne du bas avec $i = n - 1$ ou à la dernière colonne avec $j = n - 1$; de même si $j = n$ on repart à la première colonne avec $j = 0$. Si on atteint une case déjà occupée par un nombre précédemment rangé, on change de direction (Nord-Ouest) i — et j— pour reprendre à nouveau une diagonale Nord-Est.

Il est bien évident qu'on doit chaque fois vérifier les cas de débordement du carré. Celui-ci, peut d'ailleurs s'imaginer comme un cylindre pour chaque dimension où la ligne du bas est située au-dessus de la ligne du haut et la dernière colonne est située avant la première colonne.


```
/****** programme carre magique d'ordre impair ******/

#define MAX 13
int carre[MAX][MAX];      /* tableau externe */
main()
{
        int i; /* indice ligne */
        int j; /* indice colonne */
        int k; /* nombre entier */
        int n; /* ordre du carre */

        printf ("donner l'ordre du carre magique :");
        scanf  ("%d",&n);
        if ( n<=2 || n>MAX || n%2==0 )
                printf ("l'ordre du carre est incorrect\n");
        else {
                razcarre(n);              /* remise a zero du carre */
                j = n/2; i = j-1;         /* coordonnees initiales */
                k = 1;                    /* valeur initiale */
                carre[i][j] = k;
                i-- ; j++ ; k++ ;
                while ( k <= n*n ) {      /* boucle centrale */
                      if (i < 0) i = n-1;
                      if (j < 0) j = n-1;
                      if (j >=n) j = 0;
                      if (carre[i][j] == 0) { /* test si case vide */
                            carre[i][j] = k;  /* oui on range k */
                            i-- ; j++ ; k++;  /* direction N.E. */
                      }
                      else {
                            i-- ;   j-- ;     /* direction N.O. */
                      }
                }
                printcarre(n);
        }
}
razcarre(n)
int n;
{       register i,j;
        for (i=0 ; i <= n-1 ; i++)
                for (j=0 ; j <= n-1 ; j++)  carre[i][j] = 0;
}
printcarre(n)
int n;
{       register i,j,som;
        for (i=0 ; i <= n-1 ; i++) {
                for (som=0 , j=0 ; j <= n-1 ; j++)
                        printf(" %4d",carre[i][j]);
                        som += carre[i][j];
        }
        printf("  %4d\n",som);        /* impression somme des lignes */
        printf ("\n");                /* saut d'une ligne */
        for (j=0 ; j <= n-1 ; j++) {
               for (som=0 , i=0 ; i <= n-1 ; i++)  som += carre[i][j];
               printf(" %4d",som); /* impression somme des colonnes */
        }
}
```

2. LES POINTEURS

2.1. Déclaration d'un pointeur

Un pointeur est une variable qui contient l'adresse d'une autre variable. Nous avons déjà vu que l'opérateur unaire & permet d'obtenir l'adresse d'une variable. Si **v** est une variable:

```
&v   est son adresse
ptr_v = &v ;     /* ptr_v = pointeur de v */
```

L'opérateur & peut s'appliquer à n'importe quelle variable.

L'affectation d'un pointeur répond à la syntaxe suivante:

pointeur = &*variable*;

ou

pointeur = & *tableau [indice de l'élément]*;

Connaissant l'adresse d'une variable, il est possible d'obtenir son contenu à l'aide de l'opérateur unaire d'indirection *.

```
*&ptr_v = v ;    /* car *&v == v */
```

Les objets pouvant avoir des types différents, il est nécessaire de déclarer le pointeur avec le type de l'objet qu'il pointe. Il suffit pour cela de définir le type de la variable pointée pour définir un pointeur. En C, un pointeur se déclare de la façon suivante:

type_de_l'objet_pointé *pointeur;

Exemple:

```
int *ptri ;     /* ptri est un pointeur d'entier */

char *ptrc;     /* ptrc est un pointeur de caracteres */
```

s'écrit aussi:

```
char ptrc[];    /* ptrc est une chaine de caracteres */
```

L'opérateur * ne peut s'appliquer qu'aux expressions de type pointeur.

2.2. Opérations sur les pointeurs

Lorsqu'on déclare un pointeur on ne sait pas encore sur quel objet il pointe, on ne connaît que son type. Un pointeur est une variable qui s'initialise comme les variables de type de base. L'affectation de l'adresse de la variable au pointeur

peut se réaliser, soit lors de sa définition (initialisation par le compilateur), soit lors de l'exécution par une instruction d'affectation. Ceci implique qu'un pointeur peut faire partie d'une expression, soit à gauche, soit à droite du signe = d'affectation.

Exemple:

```
int *p;
int *q = p;   /* le pointeur q est egal au pointeur p */

int *p,*q;
    q = p;
```

cette instruction d'affectation copie le contenu de **p** dans **q**; **q** pointe maintenant sur l'objet que pointe **p**. Ces deux écritures donnent le même résultat. Ne pas confondre **q = p;** avec l'affectation suivante:

```
*q = *p;
```

où l'objet pointé par **p** est transféré à l'adresse pointée par **q**.

En C:

```
int i, *p, t[100] ;
```

déclare un entier **i**, un pointeur **p** d'entier et un tableau **t** de 100 entiers. On peut alors écrire les instructions suivantes:

```
 p = &i;    /* p pointe sur l'entier i */
*p = 0 ;    /* la valeur 0 est affectee a l'entier i */
```

La remise à zéro du tableau **t**:

```
for (i=0 ; i<100 ; i++)  t[i]=0 ;
```

s'écrit en utilisant le pointeur **p**:

```
for(p=&t[0] ; p < &t[100] ; p++) *p=0 ;
```

ou encore:

```
for (p=t ; p < t+100 ; p++) *p=0;
```

En fait, le nom d'un tableau est considéré comme un pointeur vers le premier élément du tableau. Le compilateur C transforme automatiquement la référence à un tableau en une expression de type pointeur. De même l'élément de tableau **t[i]** est lui aussi converti immédiatement en une expression de la forme ***(t+i)**.

```
t    <==>   &t[0]
t[i] <==>   *(t+i)
```

Il faut bien différencier la notion de pointeur de celle d'entier. Un pointeur correspond à l'adresse machine d'un objet et non à un entier au sens nombre signé ou non même si le codage interne du pointeur et de l'entier est identique. C'est le cas des machines à mots de 16 bits pour lesquelles adresse et entier sont codés sur un mot machine. Sur d'autres ordinateurs à base de mots de 32 bits, la partie adresse ne s'identifie pas nécessairement au mot complet: par exemple les adresses sur 24 bits et les entiers sur 32 bits.

Exemple:

```
int i , *p ;
        p = &i;      /* est correct */
        p = i;       /* est incorrect */
```

De même une constante ne peut être affectée à un pointeur, sauf dans le cas de la constante nulle 0:

```
p = 0;       /* pointeur vide */
```

signifie que le pointeur **p** ne pointe sur rien (notation NIL en PASCAL ou en LISP).

Par contre l'addition ou la soustraction d'une variable entière ou d'une constante entière sont des opérations absolument valides. Dans l'exemple ci-dessus de remise à zéro d'un tableau, si les entiers sont représentés sur des mots de 16 bits:

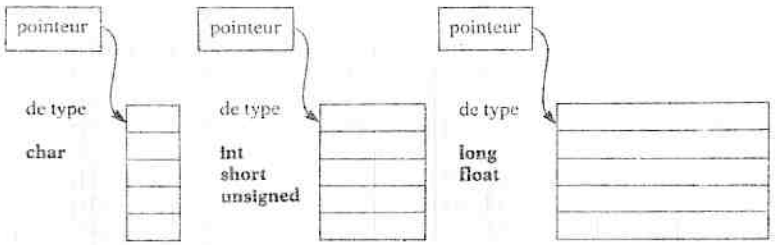
```
i++  soit i=i+1    ajoute 1 à i
p++  soit p=p+1    ajoute 2 à p
```

car **p** est une adresse octet, **p++** pointe l'entier suivant. Si **i** avait été de type **double**, **p++** aurait ajouté 8 à **p**. De même:

```
p -= 2;      /* est correct */
p += (i+5); /* est correct */
```

La valeur additionnelle est donc convertie immédiatement en un multiple de la taille de l'objet pointé.

Progression des pointeurs selon les types d'objet:



Un pointeur n'étant pas un entier, mise à part l'addition et la soustraction d'un entier, toutes les autres opérations arithmétiques et logiques:

```
*  /  %  &  |  ^  ~
```

lui sont interdites.

De même l'addition de deux pointeurs n'est pas autorisée. Par contre la soustraction de deux pointeurs est permise, et donne un résultat de type entier, c'est-à-dire la valeur de l'intervalle en octets.

Exemple:

```
int *p, *q;
      p + q    est incorrect
      p - q    est correct
```

Les pointeurs considérés comme des adresses peuvent être comparés. Les opérateurs relationnels de comparaison == != < <= > >= sont donc utilisables.

Exemple: fonction qui calcule la longueur d'une chaîne de caractères.

```
strlen(s)      /* version : 1 */
register char s[];
{       register int n;
        for (n=0 ; s[n] != '\0' ; n++) ;
        return (n);
}
```

s'écrit avec un pointeur:

```
strlen(s)      /* version : 2 */
register char *s;
{       register n;
        for (n=0 ; *s != '\0' ; s++) n++;
        return (n);
}
```

ou:

```
strlen(s)      /* version : 3 */
register char *s;
{       register n=0;
        while (*s++) n++;
        return (n);
}
```

s'écrit avec deux pointeurs:

```
strlen(s)      /* version : 4 */
register char *s;
{       register char *p = s;
        while (*p) p++;
        return(p-s); /* pointeur fin - pointeur debut */
}
```

Dans les exemples précédents, l'utilisation de l'indirection sur un pointeur est plus efficace que l'indiçage d'un tableau, lequel nécessite une multiplication et une addition avant d'effectuer le test.

La fonction qui copie une chaîne de caractères source **s** vers une chaine de caractères destinatrice **d**, s'écrit:

```
copystr(s,d)
register char *s, *d;
{
        while ((*d++ = *s++) != '\0');
}
```

et n'engendre que cinq octets de code sur un ordinateur VAX.

Le principal grief que l'on pourrait imputer aux pointeurs est que leur utilisation, si elle est massive, risque de rendre difficile la lecture du programme, voire illisible. De plus, des erreurs de manipulation de pointeurs peuvent provoquer des écrasements de zones mémoire, en particulier par les opérations d'indirection.

Pour illustrer l'utilisation des pointeurs nous allons reprendre l'exemple du programme "accolade". Dans sa version initiale, "accolade" ne se contentait que de comptabiliser les accolades ouvrantes et fermantes d'un programme source écrit en C.

S'il y a un déséquilibre entre les accolades ouvrantes et fermantes, ce programme ne nous permet pas de dire où s'est produit l'oubli. C'est pourquoi, nous proposons ci-dessous une version de ce programme qui visualise toutes les lignes où apparaissent les accolades. D'autre part les lignes sont numérotées pour faciliter la tâche de l'utilisateur, s'il doit appeler l'éditeur de texte pour effectuer sa correction.


```
/* nouvelle version du programme verification des accolades */

#include <stdio.h>

main()
{
    register char c;
    register brace;
    register n;
    int begin, end;             /* compteurs d'accolades */
    char *pl;                   /* pointeur de ligne source */
    char ligne[256];            /* tampon ligne source */

    for (n=0, begin=0, end=0, pl=ligne ; (c=getchar()) != EOF ; ) {
        switch (c) {
            case '{' : *pl++ = c; ++begin; brace = 1; break;
            case '}' : *pl++ = c; ++end  ; brace = 1; break;
            case '\n': *pl++ = c; n++;
                                if (brace) {
                                    brace = 0;
                                    *pl = '\0';
                                    printf("%4d:%s", n, ligne);
                                }
                       pl = ligne;  break;
            default  : *pl++ = c ;  break;
        }
    }
    printf("nombre de { = %d\n",begin);
    printf("nombre de } = %d\n",end);
    if (begin-end)
        printf("erreur de {}\n");
}
```

Ce qui donne à l'exécution, en lançant la commande "accolade" à partir du **Shell**:

```
            $cat accolade.c | accolade

  4:{
 12:  for (n=0, begin=0, end=0, pl= ligne ; (c=getchar()) != EOF ; )
 13:      switch (c) {
 14:          case '{':*pl++ = c; ++begin; brace = 1; break;
 15:          case '}':*pl++ = c; ++end  ; brace = 1; break;
 17:                               if (brace) {
 21:                               }
 24:      }
 25:  }
 26:  printf("nombre de { = %d\n",begin);
 27:  printf("nombre de } = %d\n",end);
 29:      printf("erreur de {}\n");
 30:}
nombre de { = 7
nombre de } = 7
```

2.3. Tableaux de pointeurs

Nous avons déjà vu que l'opérateur [ ] signifiait *"tableau de"*. En fait, du point de vue physique, il s'agit d'un tableau monodimensionnel ou vecteur. La description d'un tableau physique s'obtient alors en juxtaposant les deux opérateurs *"tableau de"*, soit **t2d[L][C]**, où **L** et **C** représentent les dimensions respectives des lignes et des colonnes du tableau bidimensionnel. Au sens C, cette écriture signifie que l'on dispose d'un tableau de lignes et que chaque ligne est elle-même un tableau de variables dont le type est spécifié devant le nom du tableau. Dans ce cas de figure, **L * C** objets de ce type sont alloués en mémoire. On est donc en présence d'un tableau rectangulaire classique où toutes les lignes ont la même dimension.

Il est cependant fréquent de trouver des structures où la dimension de chaque ligne peut être quelconque. C'est par exemple le cas des lignes d'un texte source tapé au clavier du terminal ou stocké dans un fichier. Les lignes sont alors des chaînes de caractères terminées par le caractère séparateur de fin de ligne (newline). La dimension de chacune des lignes est alors variable et inconnue à l'avance. Le langage C permet pourtant de représenter cette structure de fichier source par la déclaration suivante:

```
char *ptligne[NLIGNES]; /* NLIGNES = nombre de lignes du fichier */
```

Cette déclaration crée un tableau de NLIGNES pointeurs de caractère (type **char ***). Pour exploiter le contenu du fichier source, il sera donc nécessaire de procéder à une phase d'initialisation des pointeurs, par exemple par une lecture du fichier ligne par ligne, afin que chacun d'eux pointe sur le début d'une ligne.

L'allocation supplémentaire du tableau de pointeurs et son initialisation peuvent paraître comme des inconvénients, mais l'accès par indirection et la dimension variable sont des avantages indéniables qui permettent de traiter avec souplesse ce type de structure de données. On évite ainsi de manipuler des tableaux fixes garnis de cellules vides.

La phase d'initialisation d'un tableau de pointeurs, si elle est fastidieuse au niveau de l'exécution dynamique, peut également se réaliser de manière statique au niveau de la compilation.

Exemple:

```
static char *jour[] = {
                      "Lundi",
                      "Mardi",
                      "Mercredi",
                      "Jeudi",
                      "Vendredi",
                      "Samedi",
                      "Dimanche"
                      } ;
```


où **jour** est un tableau de pointeurs sur les noms des jours de la semaine. Chaque nom de jour est une chaîne de caractères terminée par le caractère nul '\0'. Le tableau lui-même est terminé par un pointeur nul.

```
 jour[0] pointe sur "Lundi"
*jour[0] = L
*(jour[0]+1) = u
 jour[1] pointe sur "Mardi"
```

2.4. Arguments des commandes Shell

Sous le système d'exploitation UNIX, une commande utilisateur se présente sous la forme d'une suite de chaînes de caractères séparées par des espaces et dont la première est par définition le nom de la commande.

```
$commande argument_1 argument_2 ... argument_n
```

Le programme **Shell** qui réalise l'interface entre l'utilisateur et le système fournit au programme utilisateur, celui qui correspond à la commande, un pointeur de nom **argv** qui permet de retrouver les arguments placés dans la commande.

