www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Langage C

BIBERSTEIN Olivier

QUELOZ Pierre-Antoine

BRULHART Dominique

Janvier 1993

Le cours de Turbo-C niveau 1 se déroule en 15 leçons de 3 heures.

Chacune d'elles débutera par une révision de la leçon précédente et il y sera répondu à toutes les questions que vous pourriez poser.

Une brève introduction indiquera quels seront les points traités pendant la leçon courante.

En général, la première moitié de la leçon sera consacrée à des notions théoriques de programmation.

Durant la seconde moitié, un ou deux exercices, illustrant les notions vues juste avant, permettront de les mettre en pratique.

D'autres exercices seront proposés, qui seront à faire pour la semaine suivante. Le meilleur moyen d'apprendre à programmer étant par la pratique, il est indispensable que chacun s'efforce de faire le maximum d'exercices pour lui-même et de poser toutes les questions nécessaires à la compréhension des sujets présentés.

Une correction sera distribuée et discutée au début de la leçon suivante.

Deux épreuves obligatoires auront lieu, la première vers le milieu du cours et la seconde lors du dernier cours. La première ne sera pas prise en compte pour l'obtention du certificat, elle vous permettra de mesurer vos connaissances et de faire le point sur le travail accompli. La seconde devra par contre être réussie. Vous aurez le droit d'utiliser tous les documents que vous voudrez, l'essentiel étant que vous sachiez vous débrouiller, pas que vous connaissiez tout par coeur.

I.PRESENTATION DU COURS
A. Organisation du travail 1
B. Table des matières 2 à 4
C. Mise en page du support de cours 5

II. NOTIONS INFORMATIQUES DE BASE
A. Schéma d'un système informatique 6 à 8
1. Architecture du matériel 6
2. Environnement logiciel 7 à 8
B. Programmation structurée 9 à 13
1. Structuration du traitement 9 à 10
2. Structuration des données 11
3. Qualités d'un programme 12 à 13

III. ENVIRONNEMENT DE TRAVAIL
A. DOS 14 à 16
1. Disques 14
2. Fichiers 14
3. Arborescence 14
4. Programmes exécutables 14
5. Résumé des commandes 15 à 16
B. TURBO C 17 à 22
1. Menus 17 à 18
2. Edition 19
3. Compilation et exécution 20
4. Déverminage 20
5. Touches de fonction 21
6. Compilations sous DOS 121 à 122

IV. DONNEES
A. Constantes 26
B. Typage
1. Notions intutives 29
2. Types scalaires 30
3. Types énumérés 63 à 65
4. Tableaux
a. Une dimension 48 à 50
b. Chaînes de caractères 51
c. Dimension supérieure 107 à 108
5. Structures
a. Simples 66 à 70
b. Unions 112
6. Définition de types 84 à 85
7. Pointeurs
a. Définitions 86 à 87

b. Passage de paramètres par référence 91 à 92
c. Tableaux 113
8. Conversions implicites 82
C. Variables
1. Mémoire 28
2. Définition et déclaration 28
3. Typage 29
4. Valeurs par défaut
a. Types scalaires 28
b. Tableaux 109 à 110
c. Structures 111
5. Classes d'allocation 58 à 59
D. Codage de l'information 73 à 78
1. Entiers 75 à 76
2. Caractères 77
3. Virgule flottante 78

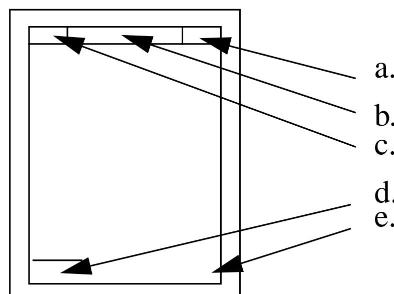
V. CONTROLE
A. Idées générales
1. Programme prinicipal 23
2. Instructions et expressions 27
3. Eléments lexicaux 24 à 25
a. Mise en page 24
b. Commentaires 24
c. Identificateurs 24
d. Mots réservés 25
e. Bloc 25
f. Points virgule 25
B. Opérateurs
1. Arithmétiques 31
2. Relationnels 32
3. Affectation 33
4. Logiques 34
5. Incrémentation 62
6. Manipulations de bits 81
7. sizeof() 80
8. Manipulations de pointeurs 88 à 90
9. Précédence 126
10. Conversions de types 83
C. Fonctions
1. Sous-programmes 43
2. Définition "originale" 44
3. Définition ANSI 45
4. Appel et retour 46
5. Exemple complet 47

6. Visibilité 57
7. Arguments de main() 114
D. Contrôle de flux 39
1. if-else 40
2. while 41
3. for 42
4. Ruptures 53 à 54
5. switch 60 à 61
6. do-while 55
7. Alternative 56

VI. DECOMPOSITION DE L'APPLICATION
A. Préprocesseur 115 à 118
1. define 115
2. undef 117
3.Compilation conditionnelle 117
4. include 118
B. Modularisation 119 à 120
C. Make 123 à 125

VII. LIBRAIRIES STANDARD
A. Manipulations de chaînes 71 à 72
B. Entrées/Sorties standard
1. Sorties formatées 35 à 36
2. Entrées formatées 37 à 38
3. Chaînes de caractères 52
4. Types spéciaux (unsigned, octal,...) 79
C. Fichiers 93 à 106
1. Fichiers de la librairie standard 93
2. Flots standard, structure FILE, stdio.h 94
3. Ouverture de fichier 95
4. Fermeture 96
5. Entrées-sorties de haut niveau 97
6. Lecture de caractères 98
7. Ecriture de caractères 99
8. Chaînes de caractères 100
9. Indicateurs d'erreur 101
10. Gestion des flux 102 à 103
11. Accès direct 104 à 105
12. Entrées-sorties binaires 106
D. Contrôle de l'écran en mode texte 135 à 138
E. Survol des fonctions de librairie 127 à 134

Toutes les pages de ce cours auront la même structure, afin que l'apprentissage et la révision soient facilités.



a. Dans le coin supérieur droit, apparaît un numéro de page. Cette numérotation est sensée refléter l'ordre dans lequel les différentes notions seront abordées.

b. Au centre, en haut, figure toujours le titre de la page. Généralement, un seul sujet est traité par page.

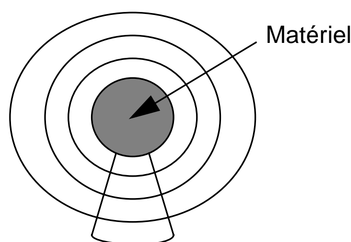
c. Dans le coin supérieur gauche, apparaît un numéro composé de chiffres romains, de lettres et de chiffres arabes. Ce numéro permet de situer la page par rapport aux autres pages traitant du même sujet. Pour des raisons de compréhension, les notions ne seront pas présentées dans cet ordre au cours, mais plutôt petit à petit, afin de respecter un rythme d'apprentissage. Ces numéros vous permettront de retrouver une notion plus rapidement, une fois que tous les sujets seront regroupés conformément à la table des matières.

d. Séparés par un trait du reste du texte, se trouvent les renvois, annotations et remarques éventuelles. Indiqués par un numéro dans le texte.

e. Enfin, un triangle dans le coin inférieur droit signifie que les explications continuent à la page suivante. Un carré que les explications sont terminées.

De plus, les différents types de caractères utilisés permettront de distinguer facilement s'il s'agit d'**explications**, de **titres** ou de `morceaux de programmes`.

Enfin, lors de l'énoncé de formes syntaxiques (commandes, fonctions, etc.) les italiques indiqueront quelles parties doivent être remplacée, les parties entre crochets carrés [...] seront considérées comme optionnelles, et les accolades {...} indiqueront une partie pouvant être répétée un nombre quelconque de fois. Commes ces symboles ont également une signification pour le langage étudié, il sera explicitement indiqué quand ils feront partie de l'expression et seront obligatoires.



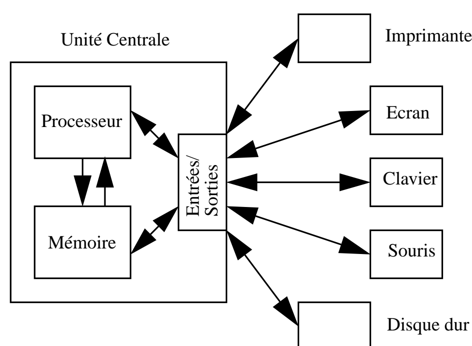
Voici une manière de schématiser un système inforrmatique par un modèle en forme d'oignon.

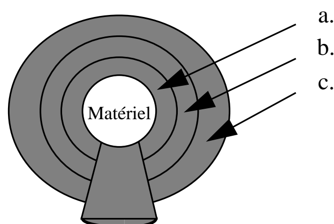
NIVEAU MATERIEL

Au centre de tout système informatique, on trouve du **matériel** électronique. On subdivise généralement cette catégorie en deux groupes.

Tout d'abord l'**unité centrale**, c'est le boîtier de l'ordinateur. Il comprend la **mémoire** où les informations que l'ordinateur est en train de traiter (données) et les programmes en cours d'exécution sont stockés et le **processeur central**, capable d'effectuer des calculs et d'exécuter des programmes. L'unité centrale contient en outre des **dispositifs d'entrées/sorties**. On regroupe sous cette applelation un ensemble de circuits dont le rôle est de faire transiter les données entre le processeur ou la mémoire et les périphériques.

D'autre part, les nombreux **périphériques** qui permettent la communication de l´unité centrale avec le monde extérieur. On trouve dans cette catégorie l'écran, le clavier, la souris, l'imprimante, les modems, etc. On regroupe aussi dans cette catégorie les **mémoires de masse**, par exemple les disquettes, disques durs, bandes magnétiques, etc. qui permettent le stockage permanent de grandes quantités de données et de programmes. Il serait bien entendu impossible de stocker tous les logiciels d'application et tous les fichiers de données d'un utilisateur en même temps dans la mémoire centrale.





NIVEAU LOGICIEL

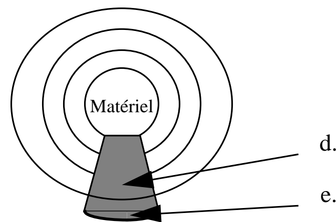
Au niveau logiciel, on s'intéresse à tout ce qui est programmé. Bien entendu, les logiciels du système dépendent fortement du matériel, ce qui explique que le matériel soit au centre. De plus, dans ce schéma, chaque niveau englobe le niveau inférieur, indiquant par là que l'on peut utiliser les programmes sans se soucier de ce qui se passe plus au centre. Cette **abstraction** est très importante en informatique, c'est elle qui permet de construire des programmes de plus en plus performants. Sans cela, il faudrait à chaque opération se demander ce qui se passe au niveau le plus bas, par exemple, savoir comment les caractères sont écrits sur une disquette quand on enregistre un texte.

On distingue généralement trois couches logicielles.

a. La première représente le **système d'exploitation**. Il s'agit du programme le plus important de tout le système informatique. C'est lui qui s'occupe de gérer les différentes ressources de l'unité centrale et des périphériques, par exemple de faire exécuter un programme ou de lire et écrire des fichiers sur disque. C'est seulement à travers lui que nous pouvons accéder au matériel.

b. La couche suivante comporte des programmes utilitaires indispensables à la programmation. Par exemple, un **compilateur**, qui traduit les programmes écrits dans un langage de haut niveau, proches du raisonnement humain comme C, en du langage machine, beaucoup moins compréhensible pour nous mais que le matériel peut exécuter. On trouve aussi à ce niveau, l'**éditeur de liens**. Il permet de lier un programme que nous écrivons avec de très nombreux autres programmes déjà présents dans des **bibliothèques**, ce qui nous permet de profiter de ce qui a déjà été fait par d'autres et nous évite de tout recommencer à zéro. De telles bibliothèques permettent par exemple de dessiner sur l'écran ou de lire et écrire des données sur une disquette.

c. Au niveau supérieur, on trouve des **programmes utilitaires**, comme des éditeurs, qui permettent simplement de taper un texte et de le sauver dans un fichier sur disque; des traitements de texte, ressemblant aux éditeurs, mais avec lesquels nous pouvons également faire de la mise en page où utiliser différentes écritures; des gestionnaires de bases de donnés, des programmes de communication, etc.



d. Cette zone montre tous les niveaux logiciels où l'on trouve bien souvent des programmes écrits en C. Le système d'exploitation UNIX est écrit en grande partie en langage C (90 %). Ils ont d'ailleurs été créés conjointement dans les années 1970. Des compilateurs ont aussi été écrits en C, ainsi que de très nombreuses applications, comme des logiciels de communication et des applications scientifiques et commerciales. La principale qualité du C est qu'il permet d'agir jusqu'au niveau du matériel avec une grande efficacité.

e. On peut encore remarquer qu'un produit logiciel dépasse généralement au niveau supérieur, par exemple des applications scientifiques, commerciales, etc. qui s'appuient sur tous les niveaux inférieurs et qui présentent une **interface** à l'utilisateur, par exemple au moyen de fenêtres, d'icônes et de menus, cachant ainsi tout le travail effectué aux étages inférieurs. Une telle interface est conçue de telle manière que même un utilisateur n'ayant aucune notion d'informatique puisse l'utiliser. On s'efforcera donc de présenter les choses de la manière la plus intuitive possible.

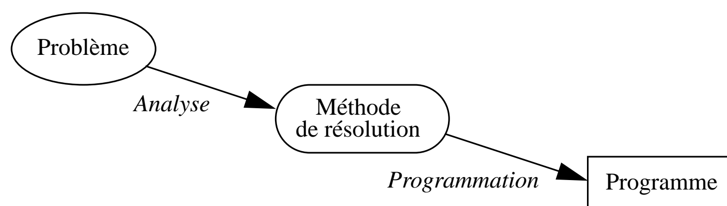
Analyse

Tout programme est issu de l'intention de résoudre un problème au moyen de l'outil informatique.

Il faudra donc tout d'abord analyser le problème, ce qui devra nous conduire à des **algorithmes** adaptés à la résolution du problème et à des **données** qui nous permettront de représenter l'information à traiter.

Il faut ensuite traduire ces concepts, afin que l'ordinateur puisse les interpréter et finalement résoudre le problème désiré. Voilà en quoi consiste l'activité de **programmation**. D'une façon générale, on peut dire que la programmation consiste, à partir d'un problème donné, à réaliser un programme dont l'exécution apporte une solution satisfaisante au problème posé.