Argv est un pointeur sur une table de pointeurs dont chacun pointe sur le début d'un argument de la commande. Chaque argument est mémorisé sous la forme d'une chaîne de caractères au sens C, c'est-à-dire terminée par un octet nul \0. La fin de la liste des arguments est elle-même représentée dans la table des pointeurs par un pointeur vide ou pointeur nul de valeur 0.

C'est le programme principal **main**, auquel le contrôle est donné en premier, qui reçoit les arguments du **Shell.**

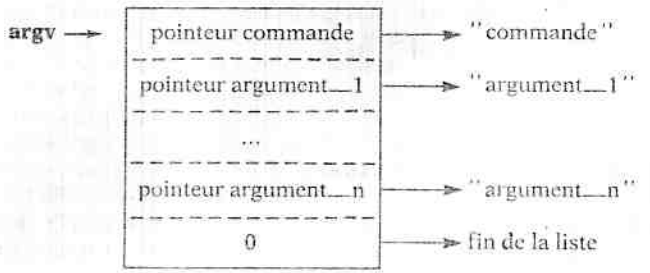
Si le programmeur désire exploiter les informations en provenance du **Shell**, il doit alors déclarer son programme principal de la façon suivante:

```
main (argc,argv)
int argc;
char *argv[];
{
        ...
}
```

argc	compte d'arguments, nom de commande inclus
argv	pointe sur la table des pointeurs
argv[0]	pointe sur le nom de la commande
***argv[0]**	où **argv[0][0]** est le premier caractère de la commande
++argv	est équivalent à **argv[1]** et pointe sur le premier argument de la commande (celui qui suit le nom de la commande).

La figure ci-dessous illustre comment on atteint un argument à partir du paramètre **argv** transmis par le **Shell** :

Table des pointeurs des arguments



Pour illustrer le passage des arguments du **Shell** vers le programme, nous allons nous baser sur un type de syntaxe courant sous UNIX. En effet, la plupart des commandes UNIX admettent comme argument un ou plusieurs noms de fichier agrémentés d'options spécifiées par une lettre précédée du caractère tiret —.

Exemple :

```
commande -a -b fichier
```

Dans notre exemple, la syntaxe de la commande se présente sous la forme suivante :

```
commande -option chaine
```

où **option** est soit : **a** (ajouter), **c** (changer), **d** (détruire), **l** (lister).


```
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    if (argc < 2)  usage();
    --argc; ++argv;
    if (**argv != '-')  usage();
    switch (*++*argv) {          /* (*(++*argv)) */
        case 'a' : ajouter(argc,argv);    break;
        case 'c' : changer(argc,argv);    break;
        case 'd' : detruire(argc,argv);   break;
        case 'l' : lister(argc,argv);     break;
        default  : usage();
    }
}
ajouter(argc,argv)
int argc; char *argv[];
{
    if (!--argc)  usage();
    ++argv;
    ...
}
...
usage()
{
    printf("usage :\n");
    printf("\t commande -a chaine  ==> [ajouter]\n");
    printf("       ...    -c   ...       [changer]\n");
    ...
    exit(1);       /* abandon du programme */
}
```

Une autre écriture consisterait à écrire :

```
{
    if (--argc < 1 || *argv[1] != '-')  usage();
    switch (*++argv[1]) {
    ...
    }
}
```

3. LES STRUCTURES

Une structure est un agrégat de plusieurs objets de différents types pouvant eux-mêmes être déjà structurés (tableau, autre structure, etc).

Dans les autres langages dits structurés, l'équivalent de la structure en C est l'enregistrement ou "record" en PASCAL, PL/1 et COBOL.

Une structure est considérée comme une entité qui peut être traitée comme *un tout* et dont les éléments qui la constituent ont une relation étroite avec cette entité. Pour être traitée comme une unité, une structure réunit l'ensemble des objets qui la composent sous un même vocable, le nom de la structure.

Par exemple l'identité d'un individu peut regrouper sous le nom identité des informations aussi disparates que son nom, son adresse, son numéro de sécurité sociale, sa date de naissance, etc.

Le nom de l'individu est par exemple une chaîne de caractères, son adresse est constituée de plusieurs chaînes de caractères séparées par des espaces avec un code postal, son numéro de sécurité sociale est une suite de treize chiffres, et sa date de naissance un ensemble de nombres identifiant le jour, le mois et l'année.

Dans le cas de l'adresse et de la date on voit qu'on est en présence d'informations complexes qui peuvent à leur tour être décomposées et donc être organisées en structures.

3.1. Description d'une structure

La description d'une structure introduit le mot réservé **struct**. Ce mot clé doit être suivi du nom de la structure et de tous les éléments, appelés membres, qui la composent. Ceux-ci sont délimités par l'accolade ouvrante { au début et l'accolade fermante } à la fin de la description de la structure

Comme nous l'avons déjà mentionné, une structure est une collection d'objets de différents types. Il est donc nécessaire de déclarer le type de chaque variable qui la composent. La déclaration d'un membre d'une structure doit être suivie du caractère délimiteur ";" comme pour une déclaration normale.

La syntaxe générale d'une déclaration de structure peut avoir l'allure suivante:

```
struct nom_structure {
        type1 variable1;
        ... ...
        typen variablen;
};
```

où **nom_structure** est le nom donné à ce type de structure ainsi définie. Ce nom n'est pas une variable mais une référence qui détermine le type particulier de cette structure.

Exemple:

```
struct record {
        long cle;
        int  taille;
        char texte[128];
} ;
```

où **record** est un nouveau type structuré qui décrit un enregistrement constitué d'une clé d'accès, de la taille de l'enregistrement et de la partie utile du texte.

3.2. Déclaration d'une variable de type structure

Il faut bien différencier le sens donné à une déclaration de structure et à une déclaration d'une variable de type structure. Une déclaration de structure décrit la structure et les membres qui la composent et lui donne un nom de modèle (type) qui permettra de la référencer. La déclaration d'un nouveau type structuré ne définit en fait que le format de cette structure pour une certaine catégorie d'objets, et ne provoque aucune allocation en mémoire.

Au contraire, la déclaration d'une variable de type structure permet d'associer une variable à la description de la structure et de l'allouer en mémoire. La variable ainsi déclarée est la structure elle-même.

Deux écritures sont possibles pour déclarer une variable de type structure:

soit:

```
struct nom1 {
        ...
        liste des déclarations des membres
        ...
} nom2;
```

soit:

```
struct nom1 {
        ...
        liste des déclarations des membres
        ...
};
```

avec:

```
struct nom1 nom2;
```

Dans les deux cas, **nom1** déclare une structure ayant une certaine forme et **nom2** est une structure de type **nom1**. A la différence de **nom1** qui est considéré comme un type, **nom2** est une variable au même titre que:

```
int nom2;
```

où **int** jouerait le rôle (sur le plan syntaxique) de **struct nom1**.

Exemple:

```
struct record {
        long cle ;
        int taille ;
        char texte[128];
} article ;
```

est une déclaration de structure où **article** est une structure de type **record**.

Au même titre que les autres variables déjà étudiées, une variable de type structure peut prendre différents aspects. La déclaration suivante:

```
struct {
        ...
        liste des membres
        ...
} s , *ps , ts[50] ;
```

déclare:

une structure **s** de ce type
un pointeur **ps** vers cette structure
un tableau **ts** de 50 structures de ce type

Dans ce cas de figure on remarque que le mot clé **struct** n'est pas suivi d'un identificateur mais de la description de la structure. Celle-ci n'a pas effectivement besoin de nom de modèle car les variables **s**, **ps** et **ts** sont directement associées à ce modèle.

3.3. Membres d'une structure

Les membres internes à une structure peuvent être de types quelconques, variables de types de base, pointeurs, mais aussi des tableaux ou encore des sous-structures possédant leurs propres membres. Une structure peut donc comporter plusieurs niveaux hiérarchiques.

Exemple:

```
struct identite {
        char nom[16];
        char prenom[20];
        struct adr adresse;
        char No_secu[13];
        struct date naissance;
} individu ;
```


avec:

```
struct adr {
        int numero;
        char rue[40];
        long code_postal;
        char localite[16];
} ;
```

et:

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

Il est bien évident que la récursivité ne peut s'appliquer au niveau des définitions de structures. Ce serait le cas où une structure contiendrait un membre étant lui-même la structure initiale. Cependant un membre d'une structure peut référencer la structure mère à l'aide d'un pointeur vers cette structure.

Exemple:

```
struct genealogie {
        char nom_prenom [30];
        struct date naissance;
        struct genealogie *pere;
        struct genealogie *frere;
} homme ;
```

où **pere** et **frere** sont des pointeurs vers une structure de type **genealogie**.

3.4. Opération sur les structures

Les opérateurs classiques, utilisés pour les variables simples, ne s'appliquent pas aux structures. Seul l'opérateur & permet d'obtenir l'adresse de tout objet en C et à fortiori d'une structure. Une des principales utilisations de l'opérateur adresse se manifeste lors de la lecture d'enregistrements structurés dans les fichiers, opérations que nous aborderons lors de l'étude des fonctions d'entrées-sorties offertes par la librairie standard.

L'affectation de structure n'était pas autorisée lors de la description du langage par Kernighan et Ritchie en 1978. Depuis, des améliorations ont été portées à C, et certains compilateurs supportent l'affectation de structure. Les structures peuvent ainsi être manipulées comme un tout.

Exemple: si

```
struct nom_st st1 st2;      /* st1 et st2 structures */
```

on peut écrire:

```
st1 = st2;
```

qui copie la structure **st2** dans la structure **st1**.

Certains compilateurs récents permettent même que les structures soient passées sous forme d'arguments aux fonctions et soient également retournées par ces fonctions. Par contre il n'est pas encore possible de comparer deux structures comme en tout.

```
if (st1 != st2)      /* test illegal */
```

3.5. Accès aux membres d'une structure

Deux opérateurs spécifiques sont fournis pour accéder aux membres des structures. Il s'agit des opérateurs "." et "->".

L'opérateur "." relie le nom d'une variable de type structure à un de ses membres.

nom_structure.membre

Dans le cas où plusieurs structures sont imbriquées, cette opération se réitère car une structure interne à une autre structure est considérée comme un de ses membres.

Exemple:

```
if (1983 - individu.naissance.annee >= 18 )
        printf("%s %s est majeur\n",individu.nom
                                    ,individu.prenom);
```

L'opérateur "->", quant à lui, s'applique aux pointeurs de structure. Cet opérateur relie le pointeur de la structure au membre que l'on désire atteindre.

ptr_struct -> membre

Exemple:

```
struct adr ad,*ptad;
```

ad.numero	est équivalent à	ptad->numero
ad.rue[0]	...	ptad->rue[0]
(*ptad).code_postal	...	ptad->code_postal

Dans le dernier exemple, les parenthèses sont nécessaires car la priorité de l'opérateur . est plus élevée que celle de l'opérateur *.

3.6. Initialisation des structures

Dans la mesure où la classe d'allocation d'une structure est statique ou externe celle-ci peut être totalement ou partiellement initialisée comme pour les autres objets déjà étudiés.

Exemple:

```
struct date {
        int jour;
        int mois;
        int annee;
} date_annee = { 1, 1, 1983 } ;

struct date date_jour = {24, 2, 1983};  /* 24/02/1983 */
```

3.7. Champs de bits

Le langage C offre la possibilité de manipuler les informations binaires à l'aide des opérateurs de traitement de bits mais aussi avec les structures. Il est en effet possible de définir, dans le membre d'une structure, des champs de bits de longueur variable. La déclaration d'un champ de bits consiste en un identificateur facultatif suivi du caractère ":" et d'une constante entière donnant la taille du champ. La taille en bits d'un champ ne doit pas excéder la taille physique d'un mot machine, c'est-à-dire d'un entier. S'il y a débordement le champ suivant est aligné sur l'entier suivant. L'identification d'un champ de bits par un nom n'est pas obligatoire, ce qui permet de considérer des zones inutiles de remplissage ("padding" en anglais). La taille 0 permet également de se recaler automatiquement sur le début de l'entier suivant.

Exemple:

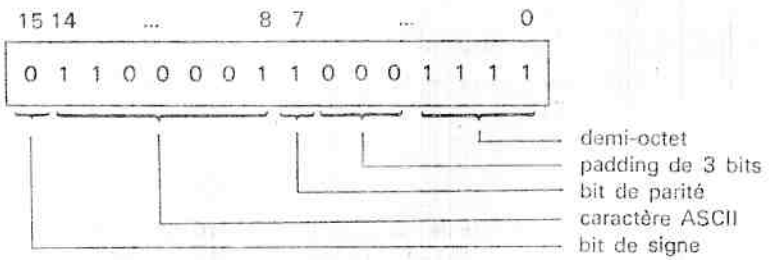
```
struct {
        unsigned bit_signe  : 1;    /* bit 15 de signe    */
        unsigned car_ascii  : 7;    /* bits 8 a 14        */
        unsigned bit_parite : 1;    /* bit 7 de parite    */
        unsigned : 3;               /* 3 bits de padding */
        unsigned demi_octet : 4;    /* bits 0 a 3         */
} champ = { 0 , 'a' , 1 , 0x0f };  /* initialisation     */

main()
{
        printf("champ = %x\n",champ);
        printf("bit_signe  = %x\n",champ.bit_signe);
        printf("car_ascii  = %x\n",champ.car_ascii);
        printf("bit_parite = %x\n",champ.bit_parite);
        printf("demi_octet = %x\n",champ.demi_octet);
}
```

donne à l'exécution:

```
champ = 618f
bit_signe  = 0
car_ascii  = 61
bit_parite = 1
demi_octet = f
```

La représentation interne du champ de bits en mémoire est la suivante:



L'implémentation des champs de bits varie selon les compilateurs, dans lesquels l'affectation des bits peut se faire soit à partir des poids faibles de la droite vers la gauche, soit inversement à partir des poids forts de la gauche vers la droite, (cas de l'exemple ci-dessus).

Il est bien évident que l'on ne peut adresser physiquement les bits, c'est pourquoi l'opérateur & ne s'applique pas aux champs.

Les principaux avantages des champs de bits sont de pouvoir compresser des données en mémoire (table des symboles avec des indicateurs dans un compilateur), ou de pouvoir manipuler des champs spécifiques (mots d'états d'un contrôleur pour un "handler" de périphérique).

4. LES UNIONS

Une union est une structure de données qui permet de manipuler alternativement des objets de différents types localisés dans une même zone de la mémoire, sans pour cela se préoccuper des implantations physiques en machine.

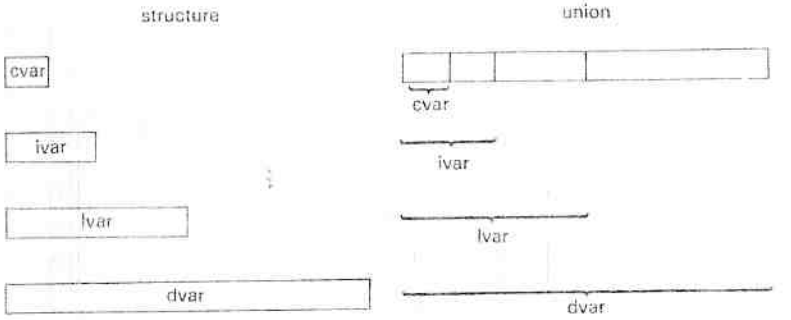
Sur le plan syntaxique, la déclaration d'une union est parfaitement identique à celle d'une structure.

Dans la déclaration:


```
union union_def {
        char cvar;
        int ivar;
        long lvar;
        double dvar;
} union_var ;
```

union_def définit une union composée de différents objets localisés à la même adresse, celle du début de l'union; **union_var** représente une variable de type **union_def**.

Le schéma ci-dessous, nous montre les différences d'implantations en mémoire entre une structure et une union.



Une union peut être considérée comme une structure dans laquelle tous les éléments sont alignés sur la même adresse. On retrouve, en particulier ce type de structure dans le langage PASCAL avec les enregistrements variables (record + case of). L'analogie est plus forte lorsque l'union est elle-même imbriquée dans une structure.

Exemple:

```
struct enregistrement {
        long cle; int taille;
        union {
                char text[80];
                float reel[20];
        } donnees ;
} article[100] , *ptr_record ;
```

Dans cet exemple, **article** est un tableau de 100 structures de type **enregistrement** dont le membre **donnees** est une union qui peut contenir soit des caractères soit des nombres réels, et **ptr_record** est un pointeur sur ce type de structure.

C'est au programmeur qu'est laissée l'initiative de sélectionner soit le tableau de caractères soit le tableau de réels, selon le contexte de son traitement.

L'accès aux membres des unions se réalise à l'aide des mêmes opérateurs qui manipulent les membres des structures. Ainsi :

```
article[i].donnees.text[j]
```

accède au **j** ième caractère du tableau **text** dans le **i** ième **article** de la structure **enregistrement**.

de même : ptr_record->donnees.reel[k]

accède au **k** ième nombre flottant du tableau **reel**.

5. LES ÉNUMÉRATIONS

Dans la version originale du langage C décrite par Kernighan et Ritchie en 1978, les types scalaires ne figuraient pas. De récentes améliorations ont été portées à C pour pallier cette lacune par rapport à PASCAL. Bien que cette nouvelle caractéristique ne soit pas implémentée sur tous les compilateurs existants, il est utile d'en faire une description succincte.

Le type énumération de C est pratiquement analogue au type scalaire du langage PASCAL, il permet de définir une liste de constantes symboliques qui appartiendront à ce type. Sa syntaxe peut s'illustrer comme suit :

enum *identificateur* { *liste de constantes symboliques* } ;

Exemple :

```
enum couleurs { blanc,jaune,vert,bleu,rouge } couleur;
```

où **couleurs** est le nom du type de l'énumération et **couleur** une variable de type **couleurs** pouvant prendre une des cinq valeurs de la liste.

Les éléments de la liste d'énumération sont considérés comme des constantes entières. Par défaut, leurs valeurs sont numérotées à partir de zéro avec un pas de progression de 1. L'exemple ci-dessus nous donne :

blanc = 0 ,jaune = 1 ,vert = 2 ,bleu = 3 ,rouge = 4

et l'instruction :

couleur = bleu ; est équivalente à couleur = 3 ;

de même :

couleur = 4 ; est équivalente à couleur = rouge ;

Si un élément de la liste est initialisé explicitement par le signe d'affectation =, les valeurs des éléments suivants progressent à partir de cette valeur.

Exemple :

```
enum mois { Janvier=1, Fevrier, Mars, Avril,
            Mai, Juin, Juillet, Aout,
            Septembre, Octobre, Novembre, Decembre };
```

donne :

```
Fevrier=2, ... , Decembre=12
```

La déclaration :

```
enum ordinateurs { VAX,ONYX,ALTOS,S8000,MICROMEGA };
```

avec :

```
enum ordinateurs machine,*modele ;
```

déclare la variable **machine** qui est du type **ordinateurs** et la variable **modele** qui est un pointeur sur un objet de type **ordinateurs**.

Les valeurs des éléments d'une énumération doivent être distinctes.

Exemple, la déclaration suivante est illégale :

```
enum direction { SUD,NORD,EST,OUEST,BAS=0,HAUT };
```

car :

```
SUD=0 BAS=0 et NORD=1 HAUT=1
```

6. LES FONCTIONS

Une fonction est par définition un objet externe, au même titre que les données externes extérieures aux fonctions. Cependant, la syntaxe de définition d'une fonction diffère de celle des données.

La définition d'une fonction répond à la syntaxe suivante :

fonction(liste d'arguments)

Même si la liste est vide, la présence des parenthèses est obligatoire et permet ainsi d'identifier facilement une fonction ou un appel de fonction. La différence, sur le plan syntaxique, entre une définition et un appel de fonction est l'absence ou la présence du caractère délimiteur ";".

Exemple :

```
f(arg1,arg2)    /* est une definition de la fonction f */

f(arg1,arg2);   /* est une instruction qui
                   appelle la fonction f */
```

Par contre dans une expression, une forme du type **f(arg1,arg2)** sans caractère délimiteur provoque un appel à la fonction **f**.

```
while (f(arg1,arg2) > 0)
```

6.1. Passage des arguments

Dans un appel de fonction, la liste des arguments, si elle existe, est composée soit de variables soit de constantes. Dans la définition de la fonction, l'image de cette liste doit également apparaître. Cette image est une liste de paramètres formels dont le nombre doit être égal au nombre d'arguments réels lors de l'appel à la fonction. Les paramètres de la liste doivent correspondre un à un aux arguments de la liste d'appel, ainsi que leur type.

Le passage des arguments réels vers la fonction se fait par valeur. Ce qui signifie que ce sont les contenus des variables qui sont transmis à la fonction et non leur adresse. Chaque contenu est recopié dans une variable propre à la fonction qui est identifiée par le nom du paramètre formel. Avant d'effectuer la recopie, la variable privée est préalablement allouée dans la pile de travail et disparaît à la fin de l'exécution de la fonction. La recopie des arguments réels dans des variables privées évite d'altérer les variables originales situées dans la fonction appelante. Le passage par valeur présente donc une sécurité non négligeable car la fonction appelée est considérée comme totalement autonome et indépendante.

Exemple:

```
char c;
int  i;
        ...
        f(c,i);      /* appel de la fonction f */
        ...
}
f(y,x)               /* definition de la fonction f */
int  x;              /* x = copie de la variable i  */
char y;              /* y = copie de la variable c  */
{
        ...
}
```

Du point de vue pratique, le passage par valeur n'est pas applicable aux tableaux en raison du temps prohibitif qu'entraînerait la copie du tableau tout entier. Le mode normalement utilisé est alors le passage par référence qui consiste à faire passer la valeur de l'adresse du tableau. De même pour un pointeur, c'est la valeur du pointeur qui est transmise et est ensuite recopiée dans la variable locale. L'adresse du tableau et le pointeur appartenant à la fonction appelante ne sont donc pas affectés. Les éléments du tableau ou l'objet pointé sont par contre directement accessibles au niveau de la fonction appelée; indicage pour le tableau et indirection pour le pointeur. Le programmeur doit cependant rester vigilant, car les variables qu'il manipule sont extérieures à la fonction courante.

Afin de permettre la recopie des arguments transmis par la fonction appelante, il est nécessaire de déclarer les paramètres formels immédiatement après la définition de la fonction et juste avant le corps de la fonction identifié par la première accolade ouvrante.

La forme générale d'une fonction a l'allure suivante:

```
fonction (liste des paramètres)
déclaration des paramètres
{
        déclaration des variables locales
        séquences d'instructions
}
```

Ce sont les déclarations qui suivent la définition de la fonction qui déterminent le type et le nom des variables privées dans lesquelles vont s'effectuer les copies des arguments. Les classes autorisées pour les variables privées sont la classe automatique et la classe registre. Les classes statiques et externes n'ont pas de sens car les variables copiées ont une durée de vie égale à celle du temps d'exécution de la fonction.

Si certains paramètres ne sont pas utilisés dans le corps de la fonction, il n'est pas nécessaire de les déclarer, même s'ils sont présents dans la liste de définition de la fonction.

Exemple:

```
main (argc,argv)
char **argv;          /* argc non utilise */
{
```

Les paramètres formels de la liste ne doivent pas être associés aux opérateurs d'adressages tels que & (adresse) ou * (indirection).

Les différentes déclarations possibles du programme principal **main** sont:

```
main (argc,argv,envp)  /* arguments transmis par le Shell */
int argc;              /* compte d'arguments */
char *argv[];          /* pointeur des pointeurs d'arguments */
char **envp;           /* pointeur de l'environnement */

main (argc,argv)
int argc;
char *argv[];

main (argc)
int argc;

main ()
```

Autres exemples de passages d'arguments à des fonctions :

```
fonct (c,i,s,l,u,f,d)  /* passage d'objets de base */
char c; int i;
short s; long l; unsigned u;
float f; double d;
{
        ...
}

fonct(pc,pi)           /* passage de pointeurs */
char *pc; int *pi;
{
        ...
}

        ...
        fonct (&var,tab,&table[0]);
        ...
}
fonct (pvar,t,tbl)     /* passage d'adresse et
                          de noms de tableaux */
long *pvar;      /* pvar est un pointeur sur var */
char t[];        /* t est un tableau de caracteres */
int *tbl;        /* tbl est un tableau d'entiers */
{
        ...
}

        ...
        fonct (*ptrvar,**ptrptr);
        ...
}
fonct (ptrv,arg)       /* passage d'objets pointés */
int ptrv;
char arg;
{
        ...
}

fonct (stru, pstru)    /* passage de structure */
struct type_struc stru;
register struct type_struc *pstru;
{
        ..
}
```


6.2. Types de fonctions

Au même titre que les autres objets manipulés par C, une fonction possède un type qui est celui de la valeur qu'elle retourne.

L'instruction **return**, si elle est présente, peut/ne pas préciser une valeur explicite. C'est le cas du **return** seul qui renvoie alors une valeur entière indéfinie. L'absence d'instruction **return** dans le corps de la fonction se ramène au cas précédent car le compilateur en rajoute un systématiquement. Pour ces raisons, dues à des simplifications d'écriture du retour de fonction, le type par défaut d'une fonction est le type **int**. Dans la pratique il n'est pas nécessaire de spécifier le type d'une fonction lors de sa déclaration dans la mesure où celle-ci retourne un **int** ou un **char**.

Les deux définitions de fonctions qui suivent sont équivalentes :

```
int fonc()
fonc()
```

Le type **int**, par défaut, s'applique également aux fonctions retournant une valeur de type **char**. C'est par exemple le cas de la fonction abondamment utilisée **getchar( )**. En effet, on peut écrire :

```
int i;
char c;
        i = getchar();
        c = getchar();
```

Dans les autres cas, lorsque la fonction ne retourne ni un **int** ni un **char**, il est nécessaire de spécifier le type de la valeur retournée dans la déclaration de la fonction. C'est par exemple le cas des fonctions de conversion disponibles dans la librairie standard qui convertissent un nombre codé en ASCII en un nombre codé en représentation interne :

```
/* declarations preliminaires pour utiliser
   les fonctions de conversion de la librairie */

long atol();      /* conversion ASCII => entier long */
double atof();    /* conversion ASCII => flottant long */

char *s;          /* s = pointeur d'un nombre ASCII */
     ...
long l;double f; /* variables de l'utilisateur */
     ...
     l = atol(s);
     f = atof(s);
     ...
```

Autre exemple:

```
char *move();     /* declaration preliminaire */

main()
{
    char *source,*dest,*p; /* declaration des pointeurs */
    ...
    p = move(source,dest);
    ...
}

char *move(from,to) /* definition de la fonction */
char *from,*to;     /* declaration des parametres */
{
    int lg;         /* longueur de la chaine */
    ...
    return(to+lg);  /* pointe apres to */
}
```

Une fonction C ne peut retourner qu'une valeur. C'est pour cette raison qu'il n'est pas concevable de retourner un tableau, une structure, une union ou une fonction. Par contre, il est possible de retourner le pointeur d'un tableau, d'une structure, d'une union et d'une fonction. Dans ce cas le nom de la fonction doit être précédé du type de l'objet suivi de l'opérateur unaire *, spécifiant que la fonction retourne un pointeur. Le type représente alors le type de l'objet pointé.

Exemples:

```
f()         /* f retourne un entier */
int f()     /* f retourne un entier */
int *f()    /* f retourne un pointeur sur un entier */
char *f()   /* f retourne un pointeur sur un caractere */
```

Le type d'une fonction précise donc le type de la valeur retournée ou de la variable pointée dans le cas d'un pointeur. Par contre le type d'une fonction ignore le nombre et le type de ses arguments. Ceux-ci doivent coïncider entre la définition de la fonction et son appel.

Par défaut la classe des fonctions est la classe externe; il est ainsi possible de définir une fonction locale au fichier source dans lequel elle se trouve au moyen du spécificateur de classe **static**. Il est bien évident qu'il est hors de question d'attribuer la classe automatique ou la classe registre à une fonction. Une fonction représentant des séquences d'instructions, l'initialisation d'une fonction n'a pas de sens.

7. COMPOSITION DE TYPES

La combinaison de plusieurs types est possible pour définir le moyen d'accès à la variable que l'on désire atteindre.

Exemples de constructions valides:

```
int i;              /* entier i */

char *ptrc;         /* ptrc pointe sur un caractere */

long *flon()        /* la fonction flon retourne un pointeur
                       sur un entier long */

int (*pfon)()=fon; /* pfon est un pointeur sur la fonction fon
                      qui retourne un entier */

union **fct()       /* fct est une fonction qui retourne un
                       pointeur de pointeur d'union */

short (*fsh())[]   /* fsh est une fonction qui retourne un
                      pointeur sur un tableau d'entiers courts */

struct *(*fst())()      /* fst est une fonction qui retourne un
                           pointeur vers une fonction retournant
                           un pointeur sur une structure */

int (*(*(*F())[])())[]  /* F est une fonction qui retourne un
                           pointeur sur un tableau de pointeurs
                           lequel pointe sur des fonctions
                           retournant des pointeurs de tableaux
                           constitués d'entiers ...Ouf!     */
```

8. DÉFINITIONS DE TYPE SYNONYMES

Le langage C permet de définir de nouveaux noms de type grâce au mot réservé **typedef**. Le mot clé **typedef** a le même effet que la directive **#define**, excepté qu'il est interprété directement par le compilateur et non par le préprocesseur C. **Typedef** définit en effet un nouveau nom de type mais ne crée pas un nouveau type. Son utilisation permet souvent une meilleure lisibilité du programme. La définition d'un nom de type répond à la syntaxe suivante:

typedef *type nom_nouveau_type*;

Exemples :

```
typedef int compteur;      /* compteur est un type entier */

typedef char octet;        /* octet est un type caractere */

typedef char *texte;       /* texte est un type pointeur
                              de caractere                 */
```

Les nouveaux noms **compteur**, **octet** et **texte**, sont respectivement synonymes de **int char** et **char** *. Ils s'utilisent alors de la manière suivante :

```
compteur cpt;              /* synonyme de : int  cpt;      */
octet  byte ;              /* synonyme de : char byte;     */
texte string;              /* synonyme de : char *string;  */
```

Typedef s'applique également aux types d'objets structurés tels que les tableaux, les structures, les unions et les énumérations.