Schéma de développement d'une application



Programmation structurée

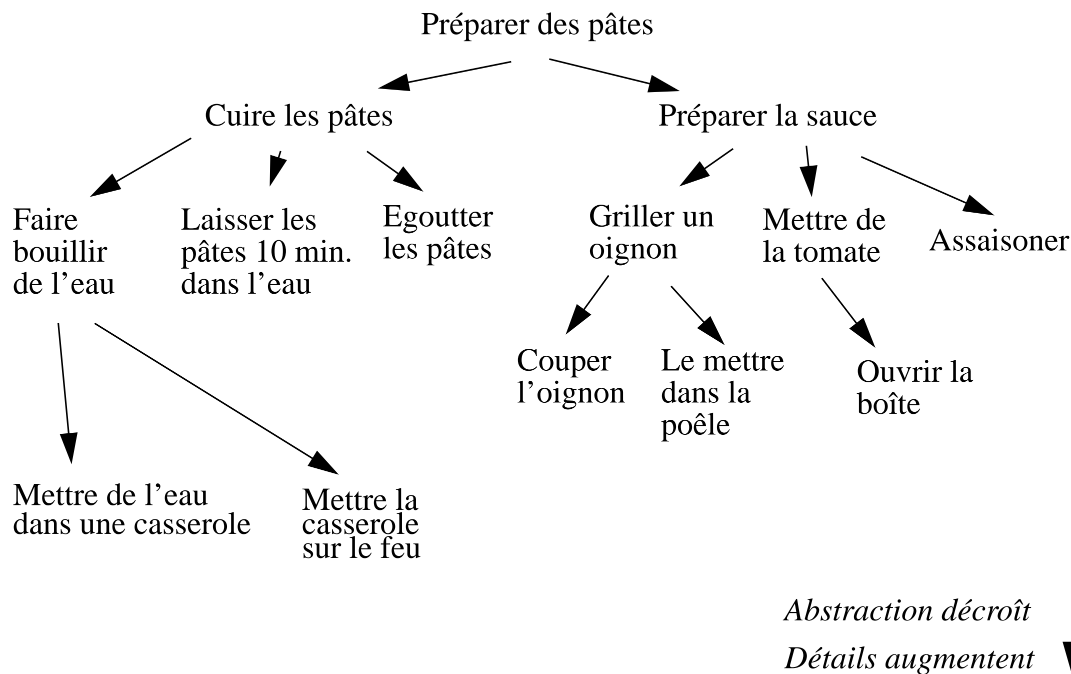
Tout problème peut se décomposer en sous-problèmes plus simples à résoudre.

Ainsi, au fur et mesure de l'analyse, on affine notre connaissance du problème, les détails augmentent et le niveau d'abstraction diminue.

Chaque sous-problème devient donc plus élémentaire. Ce processus de "raffinement" se poursuit jusqu'à arriver au niveau d'abstraction du langage de programmation que l'on va utiliser. A ce niveau, on peut utiliser les éléments du langage pour représenter le problème. Il faut remarquer que la décomposition ne s'arrête pas là, pour que la machine soit capable d'exécuter notre programme, il devra encore être traduit en langage machine. Cette partie du travail nous est épargnée par le compilateur.

On observe donc que la résolution du problème poura être décrite par une suite finie d'opérations. C'est ce que l'on nomme un **algorithme**. Il est d'autre part important que tout programme se termine, faute de quoi on risque d'attendre un résultat indéfiniment.

Une recette de cuisine est très semblable à un algorithme. Le problème est d'obtenir un mets savoureux et les éléments du langage utilisés des actions simples que tout le monde peut effectuer. L'exemple suivant montre comment on analyse (synonyme: décompose) un problème.



L'algorithme (la recette) qui découle de cette décomposition pourrait être:

- Mettre de l'eau dans une casserole.
- Mettre la casserole sur le feu.
- Laisser les pâtes 10 minutes dans l'eau.
- Egoutter les pâtes.
- Couper l'oignon.
- Mettre l'oignon dans la poële.
- Ouvrir la boîte de tomates.
- Assaisoner.

La décomposition peut continuer bien plus loin dans les détails, mais il n'est pas question de programmation ici. Toutefois, ce genre de formulation montre rapidement les incohérences de la méthode. N'avons-nous pas oublié quelque-chose?

Ce genre d'analyse peut être appliquée à une multitude de problèmes.

On appelle **données** toutes les sortes d'informations que nous désirons manipuler avec un programme informatique. Une image, un son, un texte, des résultats d'expériences, des informations boursières, un rapport financier, une fenêtre sur un écran, la position d'une fusée, la consommation d'une voiture, etc. sont autant de sortes de données différentes que l'on peut être amené à traiter à l'aide d'un ordinateur. Bien entendu, l'ordinateur n'est capable de manipuler que des données très simples, correspondant à des mots mémoire, codées en binaire et de longueur fixe. Il ne peut pas traiter ces données en tant que telles et il est nécessaire de trouver un moyen de les représenter sous une forme qui lui convienne.

Nous avons vu comment l'analyse du problème nous conduit à un algorithme par un raffinement successif en sous-problèmes. Parallèlement, il faut analyser les informations à traiter afin d'obtenir une **structure de données** cohérente que notre algorithme sera capable de manipuler. Ainsi, nous devons tirer parti de notre connaissance de l'information et de ses propriétés.

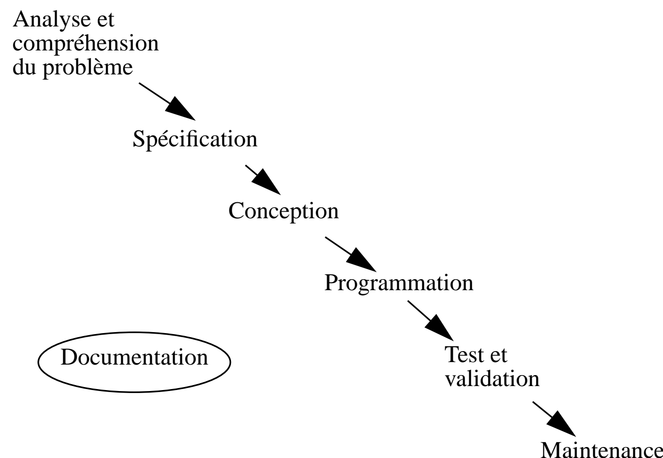
Pour cela, il faut considérer le genre des données à chaque niveau de notre décomposition. Cette décomposition ne pourra s'arrêter que lorsque le niveau d'abstraction des structures de données fournies par langage de programmation sera atteint. Comme pour la décomposition de l'algorithme, le niveau d'abstraction sera aussi bien souvent encore trop élevé pour que la machine puisse interpréter ces données telles que nous les décrivons dans notre programme. A nouveau, le compilateur finira le travail pour nous, descendant ainsi jusqu'au niveau du mot mémoire par le biais de différents codages.

L'exemple suivant montre une manière d'analyser une donnée complexe.

- Imaginons que nous avons à programmer une application qui permette à l'utilisateur de mémoriser des extraits d'articles de presse et de les collectionner. La donnée la plus abstraite manipulée par le programme serait une collection d'articles.
- Ensuite, il faudrait prévoir des sous-programmes capables de manipuler des articles entiers, par exemple pour les classer.
- Au niveau inférieur, un article est composé d'un ensemble de lignes de texte.
- Enfin, une ligne de texte est composée de caractères. La donnée la plus élémentaire que notre programme aurait à traiter serait un caractère. C permet justement de manipuler facilement des caractères.

Nous pouvons donc imaginer qu'au niveau le plus bas de la décomposition nous programerons la gestion des caractèrs, au niveau supérieur la gestion d'une ligne entière qui reposerait sur le niveau inférieur, encore plus haut, une partie du programme permettrait de manipuler des articles entiers, enfin, au sommet de la pyramide, l'application qui offre à l'utilisateur les moyens de gérer des collections d'article entiers et reposerait sur tous les niveaux précédents. Chaque problème est de plus en plus abstrait, à mesure que l'on s'éloigne de la gestion des caractères pour manipuler des données plus complexes. Mais il bénéficie aussi de tout le travail qui est fait par les niveaux inférieurs.

L'activité de programmation, ou plus généralement de développement de projets, se décompose en plusieurs phases qui constituent le **cycle de vie** du logiciel. Une telle décomposition est indispensable pour le bon déroulement de très grands projets, chacune des phases représentant énormément de travail, tout doit de dérouler dans l'ordre. Imaginer seulement qu'il faille changer des centaines de lignes de programmes à cause d'une mauvaise analyse du problème...



La documentation est essentielle au bon déroulement du projet et doit se faire en parallèle avec chacune des phases.

Qualités d'un programme :

Une fois qu'un programme est terminé et installé sur site, le travail n'en est pas pour autant fini. En effet, une erreur peut apparaître des mois plus tard ou alors l'utilisateur veut pouvoir **amplifier** ou modifier son application. Il faut bien avoir à l'esprit qu'un programme n'est pas juste ou faux, et que sa qualité dépend plus des points suivants que d'un modèle rigide.

• **Fiabilité**: Le programme doit résoudre le problème pour lequel il a été conçu et cela sans faute.

• **Robustesse**: Une erreur de saisie ou de manipulation, une panne ... ne doivent pas mettre en péril l'ensemble de l'application (par. ex. perte de la base de données). Pour éviter ce genre de catastrophes, l'analyse doit être faite complètement et prévoir toutes les manipulations possibles.

• **Maintenance** : Un programme doit pouvoir être relu même des années plus tard. Un autre programmeur doit être capable de continuer ou de reprendre le travail. Pour cela, il faut une programmation claire, nommer les objets que le programme manipule de façon explicite et commenter abondamment le texte du programme. Certains extrémistes préconisant même autant de lignes de commentaires que de lignes d'instruc-

tions. Sans aller jusque là, l'algorithme utilisé doit pouvoir se reconnaître au premier coup d'oeil et il ne faudrait en tout cas jamais avoir à récrire une partie incompréhensible du programme.

• **Efficacité**: L'algorithme doit être adapté afin que la vitesse d'exécution soit minimale, l'information doit être structurée de façon efficace, et ne pas être trop redondante, afin de ne pas gaspiller trop de place en mémoire.

• **Ergonomie**: L'utilisateur doit pouvoir se servir facilement et avec plaisir de l'application, l'apprentissage étant si possible aisé et des raccourcis permettant à l'utilisateur expérimenté de gagner du temps.

MS-DOS est le système d'exploitation le plus répandu sur tous les ordinateurs de la famille PC, en anglais Personal Computer, nom donné par IBM au premier ordinateur de la famille. Les initiales MS rappelant qu'il s'agit d'un produit de la firme MicroSoft. Le sigle DOS signifie en anglais Disk Operating System (donc système d'exploitation). Comme il est à la base de l'utilisation du PC, il est important d'en connaître les principes généraux.

Disques

Lorsqu'on allume un PC, le DOS est lancé et prend contrôle du système. Il affiche quelques caractères du type

```
C:\>
```

qui nous invitent à entrer une commande afin d'exécuter un programme d'application ou manipuler des données qui se trouvent enregistrées sur disque. La gestion des disques est faite par le DOS qui est capable de trouver ce qu'ils contiennent. Chaque disque est identifié par une lettre suivie de ':'. Par exemple 'A:' et 'B:' pour les lecteurs de disquette et en général 'C:' pour un disque dur.

Fichiers

Toutes les données enregistrées sur une disquette ou un disque dur sont contenues dans un fichier. Cela peut être une image, un texte, un son, un programme dans un langage de haut niveau, un fichier de commandes pour DOS, un programme exécutable par le processeur, une base de données, etc. Pour que nous sachions à qui nous avons affaire, nous pouvons donner un nom à chacun d'entre eux. C'est par ce nom que nous indiquons à DOS quel fichier nous voulons manipuler. Ce nom est composé de huit caractères, plus trois que l'on nomme extension et qui nous permettent de reconnaître le type de données contenues dans le fichier. Par exemple

Pour un programme C:	moyenne.c
Pour un programme Pascal:	loto.pas
Pour un texte:	lettre1.txt
Pour un programme exécutable:	tc.exe

Arborescence

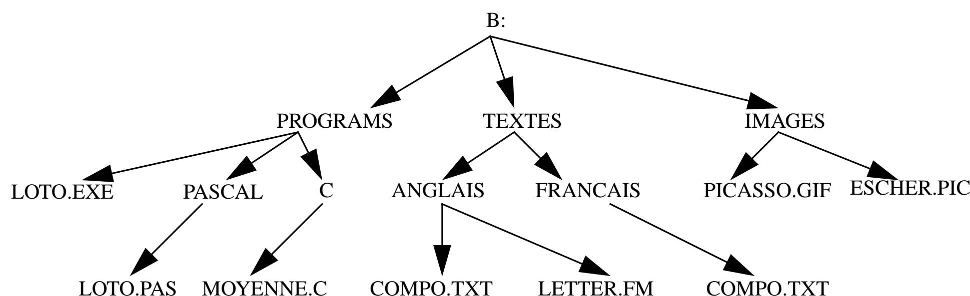
Vu la grande taille des disques durs, il est possible d'y stocker des milliers de fichiers. Afin d'y voir plus clair, DOS est capable de gérer une structure arborescente de **répertoires**. Nous pouvons ainsi regrouper les fichiers sur un sujet donné dans un même répertoire auquel nous donnons un nom. Chaque répertoire peut contenir des sous-répertoires, etc.

Exécutables

Un fichier est exécutable si l'on peut en demander l'exécution depuis le DOS. Trois types de fichiers sont exécutables. Les fichiers **batch** (extension .BAT) qui sont composés d'instructions que le DOS exécute. Ils sont très utiles pour effectuer des tâches répé-

titives. Les fichiers .COM et .EXE qui sont exécutés par le processeur après avoir été chargés en mémoire centrale.

Exemple de structuration des fichiers sur une disquette:



Résumé des commandes DOS

Caractères génériques

Pour remplacer un nombre quelconque de caractères dans un nom de fichier, on utilise '*'. Pour remplacer un seul caractère quelconque '?'. L'astérisque ou le point d'interrogation seront automatiquement remplacés par DOS lorsqu'il trouvera un fichier dont le nom correspond à ce que nous voulons.

Lancer une application

Seuls trois types de fichiers sont dit exécutables, ceux dont l'extension est `.BAT`, `.COM` ou `.EXE`. Le déclenchement d'un programme se fait simplement en écrivant son nom-code (l'extension n'est pas obligatoire) et en ponctuant par la touche [RETOUR].

Lire le contenu d'un disque ou d'un répertoire

On utilise la commande

`DIR [`*`NOM_REPERTOIRE`*`] [/P][/W]`

les arguments entre crochets sont dits optionnels, on peut les mettre seulement en cas de nécessité. [`/P`] demande que le défilement s'arrête après chaque page, [`/W`] que seuls les noms apparaissent et sur plusieurs colonnes.

Copier un ou plusieurs fichiers

On utilise la commande

`COPY [`*`NOM_SOURCE`*`] [`*`NOM_CIBLE`*`] [/V]`

[`/V`] : effectue une vérification de la copie.
Cette commande crée un duplicata du fichier *`NOM_SOURCE`* et lui donne le nom *`NOM_CIBLE`*.

Effacer un fichier

On utilise la commande

DEL *NOM_FICHIER*

Attention en utilisant les caractères génériques, DEL *.* efface tous vos fichiers.

Copier une disquette en entier

Afin de ne pas travailler sur des originaux, on peut 'dupliquer' la disquette et stocker l'original en lieu sûr. Utiliser la commande

DISKCOPY [*SOURCE* [*CIBLE*]]

[*SOURCE*] est la disquette originale. [*CIBLE*]: attention toutes les données seront effacées.

Vérifier la copie d'une disquette

Utiliser la commande

DISKCOMP [*LECTEUR_1* [*LECTEUR_2*]]

Changer le nom d'un fichier

On utilise la commade

RENAME *ANCIEN_NOM NOUVEAU_NOM*

Voir le contenu d'un fichier

La commande

TYPE *NOM*

affiche à l'écran le contenu du fichier.

Structure arborescente

Imaginer la structure d'un disque comme un arbre dont les branches sont dirigées vers le bas. La racine est le nom du disque, par exemple A:. On appelle le chemin d'accès à un fichier l'ensembre des sous-répertoires qu'il faut parcourir pour l'atteindre.

- Création d'un sous-répertoire:

MD [*CHEMIN*] *NOM*

- Aller dans un sous-répertoire:

CD [*LECTEUR*] [*CHEMIN*]

[..] : revient au ss-rep. parent
[\] : revient directement à la racine

- Effacer un sous-répertoire

RD *NOM*

le sous-répertoire en question doit être vide.