Exemple :

```
typedef struct {
    double reel;
    double imaginaire;
} complexe ;   /* complexe est le type de cette structure */
```

avec :

```
complexe z;                /* z est une variable complexe */
```

cette formulation améliore sensiblement la lisibilité par rapport à la suivante :

```
struct complexe { double reel, imaginaire } ;
```

avec :

```
struct complexe z;         /* z est une variable complexe */
```

Autre exemple :

```
typedef enum { actif, bloque, candidat, dormant } etats ;
```

avec :

```
etats etat;                /* etat = variable de type etats */
```

L'emploi de **typedef** est fortement conseillé pour paramétriser un programme vis-à-vis des problèmes de portabilité. Les tailles des objets de bases (**int**, **short**, **long**, **float**, **double**) sont souvent différentes entre machines (*machine dependent*) ou entre compilateurs (*system dependent*).

VI

L'ENVIRONNEMENT DE PROGRAMMATION C

Un environnement de programmation consiste en un ensemble d'outils logiciels mis à la disposition des utilisateurs. Dans un environnement C, ces outils se présentent généralement sous deux aspects :

— les librairies de fonctions,
— les commandes.

Les librairies de fonctions constituent l'interface de programmation, alors que les commandes constituent l'interface de dialogue entre l'usager et le système à travers un terminal conversationnel.

1. LES LIBRAIRIES

Une librairie de fonctions se présente sous la forme d'un fichier dans lequel ont été rassemblées des fonctions précompilées, préalablement écrites en C, en assembleur, ou en un tout autre langage.

C'est l'éditeur de liens qui extrait de la librairie les fonctions appelées par le programme utilisateur, les rajoute à la fin de celui-ci, puis satisfait les références externes en établissant des connexions d'adresse entre les appels de fonctions et les fonctions appelées.

Parmi les principales librairies attachées à l'environnement C, on trouve :

— la librairie standard,
— la librairie mathématique,
— la librairie système (C run time library).

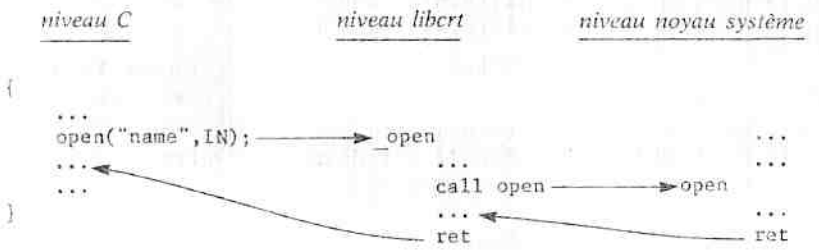
La librairie standard de nom **libc** (sous UNIX) complète avantageusement les insuffisances du langage C sur le plan des entrées-sorties et de la manipulation

des chaînes de caractères. Cette librairie est entièrement portable car les fonctions qui la composent sont toutes écrites en C et compilées par le compilateur C de l'installation.

La librairie mathématique de nom **libm** (sous UNIX) regroupe les fonctions mathématiques comme son nom l'indique. Cette librairie est généralement dépendante de l'environnement matériel. En effet, la plupart des ordinateurs possèdent des processeurs spécialisés pour le calcul rapide en virgule flottante, ou pour la résolution d'opérations complexes tels que les fonctions trigonométriques, les logarithmes, etc. La librairie mathématique est dite *"machine dependent"*.

La librairie système **crt** (C Run Time Library) réalise l'interface de programmation avec les primitives du système d'exploitation (UNIX, CP/M, VMS, etc.). A chaque point d'entrée dans le système correspond une routine écrite dans le langage d'assemblage de la machine.

Le plus souvent, pour des raisons de commodité, les modules de cette librairie sont fusionnés avec la librairie standard.



2. LES COMMANDES

A la différence des fonctions de librairie auxquelles on n'accède qu'au niveau de l'exécution des programmes, les commandes sont tapées au clavier et constituent le seul moyen de dialogue entre l'utilisateur et le système d'exploitation.

Le nom d'une commande référence un programme exécutable stocké dans un fichier. Sous UNIX et la plupart des systèmes d'exploitation, les commandes exécutables peuvent se présenter sous deux formes:

- fichier binaire directement exécutable par le processeur,
- fichier source de commandes cataloguées exécutable par un produit logiciel appelé un interpréteur de commandes (**Shell** pour UNIX, **Submit** pour CP/M).

Une commande peut, soit appartenir à l'utilisateur, soit être accessible à tous les utilisateurs. Par exemple, sous UNIX, les commandes publiques appartiennent au propriétaire système **bin**. Les fichiers exécutables associés aux commandes sont localisés dans deux catalogues (directories) spécifiques, décrits par leur chemin d'accès dans l'arborescence des fichiers:

/bin
/usr/bin

Le contrôle d'accès aux fichiers sous UNIX permet de spécifier, pour un utilisateur, un groupe d'utilisateurs ou tous les autres utilisateurs, les autorisations ou des protections en lecture, écriture ou exécution. Ainsi, dans le cas d'UNIX, les commandes publiques bénéficient de l'autorisation d'exécution. La configuration des droits d'accès pour une commande publique est la suivante:

—rwxr—x—x

où r, w, x, représentent respectivement les autorisations en lecture, écriture et exécution pour le propriétaire, le groupe et tout les autre utilisateurs.

La notion de commande publique n'est pas spécifique à UNIX et nous la retrouvons sous diverses formes dans la plupart des systèmes d'exploitation. Dans le cas d'un système mono-utilisateur du type CP/M où la notion de propriétaire n'existe pas, les commandes accessibles correspondent aux fichiers de type COM (CP/M-80) ou de type CMD (CP/M-86) présents dans le directory de la disquette ou du disque courant.

Quel que soit le système d'exploitation un environnement de programmation C nécessite au minimum les outils suivants appelables sous forme de commandes:

- un éditeur de texte pour construire, saisir et modifier les programmes source écrits en C: commande **ed**;
- un compilateur C et sa librairie standard associée: commande **cc**;
- un éditeur de liens: commande **ld**.

D'autres commandes, moins vitales mais néanmoins d'une grande utilité sont proposées dans le package UNIX:

- le vérificateur sémantique de programmes C: commande **lint**,
- l'outil de maintenance de programmes C: commande **make**,
- le reformateur de programmes C: commande **cb**.

3. LA LIBRAIRIE STANDARD LIBC

La librairie standard d'un environnement C regroupe généralement trois catégories de sous-programmes:

— les fonctions d'entrées-sorties standard,
— les fonctions utilitaires,
— les fonctions système.

Les fonctions d'entrées-sorties standard et les fonctions utilitaires sont écrites en C, compilables par le compilateur de l'installation et donc totalement transportables d'un système à l'autre. Elles fournissent divers services, inexistants dans C lui-même, dont les principaux sont :

pour les fonctions d'entrées-sorties standard :

— l'accès aux fichiers,
— les entrées-sorties caractère par caractère,
— les entrées-sorties ligne par ligne,
— les entrées-sorties formatées.

pour les fonctions utilitaires :

— la manipulation des chaînes,
— les conversions des nombres et chaînes numériques,
— la gestion mémoire,
— autres utilitaires divers.

Les fonctions système sont dépendantes du système d'exploitation. Ces fonctions sont en fait des routines écrites en assembleur (celui de la machine). Vues d'un programme C, elles se comportent comme des fonctions standard. Elles permettent au programmeur d'accéder aux services intimes offerts par le système, c'est-à-dire aux primitives. Parmi les plus importantes citons :

— la gestion des processus,
— les entrées-sorties directes,
— les séquences d'exception.

La liste des primitives du système UNIX est fournie en annexe.

3.1. Le fichier stdio.h

Tous les programmes C qui appellent les fonctions de la librairie standard doivent se conformer à un certain nombre de conventions. La plupart d'entre elles sont exprimées, en ce qui concerne les entrées-sorties, dans le fichier en-tête **stdio.h**. Ce fichier doit être inclus au début de tout programme C qui sollicite des fonctions de la librairie d'entrées-sorties par l'insertion de la directive :

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

Ce fichier ne comporte aucune séquence de code, mais des définitions de constantes et des déclarations d'objets et de fonctions nécessaires à l'accès des fonctions d'entrées-sorties de la librairie standard.

Exemple de fichier **stdio.h** :

```
#define BUFSIZ 512     /* taille d'un tampon d'E/S */
#define EOF    (-1)    /* code de fin de fichier   */
#define NULL   0       /* pointeur nul */
#define NFILE  15      /* nombre de fichiers ouverts par processus */

extern struct _iobuf {
    int    _cnt;       /* nombre d'octets traités dans le tampon */
    char  *_ptr;       /* pointeur sur l'octet courant */
    char  *_base;      /* adresse du debut du tampon */
    char   _flag;      /* mode d'ouverture du fichier (0, 1, 2) */
    char   _file;      /* descripteur systeme du fichier (numero) */
} FILE[NFILE];         /* tableau de NFILE structures */

#define stdin  (&FILE[0])  /* pointeur du fichier standard d'entree */
#define stdout (&FILE[1])  /* pointeur du fichier standard de sortie */
#define stderr (&FILE[2])  /* pointeur du fichier standard d'erreur */

#define getc(p)     (--(p)->_cnt>=0? *(p)->_ptr++&0xff: _fillbuf(p))
#define getchar()   getc(stdin)
#define putc(x,p)   (--(p)->_cnt>=0? ((*p)->_ptr++=x): _flushbuf(x,p)
#define putchar(x)  putc(x,stdout)

FILE *fopen();         /* fonction d'ouverture d'un fichier */
FILE *fdopen();
FILE *freopen();
long ftell();          /* fonction de positionnement dans un fichier */
char *fgets();         /* fonction de lecture d'une chaine d'octets */
```

3.2. Accès aux fichiers

Un fichier peut être identifié de deux manières :

— par son nom externe,
— par son nom interne.

Le nom externe du fichier est le nom qui lui a été donné au moment de sa création par son propriétaire. Ce nom, constitué d'une chaîne de caractères, référence le fichier de manière permanente.

Le nom interne, au sens de la librairie standard, est un pointeur sur une structure qui contient les informations propres au fichier (index de l'octet dans le tampon, pointeur de l'octet courant, adresse de début du tampon, mode d'ouverture, numéro du descripteur de fichier au niveau système). La durée de vie de ce nom interne est bien entendu liée à celle de l'exécution du programme qui le manipule.

Sa déclaration s'écrit :

```
FILE *pfile;     /* pfile est un pointeur de fichier */
```

La valeur de ce pointeur est retournée par l'appel de la fonction d'ouverture du fichier :

fopen("fichier",mode)

où "**fichier**" représente le nom externe tel qu'il existe dans le catalogue, et où **mode** détermine le mode d'ouverture du fichier :

"r"	ouverture en lecture	(consultation)
"w"	ouverture en écriture	(création ou écrasement)
"a"	ouveture en ajout	(rallongement)

Exemple :

```
pfile = fopen("fichier", "r");     /* ouverture en lecture */
```

La fonction **fopen( )** retourne :

- soit le pointeur associé au fichier s'il existe,
- soit le pointeur nul 0 si le fichier n'existe pas.

A partir de ce moment, dans la suite du programme, toutes les opérations sur le fichier seront effectuées avec son pointeur.

Seuls les trois fichiers standards **stdin**, **stdout** et **stderr** n'ont pas besoin d'une ouverture préalable. Ces trois fichiers sont pré-ouverts par le système dès le début de l'exécution de tout programme.

Le fichier standard d'entrée est normalement le clavier du terminal de l'utilisateur, tandis que les fichiers standard de sortie et d'erreur sont l'écran ou l'imprimante du terminal. Les pointeurs **stdin**, **stdout**, **stderr** associés à ces fichiers sont prédéfinis dans le fichier **stdio.h**.

Pour illustrer l'utilisation de la fonction **fopen( )** prenons l'exemple d'un programme qui réalise un traitement successivement sur un ou plusieurs fichiers. La commande de l'utilisateur se présentant sous la forme :

prog [fich1] [fich2] ... [fichn]

où **fich1**, **fich2**, ..., **fichn** sont des noms de fichiers facultatifs. Si aucun nom n'est présent, le traitement s'effectuera sur le fichier standard d'entrée **stdin**, c'est-à-dire le clavier du terminal par défaut. La commande sera alors réduite à sa plus simple expression :

prog

Si l'exécution de ce programme a lieu sous le système UNIX ou un dérivé "UNIX-like", l'entrée standard peut être produite par un autre fichier grâce à la convention < :

prog < fichin

De même, la sortie du résultat peut être redirigée dans un autre fichier grâce au signe > :

prog < fichin > fichout

Ces opérations de redirection sont bien entendu ignorées du programme utilisateur. Les chaînes <**fichin** et >**fichout** n'étant pas transmises au programme par l'intermédiaire de la variable **argv**.

Exemple :

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    FILE *pf;          /* declaration du pointeur de fichier */
    if (argc <= 1)
        traitement(stdin);
    else {
        while (--argc > 0) {
            if ((pf = fopen(*++argv,"r")) == NULL) {
                fprintf(stderr,"fichier: %s non trouve\n", *argv);
                exit(1);     /* abandon du programme */
            }
            traitement(pf);
            fclose(pf);
        }
    }
    exit(0);                 /* fin de traitement */
}
traitement(ptrf)
FILE *ptrf;
{
    ...
}
```

Cet exemple illustre l'utilisation de **fopen( )**, **stdin**, **stderr** et NULL définis dans **stdio.h**, mais aussi l'utilisation de trois nouvelles fonctions : **fprintf**, **fclose** et **exit**.

La fonction **fprintf** agit comme un **printf** sur un fichier repéré par son pointeur. Celui-ci doit être le premier argument de la liste (**stderr** dans l'exemple). La syntaxe générale de **fprintf** répond à :

fprintf (pointeur, "format", variables)

La fonction **fclose** permet de fermer le fichier repéré par son pointeur.

fclose (pointeur)

La fonction **exit** est une fonction système qui termine le processus en cours et redonne le contrôle à l'interpréteur de commande. Cette routine se charge de fermer automatiquement tous les fichiers restés ouverts dans le programme. D'où la redondance de **fclose** dans l'exemple, mais néanmoins utile pour la portabilité et la logique de programmation.

3.3. Les entrées-sorties

La librairie standard offre toute une panoplie de fonctions permettant d'effectuer des entrées-sorties quel que soit le système d'exploitation. On peut classifier les entrées-sorties en trois groupes:

- les lectures,
- les écritures,
- les positionnements.

Les lectures et écritures de données peuvent elles-mêmes être subdivisées en plusieurs sous-classes:

- les entrées-sorties caractère par caractère,
- les entrées-sorties mot par mot,
- les entrées-sorties de chaîne de caractères,
- les entrées-sorties binaires bufferisées,
- les entrées-sorties formatées.

3.3.1. Entrées-sorties caractère par caractère

La librairie standard offre deux fonctions en lecture et deux fonctions en écriture selon que le fichier d'entrée ou de sortie est standard ou non.

Pour un fichier quelconque la fonction de lecture d'un caractère est:

```
getc(pf)          /* lecture d'un octet */
FILE *pf;
```

où **pf** est le pointeur du fichier concerné. La fonction **getc** retourne l'octet courant dans le fichier et se positionne sur l'octet suivant. Si la fin du fichier est rencontrée, c'est la valeur EOF (– 1) qui est retournée.

Pour l'entrée standard on utilise de préférence la fonction:

```
getchar()
```

Celle-ci est définie dans le fichier **stdio.h** par la directive de macro-substitution:

```
#define  getchar()  getc(stdin)
```


En ce qui concerne l'écriture d'un caractère, la librairie standard fournit les deux fonctions **putc** et **putchar** symétriques de **getc** et **getchar**:

```
putc(c,pf)          /* ecriture d'un caractere */
char c;
FILE *pf;

putchar(c)       /* putchar(c) <==> putc(c,stdout) */
```

Le caractère de la variable **c** est écrit dans le fichier correspondant.

On peut illustrer l'utilisation de la fonction **getc** en complétant l'exemple précédent qui appelait la fonction **traitement(pf)**, non encore détaillée. Dans l'exemple ci-dessous, cette fonction détecte les caractères non imprimables et les visualise sous forme hexadécimale précédée du signe \. La fin de ligne est également signalée par le caractère \.