Préparer une nouvelle disquette

Une disquette neuve ne peut pas être utilisée si elle n'a pas été formatée, c'est-à-dire structurée pour recevoir des données. Méfiance lors de cette opération, si la disquette contient déjà des données, elles seront toutes perdues! Utiliser la commande

FORMAT [*LECTEUR*] [/S][/V]

[/S] : après le formatage, les fichiers du système d'exploitation (DOS) permettant de 'démarrer' l'ordinateur sont copiés.
[/V] : permet de donner un nom à la disquette (max.11 car.)

On lance l'environnement intégré TURBO C depuis le DOS par la commande

```
tc
```

Cet environnement utilise les techniques de menus déroulants à structure arborescente et de boîtes de dialogue (fenêtres).

L'environnement est intégré signifie que toute la programmation d'une application peut se faire sans utililser d'autre logiciel. Pour ce faire, il contient plusieurs modules. Un module d'édition qui permet de taper et de corriger le programme, tout en le sauvant dans un fichier. Un module de compilation, qui permet de vérifier que la syntaxe est correcte et de générer du code exécutable. Si une erreur est détectée à la compilation, la ligne où elle s'est produite peut être vue directement dans l'éditeur pour aller la corriger. TURBO C possède enfin un module d'exécution qui permet d'exécuter le programme sans retourner au DOS et qui permet d'autre part de surveiller l'exécution d'un programme pas à pas, très utile pour le déverminage (debugging en anglais). Il propose en plus une fonctionnalité d'**aide 'en ligne'**, ce qui signifie qu'à tout moment, vous pouvez presser la touche de fonction **[F1]** et qu'il vous donnera des renseignements sur ce que vous êtes en train de faire.

L'écran se divise en quatre parties:

- La barre de menu qui permet la sélection de commandes.
- La fenêtre d'édition: partie permettant l'édition de programmes.
- La fenêtre de sortie qui contient les résultats affichés.
- La ligne d'aide qui affiche certaines commandes et les touches de fonction associées.

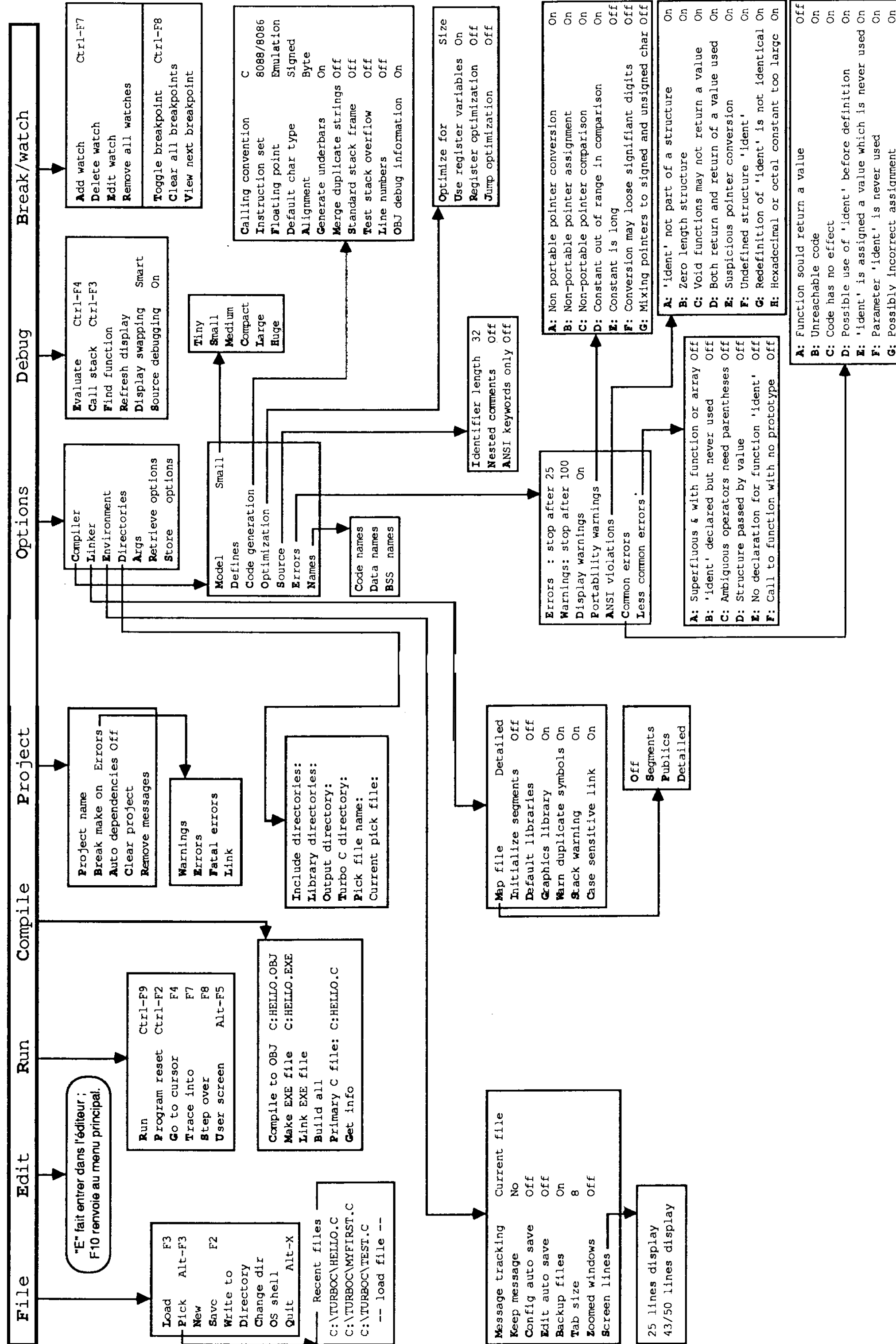
Les menus

La plupart des opérations que TURBO C peut effectuer pour nous sont commandées par le biais des menus. Par ailleurs, ceux-ci permettent de configurer l'environnement pour les besoins de l'application.

Les menus sont regroupés selon les opérations auxquelles ils donnent accès. Ils s'utililisent très facilement avec un peu d'habitude.

Pour sélectionner un élément ou une action dans un menu, il faut en général utiliser la touche de fonction [F10]. Ensuite, on peut sélectionner ce qui nous intéresse soit avec les flèches de déplacement du curseur et en validant par [RETURN], soit en tapant l'initiale de l'élément désiré. Pour remonter au menu du dessus, il faut presser la touche d'échappemnt [ESC]. Si vous désirez obtenir de l'aide sur un élément de menu, vous pouvez, comme partout ailleurs dans l'environnement, presser la touche de fonction [F1].

Le tableau suivant représente la totalité des menus utilisés dans l'environnement TURBO C. Il est tiré du *Manuel de l'utilistateur TURBO C, BORLAND 1988.*



ENVIRONNEMENT

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

L'éditeur

L'éditeur du TURBO C est intégré à l'environnement et répond aux conventions du logiciel WordStar. C'est un éditeur dit pleine page car il permet de voir une portion du programme sur l'écran, de se déplacer à l'intérieur et de faire défiler le programme vers le haut et vers le bas. Il est à noter qu'il n'est pas nécessaire d'utiliser cet éditeur, n'importe quel autre peut faire l'affaire.

Fichiers

On se sert du menu FILE pour la gestion des fichiers de programme:

- Load permet de récupérer un programme stocké sur disque dans l'éditeur.
- Pick permet de retrouver rapidement les derniers fichiers utilisés.
- New prépare un nouveau travail **!sauver l'ancien!**
- Save sauvegarde le travail en cours.
- Write To permet de sauver le fichier en cours sous un nouveau nom.
- Change dir : change le répertoire actif.

Edition

Pour éditer un programme, il faut que le curseur se trouve dans la fenêtre d'édition. Si ce n'est pas le cas, sélectionner EDIT dans le menu principal. Sur la première ligne sous les menus, se trouvent des indications sur l'édition en cours.

- position du curseur (ligne, colonne)
- les modes d'édition (insertion / suppression et indentation/non)
- le chemin et le nom du fichier édité.

Pour exécuter une commande spéciale d'édition, il faut la sélectionner par une séquence de touches dont les principales sont décrites ci-dessous.

Contrôle du curseur

- un écran vers le haut CTRL+R ou PgUp
- un écran vers le bas CTRL+C ou PgDn
- début de ligne Home
- fin de ligne End
- mémoriser positin du curseur CTRL+K suivi de 0, 1, 2 ou 3
- ancienne position du curseur CTRL+Q suivi de 0, 1, 2 ou 3
- tabulation CTRL+I ou Tab

Modifications

- mode insertion/superposition CTRL+V ou Ins
- effacement vers la droite CTRL+G ou Del
- effacement vers la gauche CTRL+H ou BS
- suppression d'une ligne CTRL+Y
- insertion d'une ligne CTRL+N
- rappel de la dernière ligne effacée CTRL+QL

ENVIRONNEMENT

Commandes de blocs

- marquage du début d’un bloc CTRL+KB
- marquage de la fin d’un bloc CTRL+KK
- copie d’un bloc marqué CTRL+KC
- déplacement d’un bloc marqué CTRL+KV
- lecture d’un bloc depuis le disque CTRL+KR
- écriture d’un bloc sur le disque CTRL+KW
- effacement d’un bloc marqué CTRL+KY
- impression d’un bloc marqué CTRL+KP
- bascule surbrillance/non CTRL+KH

Recherches et remplacements :

- recherche d’une séquence CTRL+QF
- remplacement d’une séquence CTRL+QA
- répétition de la recherche CTRL+L

Divers

• il est également possible d’obtenir de l’aide sur le mot qui se trouve positionné sur le curseur en pressant CTRL+[F1]

Compilation

Une fois l’édition de votre programme terminée, vous devez demander à ce qu’il soit compilé, c’est-à-dire traduit en code exécutable. Si votre programme est simple, vous pouvez utiliser le menu **RUN**. TURBO C va lancer la compilation si le programme a été modifié depuis la dernière, puis exécuter le programme s’il n’y a pas eu d’erreur.

Un programme C complexe peut être composé de plusieurs fichiers. Pour compiler un tel programme, il sera nécessaire d’utiliser les menus COMPILE et PROJECT.

Déverminage

Si vous avez des erreurs logiques dans votre programme, vous pouvez les détecter assez facilement à l’aide des facilités offertes par TURBO C. En effet, il vous permet d’exécuter de petites tranches de programme, tout en observant la valeur des variables. Il est ainsi possible de suivre son déroulement et de repérer des comportements imprévus.

Menu RUN

• Program reset permet de revenir au début du programme.

• Go to cursor permet d’exécuter toutes les instructions jusqu’à celle où se trouve le curseur dans la fenêtre d’édition.

• Trace into permet de suivre le déroulement pas à pas.

• Step over permet d’exécuter une fonction sans "entrer" à l’intérieur. Le pro-

gramme s'arrête sur l'instruction qui suit la fonction.

• User screen permet de voir ce qu'il y a sur l'écran "derrière" TURBO C. On utilise également cette fonction si le programme s'est terminé sans qu'on ait eu le temps de voir les derniers résultats. Dans ce cas, TURBO C revient sur le "devant" de l'écran, mais vous pouvez toujours voir vos données à l'aide de ce menu.

Menu DEBUG

• Evaluate permet d'évaluer une expression sans sortir du contexte. L'effet de l'évaluation sera le même que si l'expression s'était trouvée dans le programme à l'endroit où l'on se trouve. Si on quitte par [ESC] les modifications seront oubliées.

• Call stack permet d'observer la pile d'exécution. C'est utile pour voir quelles fonctions ont été appelées.

• Find fuction permet de visualiser une fonction en donnant son nom.

• Source debugging est une bascule qui doit se trouver sur ON si l'on désire effectuer du déverminage.

Menu BREAK/WATCH

• Add watch permet de surveiller la valeur d'une expression (ou d'une variable) dans la fentêtre au bas de l'écran.

• Delete watch permet de supprimer une expression surveillée.

• Edit watch de modifier les expressions surveillées.

• Remove all watches de supprimer toutes les surveillances.

• Toggle breakpoint permet d'ajouter et de supprimer un point d'arrêt dans le programme. Lorsque le programme s'exécute, il va automatiquement s'arrêter à tous les points d'arrêt.

• Clear all breakpoints supprime tous les points d'arrêt.

• View next breakpoint place le curseur sur le prochain point d'arrêt dans le texte du programme.

Touches de fonction et divers

Les touches de fonctions, combinées avec la touche [CTRL] et la touche [ALT] permettent de très nombreux raccourcis. Il n'est en effet pas nécessaire d'utiliser les menus si vous connaissez le raccourci correspondant. Lorsque vous pouvez effectuer une opération à l'aide d'une telle combinaison de touches, TURBO C vous l'indique généralement. La barre d'aide tout au bas de l'écran contient l'essentiel. Les raccourcis suivants sont les plus utilisés.

• F1 : aide. Permet d'accéder à un écran d'aide. De plus un index des sujets traités est disponible.

• F2 : sauve. Sauvegarde automatiquement le fichier courant.

• F3 : charge. Charge en mémoire un fichier de travail.

• F10 : menu. Permet d’accéder au menu principal.

• ESC : échapper. retourne l’état précédent.

• ALT-F1 : aide. Affiche l’écran d’aide le plus récemment consulté. Ou, si on se trouve dans l’éditeur des informations sur ce qui se trouve sous le curseur.

• ALT-F3 : ‘Pick’. Correspond la selection ‘Pick’ du menu File

• ALT-F5 : Affiche le dernier écran graphique composé par la dernière exécution du programme.

• CTRL-F9 : Run. Exécute le programme courant.

• ALT-X : Fin. Termine la session et retourne au DOS.

Dans le menu FILE

• Quit permet de quitter l’environnement TURBO C.

• OS shell permet de lancer DOS pour exécuter d’autres commandes, sans pour autant quitter l’environnement. Une fois que vous avez terminé, la commande

```
EXIT
```

permet de revenir dans TURBO C.

Tout programme C doit nécessairement contenir une partie principale (main en anglais). Ce sont les instructions qui se trouvent dans cette partie qui seront exécutées lorsque le programme sera lancé. On dénote un programme principal par

```
main ()
{

    Mettre ici les instructions du programme principal

}
```

Ainsi, le compilateur saura que les instructions entre l'accolade ouvrante et l'accolade fermante (on appelle cela un **bloc**) doivent êtres exécutées lorsque le programme est lancé. S'il ne trouve pas un tel bloc, le programme ne sera jamais exécuté.

Il faut noter qu'un programme en C ne doit pas nécessairement commencer par `main`, ni se terminer par une accolade et ce sera d'ailleurs rarement le cas. La seule chose qui compte est qu'il y ait un tel bloc quelque part dans le programme.

Les parenthèses sont obligatoires, elles indiquent que l'on est en train de décrire une **fonction**. Une fonction est bout de programme dans lequel on décrit un ensemble d'actions à effectuer (algorithme). Nous verrons plus tard une définition plus précise et de nombreux exemples de fonctions.

MISE EN PAGE

Comme presque tous les langages de programmation modernes, C n'impose aucune contrainte pour la mise en page des programmes. Le nombre d'espaces entre les mots peut être quelconque, des lignes blanches peuvent être insérées pour augmenter la lisibilité et plusieurs instructions peuvent se trouver sur la même ligne.

COMMENTAIRES

Toute suite de caractères encadrée par les symboles '/*' et '*/' correspond à un commentaire et ne joue évidemment aucun rôle lors de l'exécution du programme. Les commentaires sont donc ignorés par le compilateur mais sont indispensables sur le plan de la documentation du programme. Ils peuvent être placés n'importe où dans le programme et s'étendre sur plusieurs lignes mais ne doivent pas être imbriqués. On utilise également les commentaires pour "enlever" une portion du programme sans devoir la retaper si on change d'avis. En effet, le compilateur ne voit pas les instructions qui se trouvent mises en commentaires et se comporte comme si nous les avions enlevées. Voici quelques exemples de commentaires.

```
/* Ceci est un commentaire */

/* Les commentaires peuvent etre formules sur
   plusieurs lignes */

/* Il est frequent
 * de formuler
 * des commentaires ainsi
 */

/* Ceci est /* incorrect a cause */ de l'imbrication */
```

IDENTIFICATEURS

Tous les langages de programmation permettent de **nommer** des objets afin de les manipuler plus simplement. Tous les mots ne peuvent être employés à titre d'identificateur, voici les règles à respecter pour former un identificateur correct.

• Un identificateur est constitué de lettres ('a'...'z', 'A'..'Z'), de chiffres ('0'...'9') et éventuellement du caractère souligné ('_').

• Un identificateur doit impérativement commencer par une lettre ou un '_'et ne pas faire partie de la liste des mots réservés. Il s'agit d'un petit nombre de mots, généralement des commandes qui n'ont pas le droit d'être utilisés comme identificateurs à cause des confusions que cela pourrait entraîner.

• Attention, une distinction est faite entre les caractères majuscules et minuscules (NbLignes et nblignes sont deux identificateurs différents).

• La norme ANSI a fixé a 31 le nombre de caractères significatifs d'un identificateur bien que la longueur de ce dernier puisse être plus importante.

CONTROLE

MOTS RESERVES

Voici les mots réservés du langage C. Ils ne doivent pas être utilisés comme identificateurs.La signification de tous ces mots sera donnée au fur et à mesure de notre étude.

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

Les mots réservés `const`, `signed` et `volatile` sont propres a la norme ANSI.

BLOC

On a fréquemment besoin de regrouper un certain nombre d'instructions pour réaliser un traitement.
Dans ce cas, il faut encadrer ces instructions par les caractères '{' et '}' . On appelle alors ce groupe d'instructions un bloc d'instructions. Les blocs d'instructions pouvant etre **imbriqués**, on a l'habitude de décaler de quelques caractères vers la droite toutes les instructions faisant partie du bloc en question; on appelle ceci l'**indentation**. Il est important de respecter cette convention pour des raisons de lisibilité.
Partout ou on peut mettre une instruction, on peut aussi mettre un bloc d'instructions, qui sera exécuté comme s'il ne s'agissait que d'une grande instruction.
A l'intérieur de tous les blocs, on peut commencer par déclarer des variables locales, qui n'existeront qu'à l'intérieur du bloc[1], et ensuite mettre des instructions.

POINTS VIRGULE

En C, le point virgule est le **terminateur** d'instruction, on en met donc après chacune d'entre elles.
On met également un point virgule après chaque **déclaration**.
Les habitués du langage Pascal seront surpris d'en trouver à des endroits où ils n'ont pas l'habitude d'en voir, comme avant un `else` par exemple. En effet, en Pascal, on met un point virgule seulement entre deux instructions.

CONTROLE

1. Voir paragraphe V.C.5 pour les règles de visibilité.

Ce sont les données les plus simples qu'un programme C sache manipuler. Elles nous permettent d'écrire une valeur n'importe où dans le programme. On distingue quatre types de constantes:

- Les nombres entiers.
- Les nombres réels, avec partie décimale.
- Les caractères.
- Les chaînes de caractères.

Les constantes **entières** s'écrivent comme on en a l'habitude. 123, -10, 3570000 sont des exemples de constantes entières. Lorsqu'il rencontre une telle valeur dans le programme, le compilateur s'occupe de la coder en binaire et de la représenter en mémoire.

Les constantes **réelles** s'écrivent avec un **point** décimal. 123.0, -12.345, 3.14, 20349.092 sont des constantes réelles.

Les **caractères** sont entourés par des **apostrophes simples**. 'a', 'A', 'X' et '3' sont des caractères et seront reconnus par le compilateur.

Les **chaînes de caractères** servent à représenter des petites portions de texte et sont encadrées par des **guillemets**. "Bonjour", "Au revoir", "Le résultat est faux" sont des chaînes de caractères. Un chaîne de caractères peut être écrite sur plusieurs lignes dans le programme. Pour cela, il faut que le dernier caractère de chaque ligne soit '\'.

C permet de **nommer** les constantes à l'aide de la commande

```
#define NOM valeur
```

L'habitude veut que les noms de constantes soient écrits en **majuscules**. Ce n'est pas obligatoire.

Quelques règles doivent être respectées:

- Le `#` doit être le premier caractère de la ligne
- Une telle définition peut se trouver n'importe où dans le programme, elle sera connue dans toutes les lignes de code qui suivront.
- Chaque occurence de `NOM` dans la suite du programme, sera systématiquement remplacée par **tout le reste de la ligne**. Il ne faut donc généralement pas mettre de point virgule après une définition.
- Une telle constante ne peut pas être modifiée au cours du programme (cela a-t-il un sens?)

Ce mécanisme de définitions est très **utile** lorsqu'une valeur se retrouve très souvent dans le programme, et qu'elle est susceptible de changer à un moment de la vie du programme (changement de machine, etc.). Il suffit de modifier la valeur de la constante et il n'y a plus besoin de chercher toutes les occurences pour les modifier.

DONNEES

Pour que l'ordinateur fasse quelque chose, il faut que le programme lui dise quoi faire. Une **instruction** est un ordre que l'on donne à la machine. Evidemment, nous ne pouvons pas lui demander n'importe quoi, il faut que le compilateur comprenne de quoi il s'agit. Nous devons donc nous adapter et utiliser les instructions qu'il nous offre. Ces instructions sont très nombreuses et nous en verrons une bonne partie. A noter que ces dernières sont toujours terminées par un caractère point-virgule.

Une instruction très utilisée est celle qui ordonne à l'ordinateur de nous afficher un texte ou un nombre sur l'écran. Cette instruction se nomme

```
printf( );
```

entre les deux parenthèses, nous mettrons ce que nous voulons lui faire imprimer.

L'ordinateur est également capable de calculer. Nous pouvons lui demander de calculer une valeur qui nous intéresse en écrivant une **expression**. La valeur de l'expression sera évaluée. 12*360.0+5 est une expression numérique. On appelle **opérateurs** les symboles comme * et + qui permettent de décrire des expressions, et **opérandes** les valeurs sur lesquelles l'opérateur s'applique. C offre aussi une gamme très vaste d'opérateurs, et pas seulement numériques.

En résumé

L'ordinateur **exécute les instructions** que nous mettons dans notre programme et il est capable d'**évaluer des expressions**.

Le programme suivant illustre ces deux concepts.

```
#include <stdio.h>
main()
{
      printf ("Bonjour tout le monde\n");
      printf ("Je suis capable de calculer 2+2=%d\n",2+2);
      printf ("Au revoir");
}
```

La première ligne du programme n'est ni une instruction, ni une expression, c'est seulement une note disant au compilateur que nous allons utiliser une fonction d'entrées/sorties (`printf();`). Les autres éléments sont simples. Un programme principal contenant trois instructions, toutes des `printf()`. Entre les parenthèses, nous avons placé des **arguments**. Les arguments du premier et du dernier `printf()` sont des constantes du type chaîne de caractères. La chaîne dans la première instruction se termine par `\n`, cela signifie qu'il faut revenir à la ligne à la fin. Le second `printf()` est un peu différent, on lui a donné deux arguments séparés par une virgule. Le premier est une chaîne, le second une expression numérique. Lorsqu'il va exécuter cette instruction, l'ordinateur va tout d'abord évaluer l'expression `2+2`, donc appliquer l'opérateur `+` aux deux constantes numériques `2`.Ensuite, il va afficher la chaîne en remplaçant le `%d` par le résultat de l'évaluation.

CONTROLE

Nous aimerions **stocker** les données que notre programme va traiter de manière efficace. A cet effet, une partie de la **mémoire centrale** du PC peut être réservée et utilisée. Dans cette zone, nous pouvons écrire nos données et ensuite aller les relire, les modifier, etc. Il serait vraiment fastidieux de se souvenir quel endroit exact de la mémoire correspond à l'une ou l'autre des nombreuses données que notre programme doit stocker pour travailler.

En réponse à ce problème, même les langages les moins évolués comme l'assembleur permettent de définir des **variables**. Cela consiste à associer un nom (un **identificateur**) à une zone de mémoire. Une zone de mémoire ainsi nommée pourra être référencée au moyen de ce nom. Ainsi, plus besoin de savoir où exactement sont stockées nos données, nous les appelons par leur nom et C s'occupe de les trouver.

Pour comprendre cette idée, il suffit d'imaginer les cases mémoire comme les tiroirs d'un très grand meuble, l'identificateur est une étiquette que l'on colle sur le tiroir. Si nous demandons à C de nous fournir ce qui se trouve dans un certain tiroir au moyen de ce nom, il pourra le retrouver et nous le mettre à disposition. De même, si nous désirons déposer quelque chose dans l'un des tiroirs, il suffit de donner son nom à C et il s'occupera de le ranger pour nous.

Par ailleurs, il faut, pour des raisons de **codage** et d'**occupation mémoire** que chaque variable soit d'un type bien déterminé. Une chaîne de caractères et un entier ne seront pas codés de la même manière en mémoire centrale en ne prendront pas le même nombre de bits.

Pour que le système fonctionne sans accroc, il est nécessaire que toutes les variables que nous utilisons soient **définies** avant que l'on essaie de les utiliser. Sinon, C ne saura pas de quelle variable nous parlons et sera incapable de comprendre ce que nous voulons.

En résumé, **définir** une variable, c'est associer un **type** et un **identificateur.**

La **syntaxe** de cette opération est

```
nom_du_type ident [= val_initiale] {,ident [= val_initiale]};
```

Par exemple, si dans un bloc on désire utiliser une variable `resultat` du type entier, une variable `une_lettre` du type caractère, une variable `moyenne` du type virgule flottante, nous devrons mettre ces trois définitions au début du bloc.

```
int resultat;
char une_lettre;
float moyenne;
```

Suite à cette définition, le compilateur aura rajouté trois variables dans sa **table des symboles** et réservé la mémoire nécessaire pour les stocker. Il aura également **initialisé** les variables si nous l'avons demandé en faisant suivre le nom de la variable du signe '=' et d'une valeur. Les variables auront cette valeur la première fois que nous les utiliserons.

DONNEES

NOTIONS DE TYPAGE

Les données qu'un programme est amené à traiter sont souvent de **natures différentes**. Par exemple une température et le nom d'une personne n'ont pas beaucoup de points communs.

Simultanément, les **opérations** que l'on peut vouloir effectuer sur une donnée dépendent de sa nature propre. Ainsi, si la soustraction de deux températures est sensée, celle de deux noms l'est déjà beaucoup moins; en tout cas, elle ne devrait pas se faire de la même manière.

Le typage permet de **classifier** ces informations selon leur nature et, pour les langages de programmation, selon les opérations qui peuvent s'appliquer sur ces dernières.

La plupart des langages de programmations actuels effectuent sur les expressions un certain nombre de **contrôles de type** lors de la compilation. Ces contrôles consistent à vérifier qu'il y a bien concordance de type entre les arguments et les fonctions ou les opérateurs. Cette vérification de types **statique** (lors de la compilation) permet de détecter grand nombre d'incohérences ou d'erreurs avant l'exécution des programmes, il en découle aussi une solidité plus importante des logiciels. C n'est toutefois pas très exigeant.

Par ailleurs, l'**espace mémoire** nécessaire au stockage d'une donnée et la manière de la coder en binaire dépend de son type.

Pour les langages de programmation, le type d'une donnée résume :

- l'éventail des valeurs possibles de la donnée
- les opérations applicables sur la donnée
- l'espace mémoire occupé par la donnée

la manière de coder la donnée en binaire.

On classifie les types en deux grandes catégories. Les **types scalaires** et les **types composés**. Les variables de type scalaire ne peuvent stocker qu'une seule valeur dans l'intervalle des valeurs possibles. Les variables de type composé permettent de stocker simultanément plusieurs valeurs (tableaux, enregistrements, etc.)

Comme tout langage de programmation, C offre des types de données scalaires de base. Nous verrons qu'il offre aussi des types composés et des manières de créer de nouveaux types.

Voici les trois **types prédéfinis** du langage C:

- les nombres entiers (`int`)
- les nombres décimaux (`float`)
- les caractères (`char`)

On appelle aussi ces trois types des **types de base** car tous les autres types que nous verrons sont, soit définis à partir de ces derniers, soit construits “par-dessus”.

On peut remarquer que le type **logique** ou **booléen** que l'on trouve dans d'autres langages n'existe pas en C. A la base, ce sont soit des valeurs entières soit directement des bits qui servent aux opérations logiques en C.

C permet par ailleurs de spécifier le nombre de bytes utilisés pour chaque type de données au moyen des mots clés `short` (court, nombres de petite taille) et `long` (entiers de grande taille ou réels de grande précision[1].

1. Voir IV.D Codage de l'information

Les notions d'instruction et d'expression sont très liées en C, en effet, on définit une instruction comme **une expression** suivie du **point virgule**.

Une expression est composée

- d'opérateurs, comme `+`, `=` ou `>`
- de constantes comme `'Z'` ou `2.618`
- de variables comme `monage`
- d'appels de fonctions comme `printf()`

Lorsque le programme s'exécute, les expressions sont **évaluées**, l'opérateur le **plus prioritaire** passant en premier; comme en algèbre, la multiplication se passe avant l'addition etc.

OPERATEURS ARITHMETIQUES

Precedence	Symbole	Operateur
1	`-`	Moins unaire
2	`*`	Multiplication
	`/`	Division
	`%`	Reste de la division entière (modulo)
3	`+`	Addition
	`-`	Soustraction

L'évaluation d'une expression, c'est-à-dire le calcul proprement dit du résultat, peut être dirigée à l'aide de **parenthèses**.

Pour ne pas surcharger par des parenthèses l'écriture des expressions, les opérateurs ont été classés par ordre de **précédence** ou priorité. Dans l'ordre de priorité nous avons l'opérateur moins unaire ('`-`'), qui change seulement le signe de l'expression, les opérateurs dits multiplicatifs ('`*`', '`/`', '`%`'), puis les opérateurs dits additifs ('`+`', '`-`'); le contenu des parenthèses reste évalué avant toute chose.

Ainsi, l'expression `-5 + 2.5 * 7 - 8.2`
sera evaluée comme `((-5) + (2.5 * 7) - 8.2)`

Lorsque les priorités des opérateurs ne conviennent plus pour évaluer une expression, il faut changer l'ordre d'évaluation en encadrant les expressions adéquates par des parenthèses.

`3 + 4 * 5` vaut `23` alors que `(3 + 4) * 5` vaut `35`

Dans le cas où les opérateurs sont de **précédence identique**, nous pouvons considérer, pour l'instant, que l'évaluation s'effectue de gauche à droite. Ceci n'est pas tout a fait exact et sera éclaici plus tard.

Les **opérandes** des opérateurs arithmétiques peuvent être de n'importe quel type de base mais des conversions implicites sont appliquées régulierement. La seule restriction concerne l'opérateur '`%`' pour lequel des opérandes de type `float` et `double` sont interdits. Il faut noter que le symbole de la division est unique. La distinction entre division entière et réelle est réalisee par le type des opérandes. Si les deux opérandes sont entiers le résultat sera entier; dans le cas contraire le résultat sera réel.