```
traitement(ptrf)
FILE *ptrf;
{
    int c, posit=0;
    while ((c = getc(ptrf)) != EOF) {
        if (c == '\n' || c == '\t' || c >= ' ' && c <= 0x7e) {
            putchar(c);
            posit++;            /* position dans la ligne */
        }
        else {
            printf("\\%02x", c);
            posit += 3;
        }
        if (c == '\n')
            posit = 0;
        if (posit > 72) {     /* 72 colonnes maximum */
            printf("\\\n");
            posit = 0;
        }
    }
}
```

3.3.2. Entrées-sorties mot par mot

Les deux fonctions **getw** et **putw** (getword et putword) sont équivalentes aux fonctions **getc** et **putc** à la seule différence que l'unité de transfert n'est plus l'octet mais le mot de type int.

```
getw(pf)          /* lecture d'un mot */
FILE *pf;

putw(w,pf)        /* ecriture d'un mot */
int w;
FILE *pf;
```

La valeur EOF est retournée lorsqu'une fin de fichier est atteinte. La valeur – 1 de EOF est ambiguë car les données peuvent avoir cette valeur. Dans ce cas, il est conseillé d'utiliser la fonction **feof** pour vérifier que l'on a ou non atteint la fin du fichier.

Exemple :

```
int tab[100];
FILE *pfich;
int i;          /* index du tableau */

...
while (!feof(pfich)) [
    for (i = 0 ; i < 100 ; i++) {
        if (feof(pfich))
            break;
        tab[i] = getw(pfich);
        ...
    }
}
printf ("fin du fichier atteinte\n");
```

3.3.3. Entrées-sorties de chaîne de caractères

Il est souvent utile de saisir ou d'imprimer une suite de caractères en une seule opération. La librairie standard offre deux fonctions en lecture et deux fonctions en écriture, selon que le fichier est standard ou non.

Les données se présentent sous deux formes différentes selon qu'elles se trouvent dans un fichier ou en mémoire. Dans un fichier, elles sont considérées comme des lignes de texte séparées par le caractère de fin de ligne \ n. En mémoire, elles deviennent des chaînes de caractères terminées par le caractère nul \ 0.

La fonction de lecture d'une ligne texte dans un fichier quelconque est :

```
char *fgets(abuf, n, pf)
char *abuf;         /* adresse du buffer ligne */
int  n;             /* taille n + 1 en octets */
FILE *pf;           /* pointeur du fichier */
```

La fonction **fgets** lit **n–1** octets maximum, sinon arrête sa lecture dès la rencontre d'une fin de ligne \ n (newline). Le caractère nul \ 0 est placé automatiquement à la fin du buffer mémoire, ce qui permet de considérer l'ensemble des informations de ce buffer comme une chaîne de caractères au sens C. La valeur retournée est le premier argument de la fonction c'est-à-dire le pointeur sur le buffer où est constituée la chaîne de caractères. Dans le cas d'une fin de fichier, c'est le pointeur vide NULL qui est renvoyé.

La fonction **gets** est pratiquement identique à **fgets**, dans la mesure où le fichier d'entrée est le fichier standard **stdin**. La taille maximale n'est pas précisée en argument et le caractère de fin de ligne \ n est remplacé par le caractère de fin de chaîne \ 0 dans le buffer mémoire.

```
char *gets(abuf)        /* retourne l'adresse abuf */
char *abuf;
```

Pour les sorties, on dispose des deux fonctions **fputs** et **puts**, parallèles aux fonctions d'entrée **fgets** et **gets**. Aucune des deux fonctions ne copie sur le fichier de sortie de caractère nul \ 0 de fin de chaîne.

Par raison de compatibilité avec **fgets** et **gets**, **fputs** ne rajoute pas le caractère de fin de ligne \ n tandis que **puts** le rajoute.

```
fputs(abuf, pf)  /* ecriture d'une ligne texte */
char *abuf;
FILE *pf;

puts(abuf)       /* uniquement sur stdout */
char *abuf;
```

3.3.4. Entrées-sorties binaires bufferisées

Les fonctions **fread** et **fwrite** proposées par la librairie standard permettent de lire ou d'écrire n'importe quel type de données. Elles permettent en particulier de lire ou d'écrire des tableaux ou des structures.

Syntaxes de définition de **fread** et **fwrite** :

```
fread(abuf, sizeof(*abuf), nbr_elem, pf)
char *abuf;
int  nbr_elem;
FILE *pf;

fwrite(abuf, sizeof(*abuf), nbr_elem, pf)
char *abuf;
int  nbr_elem;
FILE *pf;
```

La fonction **fread** lit dans une zone mémoire, débutant à l'adresse **abuf**, **nbr_elem** éléments de données dont le type est ***abuf**, à partir du flot de données pointé par **pf**. La valeur retournée est soit le nombre d'éléments effectivement lus, soit 0 si la fin du fichier est atteinte.

La fonction **fwrite** rajoute les **nbr_elem** éléments de la zone mémoire pointée par **abuf** dans le fichier repéré par **pf**.

3.3.5. Entrées-sorties formatées

Nous avons déjà entrevu la syntaxe et l'utilisation des fonctions **printf** et **scanf** dans le premier chapitre. D'autres fonctions dérivées de **printf** et **scanf** offrent des services nouveaux:

```
fprintf(pf, format [,variable] ... )
FILE *pf;
char *format;

sprintf(pm, format [,variable] ... )
char *pm;
char *format;
```

Dans les deux cas précédents, comme pour la fonction **printf**, **format** représente une chaîne de caractères composée de texte et/ou de spécificateurs de conversion. Le nombre et la position des spécificateurs de format annoncés par les caractères %, doivent coïncider avec le nombre et la position des variables placées en arguments.

Les deux écritures suivantes sont équivalentes:

```
printf("valeur = %d\n",var);
```

et:

```
char mesval[] = "valeur = %d\n";
...
printf(mesval, var);
```

La fonction **fprintf** se distingue de **printf** par le fait qu'on spécifie un fichier de sortie référencé par son pointeur **pf** alors que **printf** n'utilise que le fichier de sortie standard **stdout**.

Exemple:

```
fprintf(stderr, "le fichier %s n'existe pas\n",*argv);
```

La fonction **sprintf** diffère de **fprintf** par le fait que le fichier se substitue à une zone mémoire pointée par **pm**.

Exemple:

```
float ht;          /* prix hors taxe */
float TVA = 1.186; /* coefficient TVA */
char prix[16];
...
sprintf(prix, "%f %f\0", ht, ht * TVA);
```


Les mêmes considérations que celles portées sur **fprintf** et **sprintf** sont applicables aux fonctions de lectures formatées **fscanf** et **sscanf**. Se reporter au premier chapitre pour l'étude de **scanf**.

```
scanf(format [,pointeur] ... )
char *format;

fscanf(pf, format, [,pointeur] ... )
FILE *pf;
char *format;

sscanf(pm, format, [,pointeur] ... )
char *pm;
char *format;
```

Les fonctions **scanf**, **fscanf** et **sscanf** retournent le nombre d'éléments lus, c'est-à-dire effectivement affectés à des variables. La valeur EOF est retournée lorsque la fin du fichier a été atteinte (pour **scanf** et **fscanf**).

3.3.6. Les fonctions de positionnement

La librairie standard offre quatre fonctions de positionnement de fichier: **fseek**, **ftell**, **rewind**, **ungetc**.

La fonction **fseek** permet de réaliser un accès direct au niveau de l'octet dans un fichier. Elle initialise le positionnement de la prochaine lecture ou écriture. La nouvelle position est donnée par le déplacement en octet par rapport:

```
au début du fichier      : mode = 0
à la position courante   : mode = 1
à la fin du fichier      : mode = 2
```

L'argument qui spécifie le déplacement doit être de type **long** pour pouvoir adresser des fichiers de plus de deux milliards d'octets.

```
fseek(pf, deplacement, mode)
FILE *pf;
long deplacement;
int mode;              /* 0 | 1 | 2 */
```

La fonction **rewind** qui positionne le fichier à son début est équivalente à la construction:

```
fseek(pf, 0L, 0);    /* deplacement = 0 long */

rewind(pf)
```

La fonction **ftell** retourne la valeur (**long**) de la position courante par rapport au début du fichier. Cette fonction est utile pour noter une position afin de s'y repositionner (**fseek**) après un traitement intermédiaire.

```
long ftell(pf)        /* position de type long */
FILE *pf;
```

Exemple:

```
long position;
...
position = ftell(pf);
```

La fonction **ungetc** permet de revenir d'un caractère en arrière lors de la lecture du fichier. On doit fournir dans le premier argument de **ungetc** le dernier caractère lu pour que le positionnement soit valable. La valeur retournée par **ungetc** est son premier argument si le retour en arrière a été réussi, sinon EOF. Il n'est pas autorisé de faire **ungetc**(EOF).

C'est le **getc** suivant qui lira le caractère précédent.

```
ungetc(c, pf)
char c;
FILE *pf;
```

Exemple:

```
FILE *pfil;
char c;
...
while ((c=getc(pfil)) != EOF) {
    if (c == '*') {
        if (ungetc(c,pfil) == c) {
            if ((c=getc(pfil)) == '/') {
                printf ("commentaire\n");
                ...
            }
        }
    }
}
```


3.4. Manipulation de chaînes

La librairie standard offre un certain nombre d'opérations sur les chaînes de caractères que l'on peut classifier en cinq types:

- concaténation de deux chaînes,
- comparaison de deux chaînes,
- copie d'une chaîne dans une autre,
- détermination de la longueur d'une chaîne,
- recherche d'un caractère dans une chaîne.

3.4.1. Concaténation de chaînes

La seconde chaîne est recopiée à la fin de la première. Le caractère nul de la fin de la première chaîne est écrasé par le premier caractère de la seconde chaîne.

```
char *strcat(chaine1, chaine2)
char *chaine1, *chaine2;

char *strncat(chaine1, chaine2, n)
char *chaine1, *chaine2;
int  n;
```

La fonction **strncat** copie au plus **n** caractères. Les deux fonctions retournent un pointeur sur le dernier caractère des deux chaînes concaténées, c'est-à-dire le caractère terminateur \0.

Exemple de programmation des fonctions **strcat** et **strncat** dans la librairie standard:

```
/*
 * strcat : concatenation de la chaine ch2 dans la chaine ch1
 *          retourne un pointeur sur le terminateur \0
 */

char *
strcat(ch1, ch2)
register char *ch1, *ch2;
{
    while (*ch1++) ;                /* parcourt de ch1   */
    --ch1;                          /* elimination du \0 */
    while (*ch1++ = *ch2++) ;       /* copie */
    return (ch2);
}
```

```
/* strncat : concatene au plus n octets de la chaine2 */

char *strncat(ch1, ch2, n)
register *ch1, *ch2;
register n;
{
    while (*ch1++) ;
    --ch1;
    while (*ch1++ = *ch2++)
        if (--n < 0) {
            *--ch1 = '\0';
            break;
        }
    return (ch2);
}
```

Exemple : concaténation d'un nom de fichier lu sur **stdin** avec le suffixe **.pas** :

```
char *strcat();      /* declaration prealable */
char name[16];       /* espace pour le nom    */

main()
{
    printf("nom du fichier =?");
    gets(name);              /* lecture du nom */
    strcat(name, ".pas");    /* concatenation  */
}
```

3.4.2. Comparaison de chaînes

Les chaînes de caractères étant constituées de caractères ASCII ordonnés de 0 à 127, peuvent être comparées, lexicographiquement parlant, selon cette classification. Le jeu de caractères ASCII conserve l'ordre alphabétique pour les minuscules et majuscules.

Les fonctions **strcmp** et **strncmp** retournent un entier :

```
< 0  si la première chaîne se classe avant la seconde
= 0  si les deux chaînes sont identiques
> 0  si la première chaîne se classe après la seconde
# 0  si les deux chaînes sont différentes
```

soit :

```
char *strcat(chainel, chaine2)
char *chainel, *chaine2;

char *strncat(chainel, chaine2, n)
char *chainel, *chaine2;
int  n;
```


Programmation des fonctions **strcmp** et **strncmp** dans la librairie standard :

```
/* strcmp : si ch1 < ch2 retourne < 0
 *          si ch1 = ch2 retourne = 0
 *          si chi > ch2 retourne > 0
 */

strcmp(ch1, ch2)
register char *ch1, *ch2;
{
    while (*ch1 == *ch2++)
        if (*ch1++ == '\0')
            return(0);
    return(*ch1 - *--ch2);
}


/* strncmp : comparaison d'au plus n octets */

strncmp(ch1, ch2, n)
register char *ch1, *ch2;
register n;
{
    while (--n >= 0 && *ch1 == &ch2++)
        if (*ch1++ == '\0')
            return(0);
    return(n < 0 ? 0 : *ch1 - *--ch2);
}
```

Exemple d'un programme qui vérifie sa propre identité, c'est-à-dire si le nom de la commande tapée correspond bien au nom qu'elle devrait avoir :

```
main(argc,argv)
char **argv;
{
    if (!strcmp(*argv,"nomcom")) {
        /* meme nom */
        ...
    } else
        /* nom different */
}
```

3.4.3. Copies de chaînes

Les fonctions de copie **strcpy** et **strncpy** copient une chaîne existante sur une chaîne destinatrice. Les octets peuvent être quelconques à l'exception du

caractère nul \0 qui détermine la fin de chaîne. Si la taille **n** du transfert est plus longue que la chaîne émettrice, la chaîne destinatrice est complétée par des caractères nuls. Inversement, si **n** est inférieur à la taille de la chaîne émettrice, la chaîne réceptrice sera incomplète.

Les deux fonctions de copie retournent le pointeur sur le début de la chaîne destinatrice.

```
char *strcpy(dest, emet)
char *dest, *emet;

char *strncpy(dest, emet, n)
char *dest, *emet;
int  n;
```

Programmation de **strcpy**:

```
/* strcpy   copie la source dans la destinatrice */

char *
strcpy(d, s)
register char *d, *s;
{
    char *destin;
    destin = d;
    while (*d++ = *s++) ;
    return (destin);
}
```

3.4.4. Longueur d'une chaîne

La fonction **strlen** renvoie la longueur de la chaîne spécifiée en argument.

```
strlen(chaine)
char *chaine;
```

Programmation de **strlen**:

```
/* strlen : longueur d'une chaine */

strlen(s)
register char *s;
{
    register n = 0;
    while (*s++) n++;
    return(n);
}
```


3.4.5. Recherche d'un caractère

Les fonctions **index** et **rindex** recherchent respectivement la première ou la dernière occurrence du caractère **c** dans la chaîne mentionnée en argument. La valeur retournée représente le pointeur sur le caractère trouvé, sinon c'est le pointeur nul qui est renvoyé.