OPERATEURS RELATIONNELS

<	inférieur
<=	inférieur ou égal
>	supérieur
>=	supérieur ou égal
==	égalite
!=	inégalite

Ces opérateurs comparent deux expressions arithmétiques et retournent un résultat **vrai** ou **faux**.

En C, toute valeur **différente de zéro** est considérée comme **VRAIE** et toute valeur **égale à zéro** est **FAUSSE**.

Par ailleurs, les variable du type `char` peuvent aussi être comparées à l'aide de ces opérateurs, la comparaison s'effectuant sur le code ASCII correspondant.

Exemples

```
boucle >= 0            a == 67.5          c <= 'Z'
```

Les opérateurs relationnels sont **moins prioritaires** que les opérateurs arithmétiques. De cette façon, `10 + 4 > 5` sera convenablement évaluée à vrai.

CONTROLE

Opérateur d'affectation usuel

=	opérateur d'affectation

A l'inverse de Pascal, en C, le symbole d'affectation est un opérateur à part entière qui peut être utilisé au sein d'expressions. Son rôle consiste à **évaluer** l'expression du membre de droite et à **transférer** ce résultat dans l'expression du membre de gauche.

L'expression du membre de gauche doit évidemment être, pour l'instant, une **variable** et non une constante.

La **priorité** de cet opérateur est bien-sûr inférieure à celle des opérateurs déjà vus.

Attention on peut assez facilement se tromper et taper `=` alors que l'on désirait en fait utiliser l'opérateur de comparaison `==`. Le compilateur ne vous avertit pas et le programme peut se comporter de manière très étrange.

Exemples

```
n = 1                               c = 'Z'
a = 7 + 3 * nb_boucle               i = i + 1
```

Opérateurs contractés

Le programmeur C n'aime pas utiliser son clavier. On a donc créé pour lui une gamme d'opérateurs d'affectation contractés, qui lui permettent d'écrire rapidement des affectations très communes. Tous ceux qui ont déjà souffert en écrivant

```
un_identificateur_tres_long = un_identificateur_tres_long + 2;
```

savent de quoi je parle. La première solution est de raccourcir les noms des identificateurs, mais ce n'est pas toujours possible sans nuire à la clarté du code. C nous propose donc d'écrire

```
un_identificateur_tres_long += 2;
```

ce qui économise une bonne moité du travail.

Le résultat est bien entendu le même; on évalue d'abord la somme des membres de gauche et de droite, puis on affecte le résultat à l'expression de gauche.

L'affectation contractée est également possible pour la soustraction (`-=`), la multiplication (`*=`), la division (`/=`), le reste de la division entière (`%=`).

CONTROLE

C propose trois opérateurs logiques permettant d’écrire des expressions booléennes complexes.

Syntaxe des opérateurs binaires

expression_gauche **operateur** *expression_droite*

Symbole	*Opération*	*Valeur retournée*
`&&`	ET logique	vrai si le membre de gauche **et** le membre de droites sont vrais, faux sinon.
`||`	OU logique	vrai si le membre de gauche **ou** le membre de droite est vrai, faux si les deux sont faux.

Syntaxe pour la négation

! *expression*

Symbole	*Opération*	*Valeur retournée*
`!`	négation (unaire)	vrai si ce qui suit est faux, faux si ce quil suit est vrai.

Comme on en a l’habitude, faux est la valeur zéro, vrai est une valeur non nulle.

Ordre d’évaluation des expressions

Pour la négation, l’expression est évaluée en premier, ensuite on inverse sa valeur logique. Pour les deux autres, le programme évalue tout d’abord l’expression de gauche, ensuite celle de droite, seulement si le résultat n’était pas encore déductible.

.

C fournit un enseble très complexe et très riche de fonctions d'entrées/sorties. Parmi les plus utiles, on trouve les fonctions de la famille **`printf()`** (sorties formatées) et celles de la famille **`scanf()`** (entrées formatées).

Ces fonctions sont déclarées dans le fichier `stdio.h` qu'il faut inclure dans votre programme à l'aide de la commande

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

D'autres fonctions, très similaires à celles de TURBO PASCAL, permettent de faire de la mise en page, d'écrire à un endroit donné de l'écran, de l'effacer, etc. Ces fonctions sont utilisables en TURBO C, mais ne sont pas standard, donc ne devraient pas être utilisées dans un programme destiné à d'autres compilateurs C que TURBO.

Ecrire à l'écran, fonction printf()

Syntaxe

```
printf("Chaine de format" {,argument});
```

La **chaîne de format** contrôle comment les arguments seront convertis, formatés et affichés. Elle contient deux types d'informations: des **caractères ordinaires** qui sont simplement recopiés à l'écran et des caractères de **spécification de format**.

Les **spécifications de format** commencent toutes par un **caractère pourcent** ('`%`'), peuvent ensuite contenir une indication de **largeur** (minimale), de **précision** (maximale) et finalement un caractère code pour le **type de l'argument**. D'autres informations, facultatives et inutiles pour l'instant peuvent être ajoutées.

Voici les codes de type essentiels

- **`d`** pour les entiers
- **`f`** pour les nombres flottants sous forme décimale
- **`e`** pour les nombres flottants sous forme scientifique
- **`c`** pour les caractères
- **`s`** pour une chaîne de caractères
- **`%`** pour afficher le caractère '`%`'

Il doit y avoir au moins autant d'**arguments** qu'il en est prévu dans cette chaîne; sinon, le résutat n'est pas prévisible, mais sera probablement désastreux.

Le programme de la page suivante illustre différentes manières d'utiliser la fonction `printf()`.

LIBRARIES

```
#include <stdio.h>

#define NOM "Queloz Pierre-Antoine"
#define NATION "Suisse"

main() {

       float  heure = 10.30,
              accompte,
              frais;
       char dejeuner_au_lit;

       dejeuner_au_lit = 'O';
       accompte = 100;
       frais = 25.65;

       printf ("Auberge des chasseurs: fiche client\n\n");

       printf ("Nom du client : %25s\n", NOM);
       printf ("Nationalite : %25s\n", NATION);

       printf ("Numero de chambre: %10d\n", 33);
       printf ("Heure de reveil : %10.2f\n", heure);

       printf ("Sexe : %5c\n", 'M');
       printf ("Dejeuner au lit : %5c\n", dejeuner_au_lit);

       printf ("Accompte verse : %f Fr. Frais divers : %f Fr.\n",accompte, frais);

}
```

Le résultat produit:

```
Auberge des chasseurs: fiche client

Nom du client     :     Queloz Pierre-Antoine
Nationalite       :                    Suisse
Numero de chambre:          33
Heure de reveil   :      10.30
Sexe              :     M
Dejeuner au lit   :     O
Accompte verse : 100.000000 Fr. Frais divers : 25.650000 Fr.
```

LIBRARIES

Lire du clavier, fonction scanf()

La fonction `scanf()` permet au programme de demander des informations à l'utilisateur au moyen du clavier. La syntaxe est

```
scanf("Chaine de format" {,adresse});
```

Cette fonction lit une chaîne de caractères au clavier et stocke les données lues aux **adresses** indiquées.

La **chaîne de format** est très semblable à celle de `printf()`. Elle contrôle comment les valeurs seront lues et converties pour être mémorisées.

Pour chaque adresse de variable spécifiée, il faudra une **spécification de format** dans la chaîne.

Ces spécifications de format seront séparées par des **caractères séparateurs**. Les caractères séparateurs possibles sont l'espace ' ', le tabulateur '`\t`' et l'interligne '`\n`'. Lorsqu'un caractère séparateur est spécifié dans la chaîne de format, tous les caractères séparateurs rencontrés dans la chaîne d'entrée sont lus, jusqu'au prochain caractère qui ne soit pas un séparateur.

Les **spécifications de format** commencent par un **caractère pourcent** ('`%`'), comme pour la fonction `printf()`. On peut ensuite mettre une **spécification de largeur** (nombre maximal de caractères à lire) et un caractère donnant le **type de l'entrée**.

Les codes de types les plus importants sont

- **`d`** pour un entier
- **`e`** ou **`f`** pour un nombre flottant
- **`s`** pour une chaîne de caractères
- **`c`** pour un seul caractère[1]

Attention lors de la lecture d'un caractère, si le prochain caractère est un séparateur, il sera stocké quand-même.

Pour que `scanf()` puisse stocker les valeurs lues dans des variables, il ne lui suffit pas d'avoir le nom de la variable, il veut connaître l'**adresse** à laquelle la variable est stockée en mémoire centrale. Pour qu'il reçoive bien cette adresse, nous devons faire précéder chacun des noms de variables du caractère '**`&`**'. C'est un opérateur très utilisé en C qui retourne l'adresse de la variable qui suit.

LIBRARIES

1. On utilise plus volontiers d'autres fonctions plus simples pour lire des chaînes de caractères. Voir par exemple les fonction getc() et gets().

```
#include <stdio.h>

main() {
      int quantite;
      float prix_unitaire, total;

      printf ("Entrez le prix unitaire et la quantite SVP.\n");

      scanf ("%f %d", &prix_unitaire, &quantite);
      /* NOTER L'OPERATEUR '&' AVANT LES NOMS DE VARIABLES */

      total = quantite*prix_unitaire;

      printf ("\nMerci, \n\n%d articles a %.2f Fr. = %.2f Fr. au
total.\n",
            quantite, prix_unitaire, total);
}
```

Exemples d'exécution

```
Entrez le prix unitaire et la quantite SVP.
3.50 5

Merci,

5 articles a 3.50 Fr. = 17.50 Fr. au total.
```

Cet exemple montre que le type des données lues est important, `scanf()` ne se laisse pas berner par la valeur flottante.

```
Entrez le prix unitaire et la quantite SVP.
5 12.0566

Merci,

12 articles a 5.00 Fr. = 60.00 Fr. au total.
```

On peut mettre le séparateur que l'on veut dans la chaîne d'entrée, ici un '`\n`'

```
Entrez le prix unitaire et la quantite SVP.
3.50
4

Merci,

4 articles a 3.50 Fr. = 14.00 Fr. au total.
```

Evidemment, `scanf()` ne lit pas n'importe quoi, ici, il rejette notre chaîne de caractères, mais sans nous en avertir. Nous verrons plus tard comment tester le nombre de valeurs que `scanf()` a pu lire correctement.

```
Entrez le prix unitaire et la quantite SVP.
21.356 bof

Merci,

0 articles a 21.36 Fr. = 0.00 Fr. au total.
```

LIBRARIES

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Jusqu'à maintenant, les programmes que nous avons vus étaient tous **séquentiels**. Les instructions devaient s'exécuter l'une après l'autre selon l'ordre dans lequel nous les avons écrites, du haut vers le bas.

Mais les capacités de l'ordinateur à effectuer des **sauts** en avant et en arrière dans les instructions nous permettent d'imaginer d'autres types de déroulement. Le langage machine lui-même contient les instructions de saut, permettant de dire au processeur "va exécuter l'instruction qui se trouve à l'adresse mémoire X" ou "si le registre X contient la valeur Y alors va à l'instruction Z".

Ces possibilités sont bien entendu exploitées dans les langages de haut niveau qui offrent des structures **itératives** (boucles), des structures **conditionnelles** (tests) et des **ruptures** (sauts).

Les structures itératives

Elles permettent de spécifier des instructions qui seront exécutées **plusieurs fois** par le processeur. On parle aussi de boucles, si l'on suit le flot des instructions à l'aide d'un crayon sur un listing, on reviendra plusieurs fois au même endroit, dessinant ainsi des boucles. Les structures itératives de C sont

- `while` (tant que en anglais)
- `do ... while` (faire ... tant que)
- `for` (pour parcourir un intervalle)

Elles permettent de couvrir tous les cas possibles d'itérations.

Les structures conditionnelles

Elles permettent au programme de suivre plusieurs chemins différents, en fonction de conditions que l'on teste en cours d'exécution. A la manière d'un train sur des aiguillages, le programme choisira un chemin et n'exécutera qu'un sous-ensemble des instructions données. C'est par ce moyen de décider où il va qu'un programme peut faire preuve d'"intelligence".

Les structures conditionnelles de C sont

- `if ... else ...` (si ... sinon ...)
- `switch` (choix multiple)
- `l'opérateur ( ... )? ... : ...;` (alternative)

Les ruptures

- `goto` (aller à)
- `break` (arrêter)
- `continue` (continuer!)
- `return` (sortie de fonction)
- `exit` (fin du programme)

CONTROLE

Syntaxe

Sans alternative:

```
if ( expression )
      instruction ou bloc d'instructions
```

ou avec alternative:

```
if ( expression )
      instruction1 ou bloc d'instructions 1
else
      instruction2 ou bloc d'instructions 2
```

Fonctionnement

Si l'expression entre parenthèses est évaluée à **vrai** (valeur non nulle) alors l'instruction ou le bloc d'instructions qui suit le `if` est exécutée.

Si l'expression est évaluée à **faux** (valeur nulle) alors l'instruction se trouvant après le `else` est exécutée si on l'a spécifiée. Si on ne l'a pas spécifiée, le programme passe à l'instruction suivante directement.

Dans le cas où plusieurs tests se suivent, la clause `else` se rapporte toujours au `if` le plus proche.