```
char *index(chaine, c)
char *chaine, c;

char *rindex(chaine, c)      /* reverse index */
char *chaine, c;
```

Programmation des fonctions **index** et **rindex**:

```
/* index : retourne le pointeur du caractere c cherche */

#define NULL 0

char *index(s, c)
register char *s, c;
{
    do {
        if (*s == c)
            return(s)
    } while (*s++);
    return(NULL);
}
```

```
/* rindex : recherche de la derniere occurrence de c */

#define NULL 0

char *rindex(s, c)
register char *s, c;
{
    register char *r = NULL;
    do {
        if (*s == c)
            r = s;
    } while (*s++);
    return(r);
}
```

3.5. Conversions de chaînes numériques

Une chaîne numérique peut se représenter de deux façons:

- soit en notation usuelle : [−]ddd.ddd
- soit en notation scientifique: [−]d.dddde[±]dd

Exemples de chaînes numériques:

```
2048        -720
84.012      -0.99
7.45e+10    -1.3225e-4
```

La librairie standard offre des fonctions qui convertissent ces chaînes numériques en nombres directement exploitables par la machine (mots, doubles mots, flottants).

- **atoi** conversion d'une chaîne en entier
- **atol** conversion d'une chaîne en entier long
- **atof** conversion d'une chaîne en flottant.

```
atoi(chnum)
char *chnum;

long atol(chnum)
char *chnum;

double atof(chnum)
char *chnum;
```

Exemple de programmation de la fonction **atoi**:

```
atoi(s)
char *s;
{
    int x, n, signe;

    for (x=0 ; s[x] == ' ' || s[x] == '\n'
                           || s[x] == '\t' ; x++) ;
    signe = 1;
    if (s[x] == '+' || s[x] == '-')
        signe = (s[x++] == '+') ? 1 : -1;
    for (n=0 ; s[x] >= '0' && s[x] <= '9' ; x++)
        n = 10 * n + s[x] - '0';
    return(signe * n);
}
```


3.6. Types de caractères

Il est souvent utile de connaître la nature d'un caractère: alphabétique, minuscule, majuscule, chiffre, alphanumérique, espace, ponctuation, caractère imprimable, etc. La librairie standard de C offre un ensemble complet de fonctions donnant la nature d'un caractère que l'utilisateur peut s'affranchir de reprogrammer. L'accès à ces fonctions nécessite préalablement l'inclusion de la directive suivante:

```
#include <ctype.h>
```

Le fichier **ctype.h** est en fait un tableau de 256 entrées indexé par la valeur du caractère à identifier. Chaque case de ce tableau contient des booléens qui précisent la nature du caractère.

Chacune des fonctions suivantes effectue un test sur le caractère courant et renvoie une valeur vraie ou fausse (1 ou 0), selon que le test s'est avéré exact ou non.

```
isalpha(c)  /* test si c est une lettre */

isupper(c)  /* test si c est une lettre majuscule */

islower(c)  /* test si c est une lettre minuscule */

isdigit(c)  /* test si c est un chiffre */

isalnum(c)  /* test si c est alphamumerique */

isspace(c)  /* test si c = ' ', '\t', '\n', '\r', '\f' */

ispunct(c)  /* test si c est une ponctuation */

isprint(c)  /* test si c est imprimable */

iscntrl(c)  /* test si c < 0x20 ou c = 0x7f */

isascii(c)  /* test si 0 <= c <= 127 */
```

Exemple: saut des espaces dans une ligne source:

```
char *sautesp(s)  /* renvoie un pointeur sur le */
char *s;          /* premier caractere non espace */
{
    while(isspace(*s++));   /* saut des espaces */
    return(s);    /* restitution du pointeur */
}
```

3.7. Conversion de caractère

Dans la majorité des cas, les conversions de caractères sont relatives aux lettres majuscules ou minuscules. Ces fonctions de conversion ne nécessitent pas l'emploi d'un fichier en-tête particulier. Elles peuvent être définies comme des macro-instructions:

```
toupper(c)  /* si c est une minuscule il est converti */
            /* en majuscule, sinon pas de conversion  */

tolower(c)  /* si c est une majuscule il est converti */
            /* en minuscule, sinon pas de conversion  */


#define toupper(c)   ((c < 'a') || (c > 'z')) ? c : c-32
#define tolower(c)   ((c < 'A') || (c > 'Z')) ? c : c+32
```

Exemple: conversion des lettres minuscules en majuscule dans une chaîne de caractères:

```
minimaju(s)
char *s;
{
    while(*s) {
        if (islower(*s))
            *s = toupper(*s);
    ++s;
    }
}
```

3.8. Allocation mémoire

Les mécanismes d'allocation et de libération d'espace mémoire n'étant pas supportés par le langage C, ils sont offerts par les fonctions **malloc** et **free** de la librairie standard.

La fonction **malloc** alloue un espace de **n** octets, **n** étant l'argument de la fonction et restitue un pointeur sur le début de l'espace alloué. Si l'espace n'a pu être obtenu pour une raison ou une autre, **malloc** restitue à l'appelant le pointeur nul 0.

```
char *malloc(n)
int  n;
```


Exemple d'utilisation:

```
char *malloc();      /* declaration preliminaire */
char *ptrmem;        /* pointeur zone memoire    */
int taille = SIZE;
...
ptrmem = malloc(taille);  /* appel allocation */
...
```

Exemple: sauvegarde d'un tableau dans un espace mémoire de travail:

```
char *malloc();           /* declaration prealable */
int  tab[SIZE];           /* tableau d'entiers     */
main()
{
    register char *zone; /* pointeur zone de travail */
    if (!(zone = savetab(tab, sizeof(tab))) {
        printf("allocation impossible\n");
        exit (-1);
    }
    ...
}
char *savetab(t, size)
int  *t, size;
{
    char *ptzon, *strncpy();
    if ((ptzon = malloc(size))
        strncpy(ptzon, t, size);  /* copie */
    return(ptzon);
}
```

La fonction inverse **free**, libère l'espace mémoire précédemment alloué en spécifiant simplement le pointeur sur cet espace.

```
free(pointeur)
```

3.9. Fonctions diverses

La fonction **system** permet, à partir du programme utilisateur, de lancer une commande au système d'exploitation. Le nom de la commande est représenté sous la forme d'une chaîne de caractères.

Exemple:

```
system("who am i");
system("date");
```

lance respectivement l'exécution des commandes qui impriment le nom de l'utilisateur courant et la date actuelle.

La fonction **rand** retourne un nombre pseudo-aléatoire sur un entier:

```
int aleat;
aleat = rand();
```

Nous avons présenté ici les fonctions les plus courantes de la librairie standard qui peuvent parfois porter d'autres noms ou présenter des syntaxes légèrement différentes. En outre cette liste est loin d'être exhaustive, car de nombreuses fonctions utilitaires, parfois plus spécifiques de tel ou tel système d'exploitation viennent allonger cette liste. Pour cela, il est préférable de consulter directement les manuels du système concerné.

4. LA LIBRAIRIE MATHÉMATIQUE

Bien que les fonctions de cette librairie soit plus ou moins liées à l'architecture de la machine, en particulier si celle-ci dispose d'un processeur spécialisé de calcul, on retrouve sur la plupart des systèmes les mêmes types de fonctions. L'utilisation des fonctions mathématiques nécessite l'inclusion préalable de la directive:

```
#include <math.h>
```

Les valeurs retournées sont généralement en double précision (type **double**).

Liste de quelques fonctions mathématiques:

```
double exp(x)   double x;     /* exponentiation */
double log(x)   double x;     /* logarithme neperien */
double log10(x) double x;     /* logarithme en base 10 */
double power(x) double x;     /* puissance */
double sqrt(x)  double x;     /* racine carree */
double sin(x)   double x;     /* sinus */
double cos(x)   double x;     /* cosinus */
double atan(x)  double x;     /* arc tangente */
double fabs(x)  double x;     /* valeur absolue */
```


5. LE COMPILATEUR C

Un programme source écrit en C se présente sous la forme d'un fichier texte dont le nom se termine par le suffixe **.c**, afin de mieux le reconnaître des autres fichiers présents sur l'installation. Sous UNIX, les principaux suffixes attribués aux fichiers dans un contexte de programmation sont:

- **.c** programme source C
- **.h** fichier en-tête de déclarations de variables globales et de définitions de constantes (**#define**), ex.: **stdio.h**
- **.s** fichier source en assembleur construit, soit par l'utilisateur, soit produit par le compilateur
- **.o** fichier binaire objet produit à la fin d'une compilation ou d'un assemblage
- **.i** fichier source intermédiaire produit par le macro-préprocesseur C

L'utilisateur sollicite le compilateur C par la commande **cc**. Sous UNIX, cette commande ne correspond pas à l'exécution propre du compilateur C, mais au lancement d'un programme "maître", dont le rôle est d'une part d'analyser la chaîne des arguments de la commande et d'autre part, selon les options qui y sont présentes, de lancer et d'enchaîner les programmes spécifiques correspondants:

- le macro-préprocesseur C,
- le compilateur C,
- l'optimiseur de code,
- l'assembleur,
- l'éditeur de liens.

Le compilateur C est lui-même constitué de deux fichiers indépendants qui correspondent chacun à une passe. La première passe effectue l'analyse syntaxique et délivre le cas échéant des messages d'erreurs, tandis que la seconde passe effectue la génération du code assembleur symbolique propre à la machine à partir de l'arbre syntaxique construit par la passe précédente. Une troisième passe, l'optimisation du code assembleur, est optionnelle. Par défaut, la commande **cc** d'UNIX enchaîne automatiquement les différentes phases nécessaires à la production du fichier exécutable de nom **a.out**.

Sous le système CP/M, ces programmes spécifiques sont appelés à l'aide d'un fichier catalogué de type .SUB. Ce fichier est interprété par la commande SUBMIT qui joue le rôle de la commande **cc** sous UNIX (compilateur C de Whitesmiths). Le fichier **Submit** exécute successivement chaque passe du compilateur, l'assemblage et l'édition des liens pour un seul programme C. Le fichier exécutable porte le même nom que celui du fichier source mais avec le suffixe .COM pour CP/M ou .CMD pour CP/M-86.

Si le fichier **Submit** ne semble pas adapté à la compilation séparée, celle-ci est néanmoins possible par des commandes individuelles. L'assembleur produit en

effet des fichiers relogeables de types .REL compatibles Microsoft et éditables par l'éditeur de liens **L80**.

Sur d'autres systèmes les mises en œuvre de production d'un programme exécutable à partir d'un programme source C se distinguent par quelques variantes, mais le déroulement de la procédure est sensiblement équivalent.

La procédure de compilation simplifiée pour produire un fichier exécutable à partir du fichier source **fichier.c**, se réalise par la commande:

```
cc fichier.c
```

qui successivement appelle le préprocesseur, le compilateur, l'assembleur et l'éditeur de liens. L'exécution du programme s'obtient en tapant:

```
a.out     (UNIX)          fichier     (CP/M)
```

Le nom du fichier exécutable peut être rebaptisé à l'aide des commandes de changement de nom de fichier (**mv**), de copie d'un fichier sur un autre (**cp**), ou encore en spécifiant l'option **–o** suivie du nom du fichier à exécuter, dans la commande **cc**.

Exemples:

```
mv  a.out fichier
cp  a.out fichier
cc  fichier.c –o fichier
```

L'exécution du programme s'obtient alors en tapant:

```
fichier
```

Les principales options de la commande **cc** propres à la compilation sont les suivantes:

–c : production d'un module objet de même nom que le fichier source mais terminé par le suffixe **.o**.

–f : utilisation du module de simulation logicielle de la virgule flottante si le matériel ne la supporte pas.

–O : sollicitation de l'optimiseur de code avant la phase d'assemblage.

–S : compilation et production d'un fichier source dans le langage d'assemblage de la machine. Le nom du fichier se termine par le suffixe **.s**.

–P : exécution seule du macro-préprocesseur C qui transforme les fichiers source .c en fichiers source **.i** prêts à être compilés. Cette option reconstruit un fichier source directement compilable en traitant les directives de compilation (≠ **include**, ≠ **define**, ≠ **if**, etc.). Les fichiers en **.i** ne contiennent plus une seule ligne commençant par ≠ .

Résumé des options de cc

option	action de cc	fichier
-P	préprocesseur	.i
-S	préprocesseur+compilation	.s
-c	préprocesseur+compilation+assemblage	.o
défaut	préprocesseur+compilation+assemblage+édition de liens	a.out

Les principales opions de **cc** propres à l'édition de liens sont:

–lx : référence une librairie **x** dont le nom complet est du type **/lib/libx.a** (sous UNIX).
si x = c **libc** = librairie standard
si x = m **libm** = librairie mathématique

–o : prend le nom explicite qui suit l'option **–o** comme nouveau nom du fichier exécutable.

–s : supprime la table des symboles et la table des bits de relogement afin d'optimiser de l'espace mémoire en exécution.

–i : arrange le code exécutable de manière à ce qu'il soit réentrant. Les espaces mémoires réservés au code et aux données sont séparés et le code est protégé en écriture.

Exemples:

```
cc prog.c
```

produit un fichier exécutable de nom **a.out**.

```
cc –c prog.c
```

produit un fichier binaire objet de nom **prog.o** et n'appelle pas l'éditeur de liens.

```
cc –O prog.c
```

sollicite l'optimiseur de code et produit le fichier **a.out**.

```
cc prog1.c prog2.c prog3.c
```

compile successivement les trois fichiers source pour produire un seul module exécutable **a.out**, tout en conservant les modules objets **prog1.o**, **prog2.o**, **prog3.o**.

```
cc prog.c –lm
```

sollicite la librairie mathématique **libm** au niveau de l'édition de liens.

Dans la mesure où les options de la commande **cc** ne sont pas contradictoires, on peut mettre plusieurs options dans la commande, celles-ci pouvant apparaître dans un ordre quelconque.

Exemple:

```
cc -O prog.c -i -s -lc -o prog
```

cette commande produit un fichier exécutable de nom **prog** (**-o prog**), débarrassé de sa table des symboles (**-s**), réentrant (**-i**), édité avec la librairie standard (**-lc**), et optimisé (**-O**).

Dans le cas où l'on doit recompiler un fichier source alors qu'il était compilé avec d'autres fichiers dans la même commande, il n'est pas nécessaire de compiler à nouveau les autres fichiers. La commande **cc** doit néanmoins les mentionner mais sous la forme de fichiers objets déjà compilés:

```
cc prog1.c prog2.o prog3.o
```

On peut remarquer qu'il n'existe pas d'options relatives à la production d'un listing. En effet, contrairement à la plupart des compilateurs des autres langages, la compilation en C s'effectue silencieusement sans qu'aucun listing n'apparaisse sur le terminal ou soit produit dans un fichier de *"spooling"* ou sur une imprimante.

Le seul moyen d'obtenir le listing d'un fichier source est d'appeler une commande appropriée, par exemple:

```
cat fichier.c > /dev/lp    (sous UNIX)
pip lst:=fichier.c         (sous CP/M)
```

Cependant, lorsqu'une erreur est détectée par le compilateur, celui-ci émet un diagnostic sur le terminal de l'utilisateur en spécifiant le numéro de ligne où s'est produite l'erreur. Une erreur peut déclencher une cascade d'erreurs pour les lignes source suivantes même si celles-ci sont correctes. Ceci s'explique par le fait que l'erreur originelle produit une désynchronisation dans la compilation du programme. Dans certains cas, si les erreurs sont trop nombreuses ou trop graves, le compilateur abandonne son traitement. Le compilateur ne produit un fichier objet que si la compilation ne détecte aucune erreur.

Cependant, certaines erreurs vénielles dues à des considérations de portabilité, appelées des *"warnings"*, peuvent survenir. Le compilateur les signale par des messages de diagnostics, mais la traduction est tout de même effectuée.

Exemples:

```
line 6: warning: illegal combination of pointer and integer
line 9: warning: old-fashioned assignment operator
```


6. OUTILS SPÉCIFIQUES A UNIX

6.1. Lint

Comme nous avons pu le constater, le langage C est un langage très permissif sur les types de données. Il offre des possibilités de conversions implicites au niveau des expressions arithmétiques et explicites (opérateur cast), qui ne seraient pas autorisées dans des langages très rigoureux sur les types tels que PASCAL et surtout ADA.

En outre le langage C ne dispose pas de moyens, intrinsèques ou externes (Run Time Library), pour détecter les débordements de tableaux ou contrôler les types d'arguments transmis à une fonction.

En fait, les auteurs de C, conscients de ces faiblesses, ont préféré concevoir un langage simple d'utilisation et non restrictif. L'idée initiale était de compiler rapidement des programmes, quitte à les recompiler dans l'éventualité d'une erreur à l'exécution.

Le compilateur C, dégagé de la production d'un listing et dépouillé de toute "l'artillerie lourde" relative aux contrôles des types et des débordements, occupe moins d'espace mémoire, s'exécute rapidement et engendre un code objet parfaitement optimisé.

Cependant, un outil de vérification de programmes source C, appelé **lint**, pallie les carences du compilateur C, particulièrement en ce qui concerne le contrôle des types et les problèmes de portabilité.

Lint contient lui-même un compilateur C, le PCC (Portable C Compiler) développé par S. C. Johnson. **Lint** est constitué de deux programmes. Le premier correspond au compilateur C amputé de la partie génération de code. L'analyseur syntaxique y est singulièrement renforcé. Le second programme est en quelque sorte un analyseur sémantique qui travaille sur un fichier source, descripteur de l'arbre syntaxique du programme à analyser produit par la première passe de **lint**.

La commande **lint** accepte plusieurs fichiers source en entrée ainsi qu'un fichier implicite qui contient toute les descriptions d'appel aux fonctions des librairies standard et mathématique.