Exemple

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

      int nbr;

      printf ("Entrez un nombre SVP ");
      scanf ("%d", &nbr);
      if (nbr > 0)

            if (nbr % 2 == 0)
                  printf ("C'est un nombre pair\n");
            else
                  printf ("C'est un nombre impair\n");

      else
            printf ("C'est un nombre negatif\n");
}
```

Résultats

```
Entrez un nombre SVP 12
C'est un nombre pair

Entrez un nombre SVP 13
C'est un nombre impair

Entrez un nombre SVP -12
C'est un nombre negatif
```

CONTROLE

Syntaxe

```
while ( expression )
      instruction ou bloc d'instructions
```

Fonctionnement

Lorsque le programme atteint l'instruction while, il évalue l'expression entre parenthèses.

Si le résultat est **vrai** (différent de zéro), alors il exécute l'instruction ou le bloc d'instructions qui suit puis il recommence. Il évalue une nouvelle fois l'expression, etc.

Si le résultat est **faux** (au premier, deuxième,... ou Xème passage) alors il n'exécute pas l'instruction et arrête de boucler, passant à l'instruction suivante.

Le test s'effectuant au début, il est très possible que l'instruction ne soit jamais exécutée. L'expression entre les parenthèses est obligatoire.

Par ailleurs, il est important que l'instruction ou le bloc puisse influencer sur l'évaluation de l'expression entre les parenthèses (par exemple en modifiant une variable de fin ou un compteur) sinon, le programme bouclera indéfiniment (CTRL-Break ou CTRL-C pour l'arrêter).

Exemple

```
#include <stdio.h>
main() {
      char uncar='A';

      while (uncar<='Z') {
            printf ("%c, ",uncar);
            uncar += 1;            /* code ASCII suivant */
      }
      printf ("\n");
}
```

Résultat

```
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,
```

CONTROLE

L'instruction `for` est une sorte de `while` plus complexe que l'on utilise généralement dans le cas de boucles où le nombre d'itérations est connu. Son usage n'est toutefois pas limité à ce seul cas comme dans d'autres langages.

Syntaxe

```
for ( initialisation; continuation; progression )
      instruction ou bloc d'instructions
```

initialisation, *continuation* et *progression* sont des expressions C quelconques.

Fonctionnement

Lorsque le programme arrive à une instruction `for`, l'expression *initialisation* est évaluée. Ensuite, l'expression *continuation* est évaluée, si elle est **vraie**, l'instruction ou le bloc est exécuté et l'expression *progression* est évaluée. On revient ensuite à l'évaluation de l'expression de continuation et on recommence jusqu'à ce qu'elle soit **fausse**.

Il est possible que les expressions *initialisation*, *continuation* ou *progression* soient vides; dans ce cas, il faut tout de même mettre le bon nombre de points-virgule. Il manquera une des étapes et le comportement sera différent. Si l'expression *continuation* est absente, la boucle ne s'arrêtera jamais, à moins qu'une **instruction de rupture** appropriée soit rencontrée.

Equivalence

Une instruction `for` peut être transformée en son équivalent `while`:

```
initialisation;
while (continuation) {
            instruction ou bloc d'instructions
            progression;
}
```

Exemple

Le programme suivant a exactement le même comportement que celui de la page précédente.

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

      for (uncar='A'; uncar<='Z'; uncar+=1)
            printf ("%c, ",uncar);

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

CONTROLE

SOUS-PROGRAMMES

Dans l'introduction, nous avons établi qu'un gros problème devait être décomposé en sous-problèmes de plus en plus simples pour être résolu. Cette manière se structurer les programmes nous conduit à la notion de sous-programmes.

Un sous-programme est un petit programme qui résout une partie du problème que le programme principal doit traiter. On peut ainsi isoler la résolution de **sous-problèmes** différents dans autant de sous-programmes. C'est un des mécanismes de base de la programmation structurée. S'il n'était pas possible de procéder de cette manière, nous ne pourrions pas résoudre de très gros problèmes. Pensez à des programmes comme la réservation de billets d'avion partout dans le monde ou l'analyse d'expériences de physique nucléaire qui peuvent comporter des millions de ligne...

Par ailleurs, dans un programme, on peut avoir besoin de faire plusieurs fois la même chose, avec de petites variations. Dans ce cas, il est plus économique d'appeler un sous-programme avec des paramètres différents, plutôt que d'écrire plusieurs fois les mêmes lignes de code. Cette notion de **réutilisabilité** est aussi très importante du point de vue du temps nécessaire à la réalisation ultérieure d'autres programmes qui peuvent se baser sur du travail déjà fait.

Le langage C permet de spécifier des sous-programme par le biais de **fonctions**. Il ne différencie pas procédures et fonctions comme le font d'autres langages. D'une manière tout à fait générale, un fonction peut effectuer un certain traitement dépendant de paramètres et retourner le résultat de ce traitement.

DEFINITION DE FONCTIONS

Les fonctions définies dans un programme C sont nommées par un **identificateur** construit de la même manière que les identificateurs de variables ou de constantes.

Elles peuvent **retourner** une valeur, il est donc nécessaire d'indiquer le type de cette valeur.

Par ailleurs, elles peuvent recevoir un certain nombre de **paramètres** ou **arguments** en entrée. Généralement, le comportement de la fonction est influencé par ses paramètres et il est aussi essentiel que le compilateur connaisse leurs types.

Les types de la valeur retournée et des arguments doivent faire partie des types connus au moment de la définition de la fonction.

Enfin viendra un **bloc d'instructions** définissant les variables locales et les actions que la fonction devra effectuer losqu'elle sera appelée.

Une des grandes différences entre la définition originale du langage et la norme ANSI plus récente est jutement la syntaxe d'une telle définition.

La syntaxe d'une définition de fonction selon la **spécification originale** du langage est

```
type_retourné identificateur_fonction ( liste_de_paramètres )
types_paramètres
bloc_d'instructions
```

Afin de simplifier la phase de compilation, les concepteurs du langage ont décidé de ne pas autoriser l'**imbrication** de fonctions. Elles sont donc toutes au même niveau dans un programme C.

Comme pour les variables, on distingue la déclaration d'une fonction de sa définition. La **définition** d'une fonction telle que nous venons de la voir contient toutes les informations nécessaires à l'exécution de la fonction: type de valeur retournée, type des paramètres, bloc d'instructions. Bien évidemment, ces informations sont suffisantes si elles sont connues au moment de l'appel de la fonction par une autre partie du programme.

Mais il n'est pas indispensable qu'une fonction soit ainsi définie au moment où elle est appelée, en effet, seuls les types des valeurs qui entrent et sortent de la fonction sont nécessaires au sous-programme appelant. Ainsi, il suffit de **déclarer** une fonction sans mettre le bloc d'instructions et le reste du programme saura qu'une fonction existe et comment il peut travailler avec. La définition standard du langage n'effectuant pas de contrôle sur les types des paramètres, il n'est de plus pas nécessaire de les indiquer dans une déclaration.

Syntaxe d'une déclaration de fonction selon la définition initiale du langage:

```
type_retourné identificateur_fonction ();
```

Toutes les lignes de programme suivant cette déclaration sauront que la fonction existe et pourront l'appeler. A vous ensuite de fournir une définition de la fonction, éventuellement même dans un autre fichier.

Exemple

```
/* Fonction retournant le cube de son paramètre */
float cube (un_nombre)
float un_nombre;
{
      float resultat;
      resultat = un_nombre * un_nombre * un_nombre;
      return resultat;
}
```

Cette définition indique au reste du programme que la fonction `cube` prend un nombre en virgule flottante et retourne un autre nombre en virgule flottante. Elle utilise une variable locale pour stocker le résultat avant de le retourner. La déclaration correspondante serait:

```
float cube();
```

CONTROLE

Observation fondamentale

De nombreuses erreurs sont possibles lors de l'utilisation de fonctions. En effet, les fonctions pouvant être réutilisées, il est possible que l'on se serve d'une fonction écrite des années plus tôt, ou par un autre programmeur. Par ailleurs, les distractions ne sont pas exclues et il est facile d'appeler une fonction avec un paramètre de type incorrect, par exemple un entier alors qu'il fallait un réel, ou alors d'intervertir deux paramètres si on ne connaît pas l'ordre par coeur, ou encore d'oublier un paramètre lors de l'appel.

Toutes ces erreurs peuvent être détectées automatiquement par le compilateur si il connaît le nombre de paramètres que la fonction désire et le type de chacun d'entre eux. Il peut ainsi vérifier la **concordance** entre les paramètres formels (donnés dans la déclaration de la fonction) et les paramètres effectifs (donnés à l'appel de la fonction) et signaler des erreurs.

La norme ANSI, ainsi que le langage C++ effectuent cette vérification de typage si les fonctions sont déclarées avant leur utilisation et que cette déclaration contient les types des paramètres formels.

Déclaration de fonction (prototypes)

La déclaration est différente de la déclaration «ancien style» car elle doit contenir, entre les parenthèses, l'énoncé des **types** de chacun des paramètres formels, séparés par des virgules. Si les types ne sont pas déclarés, la vérification ne pourra pas avoir lieu.

Le **nom** de chaque paramètre peut être ajouté pour la documentation mais n'est pas obligatoire et ne doit pas forcément correspondre avec le nom donné lors de la définition de la même fonction.

Exemple, déclaration de la fonction «cube»

```
float cube (float);           /* SANS nommer le paramètre */
float cube (float nombre);    /* déclaration équivalente */
```

Définitions

Dans les définitions de fonction, les types des paramètres sont donnés juste avant les noms des paramètres, entre les parenthèses et non plus à l'extérieur des parenthèses.

Les paires type-paramètre sont séparées par des virgules.

Pour chaque paramètre, il faut mettre explicitement le type, même si plusieurs paramètres sont du même type.

Exemple définition de la fonction «cube»

```
/* Fonction retournant le cube de son paramètre, définition ANSI */
float cube (float un_nombre)
{
      float resultat;
      resultat = un_nombre * un_nombre * un_nombre;
      return resultat;
}
```

APPEL

Dans le programme, partout où la définition de la fonction ou une déclaration équivalente est connue, il est possible d'appeler une fonction à l'intérieur d'une expression en écrivant son nom et ses paramètres effectifs entre parenthèses.

Lorsqu'une fonction est appelée, l'exécution du sous-programme appelant est suspendue et le contrôle passe à la première instruction de la fonction.

Les paramètres peuvent eux-mêmes être des expressions qui seront toutes évaluées **avant** d'entrer dans la fonction. L'ordre d'évaluation n'est pas précisé.

Rappelons-nous ce que nous avons fait lorsque nous voulions écrire à l'écran.

• L'instruction `#include <stdio.h>` servait à inclure une **déclaration** des fonctions d'entrées sorties afin que le reste du programme les connaisse.

• Nous avons utilisé la **fonction** `printf()` simplement en l'écrivant avec ses paramètres dans des expressions.

RETOUR

L'instruction

```
return expression;
```

n'importe où dans le bloc de la fonction a pour effet de sortir de la fonction et de retourner le résultat de l'expression au point d'appel de la fonction. Il peut y avoir plusieurs sorties de ce type dans une même fonction. Nous pouvons affecter la valeur retournée à une variable ou continuer à l'utiliser dans l'expression qui a déclenché l'appel de la fonction. Nous pouvons aussi négliger ce résultat complètement.

Il se peut aussi dans certains cas qu'aucune instruction `return` ne soit atteinte. Dans ce cas, la fonction se termine, le contrôle revient au programme appelant et la valeur de retour n'est pas définie. S'il est prévu qu'une fonction ne retourne pas de valeur (elle peut toutefois se terminer par l'instruction `return;`) elle doit être définie du type **`void`**. Dans ce cas, ce sera l'équivalent d'une procédure Pascal ou Fortran. Le mot réservé **`void`** est aussi utilisé pour indiquer l'abscence de paramètres.

• La fonction `printf()` **retourne** le nombre d'octets qu'elle a pu écrire, mais nous ne nous sommes jamais souciés de ce résultat; on peut toutefois imaginer des cas où ce résultat est important et qu'il serve pour la suite du traitement.

PARAMETRES

En C, le passage des paramètres entre le sous-programme appelant et la fonction appelée se fait par **valeur**, ce qui signifie que la fonction reçoit une copie de toutes les valeurs transmises. Elle peut modifier ces valeurs tant qu'elle veut, les valeurs originales du sous-programme appelant ne seront pas modifiées. Par ailleurs, même si les noms des variables sont les mêmes, il y a totale étanchéité entre l'intérieur et l'extérieur.

Exemple agricole

```
/*     Exemple d'utilisation d'une fonction pour calculer la surface d'un
       disque et le volume d'un cylindre ( un silo ) */

#include <stdio.h>

/* Declarations */
float surface(); /* Surface du silo de rayon r */
float volume();/* Volume du silo de rayon r et hauteur h */

/* Programme principal */
main () {
       float rayon, hauteur;
       printf ("Entrez le rayon du silo : ");
       scanf ("%f", &rayon);
       printf ("La surface du silo est %.2f\n", surface (rayon));
       printf ("Entrez la hauteur du silo : ");
       scanf ("%f", &hauteur);
       printf ("Le volume du silo est %.2f\n", volume (rayon, hauteur));
}

/* Definitions de fonctions */
float surface (r)/* r est le rayon du silo */
float r;
{
       return r*r*3.14;
}

float volume (r, h)/* r comme rayon et h comme hauteur */
float r, h;
{
       return h*surface (r);
}
```

On obtient comme trace d'exécution :

```
Entrez le rayon du silo : 4
La surface du silo est 50.24
Entrez la hauteur du silo : 20
Le volume du silo est 1004.80
```

Remarques

Le programme principal `main` est maintenant une fonction comme toutes les autres. Nous verrons plus tard quels sont ses paramètres (ils viennent de la ligne de commande).

Par ailleurs, la déclaration préalable des fonctions permet de mettre les définitions des fonctions dans un ordre quelconque, on préfère généralement mettre les choses les plus générales au début, ce qui évite de devoir lire les listings à l'envers comme en Pascal.

Noter comme le programme principal devient clair sans les formules de calcul qui sont déléguées au niveau inférieur; et comment la fonction `volume` utilise la fonction `surface` dans un but d'économie.

CONTROLE

Motivations

Nous connaissons déjà les types scalaires prédéfinis de C[1]. Imaginons que nous voulons manipuler les résultats d'examens de 20 personnes. Pour ce faire, nous devrions déclarer 20 variables de type entier ou flottant et ensuite, toutes les traiter séparément avec leur identificateur propre. Un tableau permet de simplifier grandement notre tâche. En effet, une seule variable de type tableau peut contenir **plusieurs informations du même type**, ici les notes de chaque élève et de les manipuler par un numéro.

Un tableau représente donc une "famille" de variables de même type à laquelle nous donnons un nom. C'est donc une structure de données de type **composé** car elle contient plusieurs valeurs, de **taille fixe** car l'occupation mémoire doit pouvoir être calculée à la compilation et **homogène** car toutes ces valeurs sont du même type de base.

Définition

```
type_de_base nom[expression_taille];
```

Les **crochets** font partie de la définition et ne signalent pas une partie optionnelle.

Une telle définition indique que nous allons utiliser un tableau appelé par `nom`, comportant `exprssion_taille` éléments de type `type_de_base`.

Le nombre d'éléments peut être n'importe quelle expression constante dont le résultat (`taille`) est un nombre entier non négatif. Cette expression ne peut toutefois pas contenir de variables ou d'appels de fonctions.

Le type de base doit être complet; tous les types que nous avons vus jusqu'ici (même les tableaux) sont complets.

Utilisation

Les éléments sont **numérotés** de `0` à `taille-1`. Chacun d'entre eux peut être adressé séparément et instantanément en indiquant

```
nom_du_tableau[numéro_d_élément]
```

Toutes les utilisations possibles d'une variable sont aussi possible avec un élément de tableau comme celui-ci. Le numéro d'élément doit être dans l'intervalle {0, 1, ..., taille-1}. Dans le cas contraire, vous ne serez pas avertis de l'erreur, mais le résultat pourrait bien être catastrophique, votre programme allant lire ou écrire (plantée garantie) n'importe où dans la mémoire centrale.

Ce numéro pourra être calculé par une **expression quelconque** (contenant par exemple des variables et des appels de fonctions).

1. Voir TYPES SCALAIRES à la page 30.

DONNEES

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Exemples

```
float notes[20];
```

Cette définition indique que nous allons utiliser un tableau de 20 nombres réels. Dans chacune de ces vingt positions, nous pouvons stocker un nombre, aller lire la valeur stockée, etc. L'instruction

```
notes[2] = 5;
```

aura pour effet de stocker la valeur 5 à la **troisième** position du tableau `notes`. Et

```
i = 7;
printf("%d", notes[i]);
```

imprimera la valeur stockée à la **huitième** position du tableau.

Déclaration

Un tableau peut être passé comme **paramètre** lors d'un appel de fonction. Dans ce cas, il doit être déclaré dans la liste de paramètres de la fonction.

Pour que notre fonction fonctionne sur des tableaux de tailles diverses, on aimerait bien ne pas avoir à fixer la taille du tableau. C'est possible car une déclaration de tableau à un indice peut être faite **sans spécifier la taille** de celui-ci. Cette opération est autorisée car une déclaration de tableau n'entraîne pas de réservation mémoire, contrairement à une définition.

Notre fonction pouvant à priori travailler sur des tableaux de taille quelconque, il faudra généralement prévoir de lui passer la taille du tableau en paramètre ou d'utiliser un marqueur de fin comme pour les chaînes de caractères.

Il faut encore savoir que contrairement aux variables, les tableaux ne sont pas passés par valeur mais **par référence**, sans doute pour des raisons d'efficacité, avec pour conséquence que toute modification d'un tableau à l'intérieur d'une fonction se répercute aussi à l'extérieur!

Exemple

```
float moyenne (donnees, ndonnees)  /* fonction de calcul de moyennes */
float donnees[];                   /* tableau de taille variable */
int ndonnees;                      /* nombre de donnees */
{
      int i;
      float somme = 0;

      for (i=0; i<ndonnees; i+=1)
            somme += donnees[i];
      return somme/ndonnees;
}
```

Limitation

Il n’y a **pas d’opérations globales** (affectation, comparaison) sur les tableaux ou sur des tranches de tableaux dans le langage, ces opérations étant en général réalisées facilement par des fonctions.

Cet exemple illustre l’utilisation des tableaux pour le traitement de données semblabes, ici, des consommations d’essence.