```
lint prog1.c prog2.c
```

Les principales fonctions de **lint** portent sur:

1. Le contrôle renforcé des types par rapport au compilateur C; particulièrement sur:
   - les opérateurs binaires et les affectations concernées,
   - les opérateurs de sélection de membre de structure,
   - les différences entre les définitions et appels de fonction,
   - les opérations sur les énumérations.

2. La détection d'expressions non portables:

Exemple:

```
char c;
if (((c = getchar( ))  < 0) /* sur PDP-11 */
```

3. L'utilisation des variables:

   — détection des variables et fonctions inutilisées,
   — détection des variables automatiques utilisées avant d'être initialisées:

```
int i;     /* variable automatique */
if (i)
```

4. Le déroulement de l'exécution:

   — détection des séquences de code jamais atteintes,
   — détection des boucles infinies: **while**(1) **for**(;;),
   — détection des boucles dont l'accès ne se fait pas par le sommet (cas des **goto**).

5. Les valeurs de fonction:

   — détection des fonctions qui retournent à la fois une valeur définie et une valeur indéfinie:

```
return (expr);
return;
```

   — détection des fonctions qui retournent une valeur utilisée de manière intermittente:

```
c = getchar( );   /* utilisée */
getchar( );       /* inutilisée */
```

6. Expressions invalides:

   — détection des expressions constantes:

```
if (1 != 0)
```

   — détection des expressions sans effet:

```
*ptr--;          /* pas d'affectation */
```

   — détection des expressions à effet de bord:

```
t1[x] = t2[x++];
```


6.2. Make

Le développement d'une importante application se matérialise le plus souvent par un découpage en différents modules source relativement indépendants les uns des autres. Ces derniers pouvant d'ailleurs être conçus par des programmeurs différents au sein d'une même équipe de travail.

En pratique un fichier source C ne doit pas excéder plus de deux à trois cents lignes de source, ceci, au moins pour trois raisons:

   — sa fonctionnalité,
   — sa lisibilité,
   — sa maintenance.

Cependant, la multiplicité des petits modules fonctionnels compilés séparément présente deux inconvénients majeurs:

   — la dispersion des modules dans de multiples catalogues (directories),
   — la mise en œuvre au moment de l'édition des liens finale.

La commande **make** résoud admirablement ces problèmes de tenue d'un projet et de maintenance de tous les programmes qui constituent l'application.

Le chef de projet peut, dès l'analyse organique de son application terminée, décrire les modules et les mécanismes nécessaires à son édification, dans un fichier "memento", appelé par défaut **makefile** ou **Makefile**.

En général, un fichier **makefile** contient une séquence d'entrées associées à des fichiers exécutables dits *fichiers cibles*. Une entrée est composée de deux parties principales. La première définit le nom du fichier cible à partir des fichiers qui sont nécessaires à sa constitution, tandis que la seconde partie décrit les actions à réaliser pour le construire.

Le but de la commande **make** est de mettre à jour le fichier cible à partir des fichiers dont il dépend. Cette action est effective si le fichier cible n'existe pas, ou si un ou plusieurs fichiers de sa dépendance ont été modifiés depuis sa précédente création.

La première opération de **make** est donc de s'assurer que tous les fichiers existent et sont à jour par rapport au fichier cible. Pour cela **make** réalise une recherche dans le graphe des dépendances pour déterminer le travail qu'il a réellement à faire.

Make opère en utilisant trois sources d'information:

   — la description des dépendances et des actions fournies par l'utilisateur dans le fichier **makefile**,
   — les dates des dernières modifications des fichiers données par le système,
   — des règles internes.

Les deux parties d'une entrée d'un fichier **makefile** sont physiquement représentées sur deux lignes de texte, bien que le caractère ; puisse introduire la

seconde partie, relative aux actions, à la suite de la ligne relative aux dépendances.

Syntaxe simplifiée:

cible: liste des noms des fichiers dépendants
actions à réaliser

La première ligne d'une entrée débute en colonne un par le nom du fichier cible suivi des caractères espace et : et se poursuit par la liste des noms des fichiers dépendants, chacun étant séparé par un espace. Le caractère ;, s'il est présent, annonce que le texte qui suit est l'action à réaliser, sinon celle-ci est décrite sur la ligne suivante, précédée d'une tabulation. Une ligne trop longue peut être repliée à l'aide du caractère \ suivi d'un retour à la ligne. Les commentaires dans un fichier **makefile** sont également possibles et sont annoncés par le caractère #.

Les actions spécifiées sont généralement des commandes **shell**, telles que la compilation d'un fichier source pour produire le fichier objet correspondant, l'impression du fichier, si celui-ci a été modifié depuis sa dernière impression, ou une tout autre commande. Il est en effet souvent utile d'inclure des commandes qui n'ont pas un rapport direct avec l'actualisation du fichier cible. Ainsi, une entrée "save" peut servir à archiver les fichiers source dans une librairie, une entrée "clear" pour supprimer les fichiers temporaires et intermédiaires, etc.

Exemple de contenu d'un fichier **makefile**:

```
application : racine.o module1.o module2.o  # description
        cc racine.o module1.o module2.o -o application
racine.o : racine.c common.h define.h
        cc -c racine.c
module1.o : module1.c common.h
        cc -c module1.c
module2.o : module2.c define2.h
        cc -c module2.c
```

La première ligne décrit les modules qui constituent le fichier cible **application**: **racine.o**, **module1.o** et **module2.o**.

La seconde ligne indique comment construire le programme **application** à partir des modules dont il dépend. Cette ligne n'est exécutée que si au moins un seul fichier du type **.o** est plus récent que le programme exécutable. La commande **cc** reconnaît qu'il s'agit de fichiers objets (suffixes **.o**) et appelle directement l'éditeur de liens **ld**. L'option **-o** précède le nom du programme exécutable à créer.

Les lignes suivantes décrivent individuellement chaque module objet, leur constitution et l'action à effectuer pour les construire, de la même façon que pour les deux premières lignes du fichier.

La troisième ligne se lit ainsi:

le module objet **racine.o** dépend des (:) fichiers source: **racine.c** (programme principal), **common.h** (fichier en-tête commun à d'autres modules), **define.h** (fichier en tête de définitions **#define**). Si l'un de ces fichiers source est plus récent que le fichier **racine.o**, alors **make** compile automatiquement le fichier **racine.c** qui contient les lignes **#include** relatives à **common.h** et **define.h**. L'option **-c** spécifie que **cc** doit produire l'objet **racine.o**, sans lancer l'éditeur de liens.

Un fichier **makefile** peut contenir plusieurs descriptions de programmes cibles. Ceux-ci pouvant être totalement indépendants les uns des autres, mais ayant trait à un même projet.

Exemple:

```
prog1 : fich1.o fich2.o
        cc -o prog1 fich1.o fich2.o
prog2 : objet1.o objet2.o objet3.o
        cc -o prog2 objet1.o objet2.o objet3.o
prog3 : modul1.o modul2.o
        cc -o prog3 modul1.o modul2.o -lm
fich1.o : fich1.c
        cc -c fich1.c
fich2.o : fich2.c
        cc -c fich2.c
objet1.o : ...
        ...
...
```

La commande **make** sans argument, traite par défaut le premier fichier décrit dans **makefile**:

```
make
```

construit le fichier exécutable **prog1** si les objets **fich1.o** ou **fich2.o** ou les sources **fich1.c** et **fich2.c** lui sont ultérieurs.

Si on veut construire le programme **prog2**, il faut le spécifier explicitement dans la commande:

```
make prog2
```

De même on peut spécifier en argument un fichier objet pour s'assurer qu'il sera bien mis à jour:

```
make modul1.o
```

Si l'on désire recréer tous les modules exécutables du fichier **makefile** on peut taper:

```
make prog1 prog2 prog3
```

La commande **make** accepte un certain nombre d'options en argument. Une option est constituée d'une seule lettre précédée d'un tiret (signe –). Les plus couramment utilisées sont:

–n : liste les actions sans les exécuter et permet ainsi de découvrir le déroulement réel des opérations effectuées par **make**.

–s : supprime l'impression des actions; **make** travaille alors silencieusement.

–t : réactualise les dates de dernière modification de tous les fichiers spécifiés dans **makefile**.

–f : indique que le nom qui suit cette option est un nom explicite de fichier de type **makefile**.

Make offre en outre la possibilité de réaliser des macros-définitions pouvant encapsuler des listes de description de fichiers ou des chaînes de commandes. La définition d'une macro-définition se présente sous la forme d'une chaîne de caractères suivie d'un signe d'affectation = et d'une liste de chaînes de caractères.

Exemple de définitions de macros:

```
TOUS : cible1 cible2 cible3
OBJETS = obj1.o obj2.o obj3.o obj4.o
OPTIONS = -c -O
```

La présence dans le texte d'une macro-définition, mise entre parenthèses et précédée du symbole $, provoque sa substitution par la chaîne correspondante placée à droite du signe égal.

Exemple:

```
cible1 : $(OBJETS)
         cc $(OBJETS) -o cible1
obj1 : obj1.c
         cc $(OPTIONS) obj1.c
...      ...
```

Une macro-définition peut être utilisée en argument de la commande. Par exemple, pour créer tous les fichiers cibles il suffit de taper:

```
make TOUS
```

On en déduit la syntaxe générale de la commande **make** qui s'exprime comme suit:

```
make [options] [macro-définition] [fichiers cibles]
```


Pour illustrer l'emploi de la commande **make**, prenons l'exemple d'un éditeur de texte pleine page. L'analyse du problème ayant été correctement menée, le découpage en modules spécifiques se présente comme suit:

–edmain programme principal qui analyse les caractères de fonction (passage en mode édition, insertion, commande, etc.) ou les commandes explicites.

–edcom module où sont regroupées toutes les commandes générales (création, destruction, chargement, sauvegarde, recherche, suppression, substitution, etc.).

–ededit module où sont regroupées toutes les commandes d'édition (insertion, effacement, remplacement, défilement du curseur, sauts de lignes et de pages, etc.).

–edmem module de gestion de l'espace mémoire dans lequel a été placé le fichier (insertion et suppression de caractères).

–edlib ensemble de sous-programmes d'utilisation générale propres à l'application.

–edio ensemble des sous-programmes d'entrées-sorties (gestion de l'écran et de la position du curseur).

–edterm ensemble des sous-programmes d'entrées-sorties dépendant des caractéristiques matérielles du terminal (adressage curseur, effacement de la totalité ou d'une partie d'écran, attributs vidéos, protections de zones d'écran, etc.).

–eddef fichier des définitions communes (**#define**).

–edctrl définition symbolique des caractères de contrôle.

La constitution du fichier **makefile** décrivant l'éditeur de texte se présente alors comme suit:

```
OBJETS = edmain.o edcom.o ededit.o edmem.o edlib.o edio.o edterm.o
CC = cc -c -O                          # compilation C
ed : $(OBJETS)                         # description des objets
       cc $(OBJETS) -i -s -o ed        # edition des liens
edmain.o : edmain.c eddef.h edctrl.h
       $(CC) edmain.c
edcom.o  : edcom.c eddef.h
       $(CC) edcom.c
ededit.o : ededit.c eddef.h
       $(CC) ededit.c
edmem.o  : edmem.c eddef.h
       $(CC) edmem.c
edlib.o  : edlib.c
       $(CC) edlib.c
edio.o   : edio.c eddef.h
       $(CC) edio.c
edterm.o : edterm.c edctrl.h
       $(CC) edterm.c
```

6.3. Cb

La commande **cb** (C beautifier), fournie dans le "package" UNIX offre au programmeur un moyen d'améliorer l'esthétique et la présentation de son programme source C. Cette commande met particulièrement en évidence la structure du programme. Elle fait principalement ressortir les différents niveaux d'imbrication des boucles par une indentation des blocs (instructions composées), délimités entre les accolades { et }. De même, des caractères espace sont rajoutés dans les expressions pour aérer les lignes trop compactes afin d'améliorer leur lisibilité.

Le nom du fichier source à formater est lu sur l'entrée standard **stdin** et son résultat est produit sur la sortie standard **stdout**. C'est pour cette raison que la commande **cb** n'accepte pas le nom de fichier en argument.

La sollicitation de la commande **cb** utilise donc les signes de redirection < et >, ce qui lui donne généralement l'allure suivante:

```
$cb <prog.c  >progbeau.c
```


ANNEXE A

STYLE DE PROGRAMMATION

L'objet de cette annexe est de donner au programmeur quelques conseils de présentation dans l'écriture de son programme afin d'améliorer sa lisibilité et sa maintenance.

Ces recommandations portent principalement sur:

- l'organisation d'un programme C,
- les conventions d'écriture en C.

1. ORGANISATION D'UN PROGRAMME C

Le mécanisme de compilation séparée offert par C doit inciter le programmeur à découper son application en plusieurs modules source distincts. Chaque module réalisant une tâche particulière de l'application. La taille d'un module ne doit pas excéder quelques centaines de lignes C.

La structure de chaque fichier source devrait adopter l'organisation proposée ci-dessous en suivant cet ordre chronologique:

1 — Un bloc commentaire de plusieurs lignes qui décrit le module en essayant de respecter les conventions de présentation suivantes:

- une identification du module par:
  - . son nom,
  - . son numéro de version,
  - . sa date de première réalisation,
  - . sa date de dernière modification,

— une description sommaire de ce que fait le module. Comment et par qui il est utilisé ; ses principales caractéristiques, etc.

— le nom du ou des auteurs ou des personnes qui ont modifié le module.

— un bref rappel historique des versions qui se sont succédées.

2 — Les inclusions de fichiers (lignes **#include**) qui énumèrent dans l'ordre les fichiers en-tête prédéfinis dans le système, de type <fichier.h>, puis les fichiers propres à l'utilisateur, de type "fichier.h".

Exemple :