```
/* Exemple d'utilisation d'un tableau: Calcul d'une consommation moyenne */
#include <stdio.h>
main () {
	float km[5], litres[5], litres_aux_cent[5], moyenne=0;
	int cpt=0;

	printf ("Consommation moyenne sur les 5 derniers pleins:\n");

/* Stockage dans les tableaux */

	for (cpt = 0; cpt < 5; cpt += 1) {
		printf ("Nb de kilometres et de litres ? ");
		scanf ("%f %f", &km[cpt], &litres[cpt]);
	}

	printf ("Votre consommation pour les 5 derniers pleins:\n");

/* Calculs */

	for (cpt = 0; cpt < 5; cpt += 1) {
		litres_aux_cent[cpt] = litres[cpt]*100/km[cpt];
		printf ("%.2f\n", litres_aux_cent[cpt]);
		moyenne += litres_aux_cent[cpt]/5;
	}

	printf ("Consommation moyenne : %.2f\n", moyenne);
}
```

Exemple d’exécution:

```
Consommation moyenne sur les 5 derniers pleins.
Nb de kilometres et de litres ? 198 12.5
Nb de kilometres et de litres ? 160 7
Nb de kilometres et de litres ? 212 11.8
Nb de kilometres et de litres ? 84 6.3
Nb de kilometres et de litres ? 145 8
Votre consommation pour les 5 derniers pleins:
6.31
4.38
5.57
7.50
5.52
Consommation moyenne : 5.85
```

DONNEES

Il n'y a pas de type chaîne de caractères prédéfini en C. Par convention, les chaînes de caractères sont représentées à l'aide de **tableaux de caractères** à un indice. Malgré cela, la manipulation des chaînes de caractères est très souple et très efficace en C.

Le principe est le suivant, dans un tableau suffisamment grand, on stocke à la suite les différents caractères de la chaîne. A la fin de la chaîne, on rajoute un caractère supplémentaire qui fait office de **marqueur** de fin de chaîne. C'est le caractère dont le code ASCII est zéro, que l'on note '\0'. Attention toutefois à ne pas le confondre avec le caractère '0' dont le code ASCII est 48!

Les caractères stockés dans la suite du tableau seront tout simplement ignorés de toutes les fonctions, la convention voulant que le caractère '\0' soit le dernier de la chaîne. De plus, si une erreur entraîne l'abscence de ce marqueur, les fonctions de manipulation de chaînes de caractères fonctionneront anormalement, puisqu'elles ne le trouveront pas.

L'inconvénient de cette technique est qu'il faut toujours se souvenir qu'un tableau de N positions ne peut contenir au maximum qu'une chaîne de **longueur** N-1, le dernier élément du tableau devant forcément contenir le marqueur de fin de chaîne.

Exemples de définitions

```
char string[10];        /* définit une chaîne de 9 caractères */
char nom[20] = "Hitchcock";
char prenom[10] = "Alfred";
char phrase[]="Bonjour tout le monde!\n\n";
```

Remarques

Lorsque le compilateur rencontre une chaîne de caractères constante, il la convertit automatiquement en tableau et rajoute le caractère nul ('\0') à la fin.

Tout comme les tableaux, il n'y a pas d'opérations globales sur les chaînes (affectation, comparaison, concaténation) dans le langage, il y a par contre de nombreuses fonctions qui assurent ces tâches.

La seule exception est pour l'initialisation, une chaîne peut être initialisée par une chaîne constante lors de sa définition comme on peut le voir dans les exemples ci-dessus. Par ailleurs, le compilateur peut calculer la longueur d'une chaîne facilement. On peut donc omettre la longueur de la chaîne entre les crochets si on veut qu'elle soit ajustée automatiquement. Dans ce cas, il faut toutefois songer que si la chaîne doit s'allonger à un certain moment dans le programme, il risque d'y avoir un problème.

Les caractères d'interligne que l'on trouve dans le dernier exemple occupent deux caractères dans le texte du programme : \ et n. Dans le programme exécutable et en mémoire centrale, ils n'occupent par contre plus qu'un seul octet.

LECTURE

`scanf()`

Cette fonction permet de lire une chaîne de caractères en spécifiant un `%s` dans la chaîne de format.

Son mécanisme de découpage ne permet toutefois pas de taper des chaînes de caractères contenant des blancs ou des tabulations. La lecture se termine aussitôt qu'un caractère séparateur est rencontré.

Par ailleurs, il ne faut pas utiliser l'opérateur adresse `&` lors de la lecture d'une chaîne de caractères, les tableaux étant de toutes façons passés par référence.

On lui préfère généralement la fonction

`gets(`*`tableau_car`*`)`

Cette fonction lit une chaîne depuis le flux d'entrée standard `stdin` et la place dans le tableau de caractères passé en paramètre.

La lecture se termine à la réception d'un caractère d'interligne (touche return). Le '`\n`' n'est par inséré dans la chaîne, il est remplacé par un terminateur '`\0`'.

Contrairement à `scanf()`, elle permet l'entrée de chaînes contenant des espaces et des tabulations.

ECRITURE

`printf()`

Cette fonction permet d'afficher une chaîne de caractères en spécifiant un `%s` dans la chaîne de format.

Elle ne rajoute pas d'interligne toute seule, mais on peut en mettre un dans la chaîne de format si on le désire.

`puts(`*`chaine`*`)`

Cette fonction envoie la chaîne de carctères spécifiée dans le flux de sortie standard `stdout`. Elle ajoute un caractère d'interligne à la fin.

LIBRARIES

Nous avons vu comment nous pouvions introduire des boucles (itérations) et des embranchements (`if`) dans un programme C. Une instruction de rupture permet de faire un **saut** dans la suite des instructions.

Saut vers une étiquette: goto

Syntaxe

```
goto etiquette;
```

Cette instruction permet d'effectuer un saut n'importe où dans le programme, en spécifiant une **étiquette** (label en anglais) **au début** d'une instruction. L'identificateur de l'étiquette doit bien entendu être construit selon les règles habituelles.

Exemple

```
sauteici: ...;
...
if (une condition quelconque) goto sauteici;
```

Remarque

Il est toujours possible de se passer d'une telle instruction. C'est un des principes de la programmation structurée. Je vous invite donc à ne l'utiliser qu'en cas d'absolue nécessité et surtout pas sur de "longues distances"...

Sortir d'une boucle: break

Cette instruction est utilisée pour sortir d'une boucle ou d'un `switch`[1]. Dès qu'elle est rencontrée, le contrôle passe à l'instruction suivant directement la boucle.

Exemple

```
#include <stdio.h>
main() {
      int i=0;
      while(1) {        /* boucle sans fin */
            printf ("%d ",i);
            i += 1;
            if (i > 10) break;
      }
}
```

Résultat

```
0 1 2 3 4 5 6 7 8 9 10
```

Passer à la suite: continue

Cette instruction semblable au `break` permet de passer directement à la prochaine itération, mais sans interrompre la boucle. Pour l'instruction `for`, l'expression de continuation est quand-même évaluée.

1. Voir L'INSTRUCTION SWITCH à la page 60.

CONTROLE

Exemple

```
#include <stdio.h>
main() {
      int i=0;
      for (i=0; i<10; i += 1) {
            if (i==5) continue;
            printf ("%d ",i);
      }
}
```

Résultat

```
0 1 2 3 4 6 7 8 9
```

Sortir d’une fonction: return

Cette instruction que nous connaissons déjà[1] est une instruction de rupture, car elle permet à tout moment de sortir de n’importe quel endroit d’une fonction pour retourner à l’endroit d’où la fonction fut appelée.

Syntaxe

`return [`*`expression`*`];`

Sortir du programme: fonction exit()

Cette fonction permet de terminer le programme n’importe où. On se retrouvera donc soit dans TURBO, soit à l’invite du système d’exploitation, selon l’environnement duquel notre programme a été lancé.

Cette fonction accepte un paramètre entier dont la valeur est retournée au système d’exploitation qui a donc une indication sur la manière dont s’est terminé le programme. Traditionellement, la valeur 0 indique que tout s’est bien passé et un nombre entier positif signale une erreur.

1. Voir RETOUR à la page 46.

Nous connaissons déjà deux instructions qui permettent d'écrire une boucle dans un programme C: le `while` et le `for`. Ces deux types de boucles fonctionnent approximativement sur le même principe. A chaque itération, une expression est évaluée **avant** d'exécuter les instructions. Tant que l'expression est vraie, la boucle continue.

Dans certains cas, les instructions doivent être exécutées au moins une fois. Il est donc utile que le test se fasse **à la fin**. Il existe donc une troisième construction qui permet de répéter un ensemble d'instructions tant qu'une expression est vraie, avec le test à la fin.

Syntaxe

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

Mécanisme

Tout d'abord, l'instruction ou le bloc est exécuté, ensuite, l'expression entre parenthèses est évaluée. Si le résultat est **vrai** (différent de `0` et de `'\0'`), alors une nouvelle itération est commencée. Lorsque le résultat de l'évaluation de l'expression est **faux** (égal à `0` ou `'\0'`), la boucle se termine.

Exemple

```
#include <stdio.h>
main() {
      char c = 'a';
      do {
            printf ("%c ",c);
            c += 1;
      } while (c<='z');
      printf ("\n");
}
```

Résultat

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Remarques

Comme pour le while, l'expression entre parenthèses est obligatoire. Par ailleurs, il est important de se souvenir que comme le test se trouve à la fin, l'instruction ou le bloc d'instructions est toujours exécuté au moins une fois.

CONTROLE

Syntaxe

expression_test **?** *expr_si_oui* **:** *expr_si_non ;*

Fonctionnement

La première expression (*expression_test*) commence par être évaluée.

Si le résultat est **vrai**, alors la seconde expression (*expr_si_oui*) est évaluée à son tour et la troisième est sautée.

Si le résultat de l'évaluation de la première expression est **faux**, alors la seconde expression est sautée et c'est la troisième (*expr_si_non*) qui est évaluée.

Dans les deux cas, le résultat de la seconde évaluation peut être utilisé comme opérande d'un autre opérateur.

Cet opérateur remplace souvent avantageusement un test ou un appel de fonction.

Exemples

```
/* Calculer le maximum de deux nombres */

/* Version "IF" */
if (a>b) max=a;
else max=b;

/* Version "?:" équivalente
max = a>b ? a : b;
```

L'expression précédente est correcte car la **priorité** de `?:` est inférieure à celle de `>` et supérieure à celle de `=`. En cas de doute, vous pouvez bien-sûr rajouter des parenthèses.

```
/* Alternative : on dit si un nombre est positif ou negatif */
#include <stdio.h>
main () {
      int n;
      printf ("Entrer un nombre entier, positif ou negatif: ");
      scanf ("%d", &n);
      printf ("Le nombre est %s.\n", n>=0 ? "positif" : "negatif");
}
```

Résultats

```
Entrer un nombre entier, positif ou negatif: 90
Le nombre est positif.

Entrer un nombre entier, positif ou negatif: -23
Le nombre est negatif.
```

CONTROLE

Toutes les variables que nous avons utilisées jusqu'à présent étaient déclarées au début d'un bloc d'instructions. Ces blocs d'instructions faisaient généralement partie d'une définition de fonction. La gestion des variables en C ne s'arrête pas là. En plus d'un type, à chaque variable sont associées une **visibilité** et une **classe d'allocation**.

La visiblité définit les portions du programme qui auront accès à cette variable, et pourront par conséquent y faire référence ou modifier sa valeur. La visibilité d'une variable est fixée par sa défintition et dépend pricipalement de l'emplacement de cette définition.

Variables globales

Les variables globales sont déclarées en-dehors de toute fonction. Elles sont visibles de l'intérieur de chaque fonction du programme définie ultérieurement.

Les variables globales sont allouées statiquement à la compilation et leur durée de vie correspond au temps d'exécution du programme. Elles se trouvent généralement dans le segment DATA du programme exécutable.

Par défaut, les variables globales sont initialisées à la valeur 0. Il est possible de spécifier une expression d'initialisation pour chaque définition de variable globale. L'expression doit toutefois pouvoir être calculée à la compilation, ce qui implique qu'elle ne doit contenir ni appels de fonctions ni références à d'autres variables. On parle dans ce cas également d'"expressions constantes".

Variables locales

Les variables locales d'un bloc sont définies au début de ce bloc. Elles ne sont visibles qu'à l'intérieur de ce bloc et des éventuels sous-blocs à l'intérieur.

Si plusieurs variables ont le même nom, elles ne sont pas confondues, la règle veut que ce la dernière définition (la plus proche) soit considérée.

Illustration, les accolades montrent les domaines de visibilité.