```
#include <stdio.h>  /* macros lib. standard d'E/S */
#include <ctype.h>  /* types des caracteres */
#include "global.h" /* variables externes communes */
```

3 — les définitions de constantes symboliques et de macros-instructions (lignes **#define**) suivies des définitions de noms de types (lignes **typedef**).

4 — les définitions des variables globales au module (classe **static**) et les déclarations des variables externes importées (spécificateur **extern**). Les variables globales externes devraient figurer dans un fichier en-tête spécifique appelé par **#include**. Par contre les variables globales du module devraient être invisibles des autres modules (fichiers) et donc être explicitement déclarées comme statiques.

Exemple :

```
static char ligne[80];  /* tampon ligne de 80 octets */
```

5 — les fonctions.

Chaque fonction devrait se plier aux règles de présentation suivantes :

— un commentaire d'introduction sur plusieurs lignes spécifiant dans l'ordre :

. le nom de la fonction,
. une description sommaire du rôle de cette fonction,
. la description des paramètres fournis en entrée,
. le nom des variables globales modifiées par la fonction,
. la nature et le type de la valeur retournée,

— la définition de la fonction : son type, c'est-à-dire celui de la valeur retournée devant précéder le nom de la fonction, soit sur la même ligne, soit sur la ligne précédente.

Les arguments de la liste devraient être séparés par une virgule suivie d'un espace pour mieux les dissocier. Tous les arguments devraient être déclarés même s'ils ne sont pas tous utilisés.

— les définitions des variables locales à la fonction, décalées d'une tabulation par rapport au début de la ligne. Placer en tête les variables de type registre puis les variables ordinaires de type automatique.

— les instructions de la fonction en s'efforçant d'appliquer les règles d'indentation pour les boucles ou les instructions de test.

2. CONVENTIONS D'ÉCRITURE EN C

1 — **Les commentaires**

On peut classifier les commentaires en trois groupes :

— les commentaires de type bloc sur plusieurs lignes. Ce type de commentaire est principalement employé pour décrire un fichier source ou une fonction comme nous l'avons vu précédemment. Il pourrait par exemple débuter par un seul /* et se terminer par un seul */ sur la dernière ligne. Les lignes intermédiaires constituant alors le corps du commentaire. Chaque ligne pourrait commencer par un caractère * aligné verticalement sur ceux du début et de la fin afin de mieux faire ressortir le commentaire du texte.

Exemple :

```
/*
 *  nom du module : motpasse.c
 *  version       : 04
 *  projet        : messagerie electronique
 *
 *  date creation     : 21/03/83
 *  date modification : 10/05/83
 *
 *  Ce module a pour but de creer de nouvelles entrees
 *  dans un fichier qui associe un nom d'utilisateur
 *  a un mot de passe pour la consultation du courrier
 *
 *  auteur : un-tel
 */
```

— les commentaires sur une seule ligne encadrés de lignes vides pour introduire une séquence de code spécifique.

Exemple :

```
/* la sequence qui suit ne sert qu'a la mise au point */
```

— les commentaires en vis-à-vis des déclarations ou des instructions. Ceux-ci doivent être courts et dans la mesure du possible être alignés verticalement sur le premier /*.

```
char c;     /* caractere courant */
int  i;     /* index dans le tampon */

        c = getchar();     /* lecture d'un caractere */
        if (c == EOF)      /* test si fin de fichier */
            ...
```

2 — Les variables

Afin de mieux identifier les variables dans le texte, on devrait adopter les conventions d'écriture suivantes :

— les variables débutant par le symbole souligné _ sont habituellement réservés au système.

— les constantes ou macros définies par **#define** ainsi que les types définis par **typedef** devraient être en lettres majuscules.

```
#define DIM 120
typedef long ADRESSE;
```

— les variables servant de masque ou de champ de bits devraient également être en majuscules.

```
#define BIT_PARITE 0x80
```

— les variables globales devraient avoir leur première lettre en majuscule et les autres en minuscules, comme pour un nom propre.

```
char Buffer[256];
```

— les variables locales devraient être en lettres minuscules.

```
int i, j, k, index ;
```

— les membres des structures et des unions devraient commencer par les deux ou trois premiers caractères du nom de la structure ou de l'union, suivis d'un _ et de l'identificateur de la variable.

```
int DAT_mois ;
```


3 — Le codage des expressions et des instructions

La partie code du module doit mettre en évidence sa structure en blocs en utilisant l'indentation. Il est déconseillé de mettre plusieurs instructions sur une même ligne. Si la ligne est trop longue, il faudrait la découper en plusieurs parties logiques, les lignes suivantes étant alors indentées par rapport à la première.

Les instructions et expressions devraient suivre les règles d'écriture suivantes :

— les mots clés suivis d'une expression entre parenthèses devraient être séparés de la parenthèse ouvrante par un espace.

```
if (expr)
for (i = 1; i < 10; i++)
switch (c)
```

— la liste des arguments d'une fonction mise entre parenthèses devrait suivre immédiatement le nom de la fonction.

```
printf(stderr, "erreur\n");
```

— les éléments d'une liste d'arguments devraient être séparés par une virgule suivie d'un espace.

```
main(argc, argv, envp)
```

— il ne devrait pas y avoir d'espace entre les opérateurs primaires ( ) [ ] . –> et leurs opérandes.

```
getchar(c)
tab[i]
date.heure
ptr->nom
```

— il ne devrait pas y avoir d'espace entre un opérateur unaire et son opérande.

```
* p ++   /* incorrect */
*p++     /* correct   */
```

— tous les opérateurs binaires devraient être entourés d'un espace de part et d'autre.

```
a=b*(c+d)          /* incorrect */
a = b * (c + d)    /* correct */
```

— les opérateurs d'affectation devraient précéder immédiatement le signe =.

```
a + = b    /* incorrect */
a += b     /* correct   */
```

— la première expression d'un opérateur conditionnel devrait être parenthésée.

```
max = a>b ? a : b       /* incorrect */
max = (a > b) ? a : b   /* correct */
```

— le terminateur d'instruction ; devrait être précédé d'un espace.

ANNEXE B

ÉTUDE COMPARATIVE
Programme de calcul des nombres premiers en plusieurs langages

C, PASCAL, ADA, FORTRAN, BASIC, COBOL, FORTH

Cette comparaison de programmes écrits dans divers langages s'inspire des articles de Jim et Gary Gilbreath parus dans les numéros de septembre 1981 et de janvier 1983 de la revue Byte. Ces articles avaient pour but de proposer un "benchmark" (programme étalon) pour comparer les performances des différents langages sur une machine donnée.

Le programme calcule les nombres premiers compris entre 3 et 2 000 par la méthode du crible d'Eratosthène, les nombres 0, 1 et 2 étant supposés connus.

Dans l'étude qui nous préoccupe, on se propose de présenter ce programme afin que le lecteur, s'il connaît déjà un langage de programmation, puisse le comparer avec la version écrite en C et en tirer les conclusions qui lui semblent positives.

VERSION C

```
/* calcul des nombres premiers par le crible d'Eratosthene */

#define SIZE 1000
char flags[SIZE+1];
main()
{
    register i,k,premier,compte;

    compte = 0;                         /* compteur */
    for(i=0 ; i <= SIZE ; i++)
        flags[i] = 1;                   /* init du tableau a 1 */
    for(i=0 ; i <= SIZE ; i++) {
        if (flags[i]) {                 /* premier trouve */
            premier = i + i + 3;
            printf ("%d\n",premier);
            for (k=i+premier ; k <= SIZE ; k += premier)
                flags[k] = 0;           /* tuer tous les multiples */
            compte++;
        }
    }
    printf ("%d nombres premiers",compte);
}
```

VERSION PASCAL

```
(* calcul des nombres premiers par le crible d'Eratosthene *)

program premiers;
const SIZE = 1000;
var flags : array [0..SIZE] of boolean;
    i,k,premier,compte : integer;
begin
    compte := 0;
    for i := 0 to SIZE do flags[i] := true;
    for i := 0 to SIZE do
        if flags[i] then begin
            premier := i + i + 3;
            writeln(premier);
            k := i + premier;
            while k <= SIZE do begin
                flags[k] := false;
                k := k + premier
            end;
            compte := compte + 1
        end;
    writeln (compte,' nombres premiers')
end.
```


VERSION ADA

```
-  calcul des nombres premiers par le crible d'Eratosthene
--
PACKAGE BODY Premiers IS
        Size : CONSTANT := 1000,          -- dimension du tableau
        Flags : ARRAY (0..Size) OF BYTE; -- tableau de flags
        I , K : INTEGER;
        Premier : INTEGER;                -- nombre premier
        Compte : INTEGER;                 -- compte de premiers
        Y : BYTE:= BYTE(1)  N : BYTE:= BYTE(0);
BEGIN   Compte := 0;
        FOR I IN 0 .. Size LOOP           -- tableau a true
            Flags(I) := Y;                -- Yes
        END LOOP;
        FOR I IN 0 .. Size LOOP           -- boucle centrale
            IF Flags(I) = Y THEN          -- premier trouve
                Premier := I + I + 3;     -- valeur du nombre
                K := K + Premier;         -- index pour multiples
                WHILE K <= Size LOOP      -- tuer les multiples
                    Flags(K) := N;        -- mettre a "false"
                    K := K + Premier;     -- prochain multiple
                END LOOP;
                PUT(Premier); NEW_LINE;   -- impression du nombre
                Compte := Compte+1;       -- comptage des nombres
            END IF;
         END LOOP;
         PUT(Compte), PUT(" nombres premiers");
END Premiers;
```

VERSION FORTRAN

```
C calcul des nombres premiers par le crible d'Eratosthene
C
      logical flags(1001)
      integer i,j,k,compte,premier

      do 10 i = 1,1001
10        flags(i) = .true.
      do 20 i = 1,1001
          if (.not. flags(i)) goto 20
          premier = i + i + 3
          write(6,100) premier
100       format(1X,I6)
          compte = compte + 1
          k = i + premier
          if (k .gt. 1001) go to 20
          do 30 j = k,1001,premier
30            flags(j) = .false.
20    continue
      write(6,200) compte
200   format(1X,I6,' nombres premiers')
      stop
      end
```

VERSION BASIC

```
1   REM calcul des nombres premiers par le crible d'Eratos
2   REM
3   DEFINT A-Z
10  dim flags(1001)
20  compte = 0
30  for i = 0 to 1000
40      flags(i) = 1
50  next i
60  for i = 0 to 1000
70      if flags(i) = 0 goto 160
80          premier = i + i + 3
90          print premier
100         k = i + premier
110         while k <= 1000
120             flags(k) = 0
130             k = k + premier
140         wend
150         compte = compte + 1
160 next i
170 print compte,"nombres premiers"
180 end
```


VERSION COBOL

```
* calcul des nombres premiers par le crible d'Eratosthene
*
IDENTIFICATION DIVISION.
PROGRAM-ID. PREMIERS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 I             PIC 9(4) COMP.
77 K             PIC 9(4) COMP.
77 PREMIER       PIC 9(4) COMP.
77 COMPTE        PIC 9(4) COMP.
01 FLAGS-TAB.
    02 FLAGS     PIC 9 COMP OCCURS 1001 TIMES.
PROCEDURE DIVISION.
MAIN.   MOVE 0 TO COMPTE.
        PERFORM INITFLAGS VARYING I FROM 1 BY 1 UNTIL I > 1001.
        PERFORM BOUCLE THRU FIN-BOUCLE VARYING I FROM 1 BY 1
            UNTIL I > 1001.
        DISPLAY COMPTE ' nombres premiers'.
        STOP RUN.
BOUCLE. IF FLAGS (I) = 0 GO TO FIN-BOUCLE.
        COMPUTE PREMIER = I + I + 3.
        COMPUTE K = I + PREMIER.
        PERFORM RAZMULTIPLES  UNTIL K > 1001.
        ADD 1 TO COMPTE.
        DISPLAY PREMIER.
FIN-BOUCLE.
        EXIT.
INITFLAGS.
        MOVE 1 TO FLAGS (I).
RAZMULTIPLES.
        MOVE 0 TO FLAGS (K).
        ADD PREMIER TO K.
```

VERSION FORTH

```
( calcul des nombres premiers par le crible d'Eratosthene )

1000 CONSTANT SIZE
0 VARIABLE FLAGS  SIZE ALLOT

: DO-PREMIERS
  FLAGS SIZE 1 FILL  (initialisation du tableau)
  0 ( 0 COMPTE ) SIZE 0
    DO FLAGS I + C@
       IF I DUP + 3 + DUP I +
          BEGIN DUP SIZE <
          WHILE 0 OVER FLAGS +  C! OVER + REPEAT
          DROP DROP 1+
       THEN
    LOOP
    . ." nombres premiers" ;
```


ANNEXE C

LES PRIMITIVES D'UNIX

nom	*description*	*exemple*
access	détermination de l'accès d'un fichier	access("name",mode);
alarm	déclenchement d'un délai en secondes	alarm(sec);
chdir	changement du directory implicite	r = chdir("dirname");
chmod	changement des protection d'un fichier	r=chmod("name",mode);
chown	changement de propriétaire	r=chown("name",own,gr);
close	fermeture d'un fichier	r = close(fd);
creat	création d'un fichier	fd=creat("name",mode),
dup	duplication d'un descripteur de fichier	dup(fd);
exec	exécution d'un fichier exécutable	exec("name",arg,0);
exit	terminaison d'un processus fils	exit(0);
fork	création d'un processus fils	pid = fork();
fstat	statut d'un fichier déjà ouvert	fstat(fd,buffer);
getgid	obtention du numéro de groupe	gid = getgid();
getpid	numéro du processus en cours	pid = getpid();
getuid	obtention du numéro d'utilisateur	uid = getuid();
ioctl	action et contrôle sur les périph.	ioctl(fd,fonct,arg);
kill	envoi d'un signal à un processus	kill(pid,signal);
link	établissement d'un lien entre fichiers	r=link("old","new");
lseek	positionnement dans un fichier	lseek(fd,(long)pos,0);
mknod	création d'un fichier périphérique	mknod("name",mode,adr);
mount	montage d'un volume dans une hiérarchie	mount("spec","name",0);
nice	modification de la priorité d'exécution	nice(0);
open	ouverture d'un fichier	fd = open("name",0);
pause	attente de l'arrivée d'un signal	pause();
pipe	ouverture d'un canal de communication	r = pipe(canal);
read	lecture d'une séquence d'octets	n = read(fd,buf,siz);
setgid	initialisation d'un groupe	setgid(group-id);
setuid	initialisation d'un numéro utilisateur	setuid(user-id);
signal	déclaration d'une exception	signal(SIGINT,int);
sleep	suspension du temps d'exécution	sleep(secondes);
stat	statut d'un fichier	stat("name",buf);
stime	initialisation de la date et de l'heure	stime(ptr-date);
sync	mise à jour des i-nodes actifs	sync();
time	obtention de la date et de l'heure	time(adresse);
times	temps d'exécution d'un processus	times(buf-time);
umount	démontage d'un volume de la hiérarchie	umount("special");
unlink	destruction d'une entrée de directory	r = unlink("name");
utime	modification de la date de mise à jour	untime("name",date);
wait	attente de la mort d'un processus fils	r = wait(0);
write	écriture d'une séquence d'octets	n = write(fd,buf,siz);