```
int a;

void funct () {
      char b;
      ...
      {
            int a;        }  d est une variable locale,
            char d;          a masque la visibilite
            ...              de la var. globale
      }                      du même nom
}                            (b est une var. locale)

main () {
      int b;              }  autre var. locale
      ...                    pas de conflit
}                            avec b de funct

(a variable globale : de int a; jusqu'à la fin de main)
```

Variables globales

Nous avons vu que les variables globales sont définies dans tout le programme, ce n'est pas tout à fait exact car un grand programme peut être composé de plusieurs fichiers. Les variables globales peuvent être limitées à un seul de ces fichiers si on fait précéder leur définition du mot réservé `static`. En général, il est important de limiter au strict minimum la visibilité d'une variable, afin d'éviter qu'une partie du programme qui n'est pas censée la connaître puisse modifier sa valeur par erreur.

Exemples

```
static int nbpers = 12;
static char car_fin = 'q';
```

Ces deux variables ne seront accessibles que des fonctions qui se trouvent définies dans le même fichier.

D'autre part, on peut vouloir **utiliser** une variable globale définie dans un autre fichier. Dans ce cas, il est nécessaire de précéder la déclaration de la variable par le mot réservé **`extern`** pour indiquer au compilateur qu'il ne s'agit pas d'une nouvelle définition de variable. Bien entendu, il doit y avoir une définition correspondante dans un autre fichier, laquelle ne doit pas être précédée du mot `static`.

Exemples

```
extern int larg_fenetre;
extern char titre[];
```

Ces déclarations indiquent que la variable entière `larg_fenetre` et la chaine de caractères `titre` sont définies dans un autre fichier.

Fonctions

On peut remarquer la grande similitude entre les définitions de variables globales et de fonctions. En effet, les fonctions que nous définissons dans un programme C sont aussi des objets globaux, visibles de toutes les fonctions suivant leur définition ou leur déclaration.

Les fonctions d'un programme C peuvent aussi être organisées dans plusieurs fichiers et les mots réservés `static` et `extern` peuvent être utilisés de la même manière, bien que `extern` ne soit pas obligatoire.

Exemple

```
static int une_fonction_statique () {
      ...
}                       /* n'est visible que dans ce fichier */

extern int utile ();    /* permet l'utilisation de utile() plus loin
                        dans le programme */

int utile ();           /* est aussi possible */
```

DONNEES

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Variables locales

Trois classes d'allocation sont possibles pour les variables locales, désignées par les mots réservés **`auto`** (automatique) qui est la classe d'allocation par défaut, **`static`** et **`register`**. On spécifie la classe d'une variable locale en précédant sa définition du mot réservé correspondant.

Variables locales de classe automatique

La mémoire nécessaire au stockage de la variable est allouée dynamiquement sur la pile en cours d'exécution.

Leur valeur initiale est indéterminée et leur durée de vie limitée au temps d'exécution du bloc qui les contient. Il est toutefois possible de les initialiser lors de leur définition. Cette initialisation se fait en cours d'exécution et l'expression d'initialisation peut donc être quelconque.

Exemple

```
{
        char a='a'+ecart; /* est de classe automatique par défaut */
        ...
        /* fin de l'existence de a */
}
```

Variables locales de classe statique

Les variables locales de classe statiques sont parfois utiles si on désire entrer plusieurs fois dans un bloc et qu'à chaque fois on désire retrouver les variables commes elles étaient quand l'exécution précédente du bloc s'est terminée. C'est une bonne alternative à l'utilisation d'une variable globale, car une telle variable n'est visible qu'à l'intérieur du bloc et ne peut pas être modifiée par inadvertance.

L'espace mémoire nécessaire au stockage des variables de classe statique est réservé lors de la compilation, leur valeur initiale par défaut est zéro, une expression constante (comme pour les variables globales) permet toutefois de spécifier une autre valeur initiale.

Bien que la visiblité des variables de cette classe soit limitée au bloc dans lequel elle est définie et à ses éventuels sous-blocs, sa durée de vie est le temps d'exécution du programme entier, temps durant lequel elle gardera sa valeur.

Variables locales de classe register

Les variables locales de cette classe seront dans la mesure du possible stockées dans un registre interne du processeur, afin d'optimiser le temps d'exécution de parties critiques du programme. On ne peut pas faire de supposition quant à leur durée de vie. Mais si le processeur ne dispose pas de suffisamment de registres, elles seront de classe automatique. Certains compilateurs optimisant très efficacement l'utilisation des registres, il est parfois nuisible aux performances d'un programme de déclarer trop de variables de cette classe.

Nous avons vu comment utiliser l'instruction `if () ... else ...` pour permettre à un programme d'exécuter différentes instructions selon une condition. Cette instruction ne permet malheureusement que deux alternatives: soit l'expression s'évalue à vrai et la première instruction (ou le bloc) est exécutée, soit l'expression est fausse et c'est l'instruction (ou le bloc) qui suit le `else` qui est exécutée.

L'instruction `switch` permet de surmonter ce problème par un "aiguillage multiple", dans lequel on peut mettre un ensemble d'instructions pour plusieurs valeurs de l'expression possibles.

Syntaxe

```
switch (expression_entiere) {
      case expr_const_entiere :
            {instructions}
            [break;]
      case expr_const_entiere :
            {instructions}
            [break;]
      etc.
      [default : {instructions}]
}
```

Le déroulement de l'exécution est le suivant

- *`expression_entiere`* est évaluée. Le résultat doit être de type entier, caractère ou énuméré.
- Le résultat est ensuite comparé à chacune des *`expr_const_entiere`* et dès que la valeur est la même, toutes les instructions jusqu'à un `break` ou la fin du `switch` sont exécutées. L'expression qui se trouve après chaque `case` ne peut pas contenir de variable ni d'appel de fonction et doit être de type entier, char ou enuméré.
- Si aucune des expressions ne correspond, la branche `default` facultative est activée.

Remarques

Comme toutes les instructions qui suivent un `case` sont exécutées si celui-ci est activé, il est souvent nécessaire de sortir avec un `break`, mais ce n'est pas obligatoire. Par ailleurs, ce mécanisme permet de faire des "ou" et de faire le même traitement pour différentes valeurs. Par exemple:

```
case valeur1:
case valeur2:
case valeur3:
      faire un ensemble de choses
      break;
```

Permet de faire la même chose pour les trois valeurs *`valeur1`*, *`valeur2`* et *`valeur3`*.

CONTROLE

Cet exemple montre comment réaliser un petit menu.

```
#include <stdio.h>
main() {
      char reponse=' ';
      printf ("Options\n\n");
      printf ("l. Lire fichier\n");
      printf ("s. Sauver donnees\n");
      printf ("n. Commencer nouveau travail\n");
      printf ("q. Quitter\n");
      printf ("\nQue choisissez-vous? ");
      while (reponse!='q' && reponse!='Q') {
            reponse=getchar();
            getchar();  /* pour consommer le \r */
            switch (reponse) {
                  case 'l':
                  case 'L':
                        printf ("ok, fichier lu\n");
                        break;
                  case 's':
                  case 'S':
                        printf ("ok, donnees sauvees\n");
                        break;
                  case 'n':
                  case 'N':
                        printf ("ok, on recommence\n");
                        break;
                  case 'q':
                  case 'Q':
                        printf ("bye...\n");
                        break;
                  default:
                        printf ("Tapez une des lettres proposees!\n");
            }
      }
}
```

Exemple d’exécution

```
Options

l. Lire fichier
s. Sauver donnees
n. Commencer nouveau travail
q. Quitter

Que choisissez-vous? s
ok, donnees sauvees
S
ok, donnees sauvees
l
ok, fichier lu
b
Tapez une des lettres proposees!
q
bye...
```

CONTROLE

Très fréquemment, un programme doit incrémenter (ajouter 1) ou décrémenter (soustraire 1) une variable, par exemple pour parcourir un tableau.

Généralement, il existe même une instruction dans le langage machine qui permet de le faire. C fournit donc deux opérateurs à cet effet.

`++`	opérateur d'incrémentation
`--`	opérateur de décrémentation

L'évaluation de ces opérateurs peut se faire de deux manières selon qu'ils sont placés avant le nom de la variable ou après.

Opérateur avant la variable
L'incrémentation ou la décrémentation se fait **avant** que la valeur de la variable ne puisse être utilisée par le reste de l'expression.

Opérateur après la variable
L'incrémentation ou la décrémentation se fait **après** que la valeur de la variable ait été utilisée par le reste de l'expression.

Exemples

```
int a = 0, b;
char c = 'u', d;
float f=14.5;

/* Expressions simples */
c++;   /* après cette instruction, c contient 'v' */
a--;   /* après celle-la, la variable a contient -1 */

/* Expressions composées, opérateur après la variable */
b = a++;     /* on commence par copier a dans b et ensuite, on l'incrémente */
             /* après cette instruction, b vaut -1 et a vaut 0 */
d = c++;     /* d vaut 'v' et c vaut 'w' */

/* Expressions composées, opérateur avant la variable */
a = --b;     /* b est décrémenté (-> -2) et copié dans a (-> -2 aussi) */
d = ++c;     /* c est incrémenté (-> 'x') et copié dans d (-> 'x' aussi) */

/* imprimer l'alphabet avec une boucle do...while */
char le_car = 'a';
do
      printf ("%c ", le_car++);/* incrémentation en fin */
while (le_car <= 'z');
```

L'extrait de programme précédent fonctionne car il utilise d'abord la valeur de la variable `le_car` pour l'imprimer et l'incrémente ensuite.

```
f++;  /* les nombres de type float peuvent aussi etre incrementes */
```

CONTROLE

Le langage C permet au programmeur de définir de nouveaux types, si ceux qu'il a à disposition ne suffisent pas pour représenter les données que son programme doit manipuler.

Il arrive fréquemment que la valeur d'une variable fasse partie d'un ensemble fini de symboles. On parle dans ce cas de **types énumérés** car il est possible de décrire complètement la liste de toutes les valeurs possibles.

Le type **booleen** en est un exemple car les seules valeurs que peuvent prendre les variables de ce type sont **vrai** (ou true en anglais) et **faux** (false en anglais et en PASCAL, FORTRAN,...).

Dans la définition originale du langage, les types énumérés n'existaient pas, ils étaient gérés directement par le programmeur à l'aide de constantes nommées (`#define`) et de nombres entiers. Par la suite, ils furent rajoutés pour des raisons de lisibilité.

Définition d'un type énuméré

```
enum nom_du_type {
      {ident_symbole [= expr_const_entière],}
} {ident_var [= valeur_initiale],};
```

- Les accolades en gras sont obligatoires, les autres signifient 0 ou plusieurs.
- ***nom_du_type*** est un identificateur facultatif du type, il permet de réutiliser la définition ailleurs dans le programme, par exemple si des objets (variables ou fonctions) déclarés plus loin doivent être du même type.
- ***ident_symbole*** est un identificateur construit selon les règles habituelles, il représente l'une des valeurs possibles pour les variables de ce type. L'identificateur ne doit pas être le même que celui d'un autre symbole ou d'une variable visibles lors de sa définition. Tous les symboles sont listés dans l'ordre, séparés par des virgules.
- ***expr_const_entière*** est une expression constante et de type entier facultative qui permet de donner un valeur bien précise à l'un des symboles. En l'abscence d'une telle expression, le premier élément énuméré prend la valeur entière `0` et chaque nouvel élément ajoute 1 à la valeur de son prédécesseur. Il est même possible de "revenir en arrière" et de donner une valeur plus basse ou égale à un élément déjà défini.
- ***ident_var*** définit une nouvelle variable du type énuméré nouvellement défini. Plusieurs variables peuvent être ainsi définies, en les séparant par des virgules. Chacune d'entre-elles peut être initialisée par une expression *`valeur_initiale`* facultative.
- un **point-virgule** termine les définitions.

Exemple

```
enum oiseau {
      pinson, rouge_gorge = 4, moineau, autruche = 400,
      canari = 4, martin_pecheur, albatros, poulet
} titi = canari, fifi;

enum oiseau cocotte=poulet;
```

Actuellement, le compilateur transforme toujours les variables et les constantes de type énuméré en **entiers**. Ainsi, toutes les opérations possibles sur un entier sont possibles avec une variable de type énuméré.

• Il n'y a pas de contrainte sur la valeur que prend une variable de type énuméré, ainsi, il est possible d'affecter directement une valeur entière à une telle variable, même si elle ne correspond à aucune valeur du type.

• Une valeur d'un type énuméré peut être affectée à n'importe quelle variable de type entier.

• Les valeurs d'un type énuméré étant stockées sous forme d'entiers, il n'est pas possible de les imprimer sous forme symbolique. Il n'y a pas de conversion automatique entre les symboles définis et des chaînes de caractères imprimables. Il est donc parfois nécessaire de faire des fonctions de conversion pour passer d'une variable de type énuméré à la chaîne de caractères correspondante et vice-versa.

Par rapport à l'"ancienne mode" où l'on faisait des `#define`, la seule restriction est la visibilité: si un type énuméré est défini au début d'un bloc, il ne sera connu qu'à l'intérieur de ce même bloc.

Exemples

```
enum oiseau birdy = 112;      /*    affectation d'une valeur qui n'est
                                    pas dans l'intervalle */
int meuble;

meuble = pinson + autruche;/* n'a pas de sens mais est autorisé */
```

Conclusion

Les types énumérés ne sont pas contraignants du tout, et bien que pratiques, ils doivent être utilisés avec maintes précauctions afin d'éviter des quantités astronomiques d'erreurs logiques.

Utilisation d’un type couleur «intelligent» utilisant un bit différent pour chaque couleur fondamentale, ce qui permet les additions et les soustractions de couleurs.

```
/* type enumere couleurs */
enum couleur {
      noir, /* 0 = abscence de couleur */
      rouge = 1, vert = 2, bleu = 4, /* couleurs fondamentales */
      jaune = rouge + vert,/* valeur 3 */
      cyan  = vert + bleu,/* valeur 6 */
      magenta = rouge + bleu,/* valeur 5 */
      blanc = rouge + vert + bleu/* valeur 7 */
};

/* fonction d'impression, recoit une couleur et imprime sa valeur */
void imprime ();

main () {
      enum couleur col;

      col = noir;
      printf ("%d ", col); imprime (col);
      col += rouge;
      printf ("%d ", col); imprime (col);
      col += cyan;
      printf ("%d ", col); imprime (col);
      col -= bleu;
      printf ("%d ", col); imprime (col);
}

void imprime (coul)
enum couleur coul;
{
      switch (coul) {
            case noir:  puts ("noir"); return;
            case rouge: puts ("rouge"); return;
            case vert:  puts ("vert"); return;
            case bleu:  puts ("bleu"); return;
            case jaune: puts ("jaune"); return;
            case cyan:  puts ("cyan"); return;
            case magenta:puts ("magenta"); return;
            case blanc: puts ("blanc"); return;
      }
}
```

Résultat

```
0 noir
1 rouge
7 blanc
3 jaune
```

DONNEES

Une structure est un agrégat de plusieurs objets de **types différents**; elle permet donc de regrouper dans une même variable plusieurs informations complémentaires de types hétérogènes en donnant un nom à chacune d'entre elles, ce qui en fait un outil très puissant de structuration de données.

Chacune des données composant une structure est appelée **un champ**. Tous ces champs peuvent être de types quelconques (tableaux, autres structures, énumérations, etc.). La seule contrainte est que ces types soient **complets**[1], autrement dit que l'occupation mémoire de chaque élément soit calculable à la compilation.

Chacun des champs possède un identificateur qui permet d'**accéder directement** à l'information qu'il contient.

Exemple

Pour une personne, on peut regrouper dans une seule variable

- son nom (chaîne de caractères),
- son âge (entier),
- son sexe (type énumére),
- son numéro AVS (tableau de 4 chiffres entiers), ...

Définition de structures

```
struct ident_struct {
      {type_champ ident_champ {, ident_champ};}
} {ident_var [= structure],} ;
```

Les accolades en gras sont obligatoires, les autres indiquent une répétition.

Exemple

```
enum sexes {feminin, masculin};

struct personnes {
      char nom[20], prenom[20];
      int age;
      enum sexes sexe;
      int numero_avs[4];
} moi = { "Holmes", "Serlock", 44, masculin, { 31, 49, 211, 39 } };
```

Ces quelques lignes définissent trois choses: un type énuméré `sexes`, une structure `personnes` et une variable `moi` du type `struct personnes`.

1. Comme pour les types de base des tableaux.

Dans la défintion

```
struct ident_struct {
      {type_champ ident_champ {, ident_champ};}
} {ident_var [= structure],} ;
```

• ***ident_struct*** est un identificateur facultatif qui permet de définir d'autres objets du même type ailleurs dans le programme (réutiliser la définition). Il n'y a pas de **collisions** entre identificateurs de variables ou de fonctions, et de structures ou d'énumérations[1]. Ainsi, au même niveau (niveau "principal" ou dans le même bloc), il est possible de déclarer une variable `x` et une `struct x` sans qu'il n'y ait de problèmes. Par contre, il ne doit jamais y avoir une variable et une fonction, ni une structure et une énumération de mêmes noms au même niveau. Par un mécanisme analogue à celui qui permet de définir des variables locales dans un bloc, il est aussi possible de définir des "structures locales" ou des "énumérations locales" qui ne sont connues que dans le bloc et masquent la définition du niveau supérieur.

• ***type_champ*** doit être un type complet (par exemple int ou char) connu au moment de la définition.

• ***ident_champ*** est un identificateur construit selon les règles habituelles. Plusieurs champs de même type peuvent être déclarés en les séparant par des virgules. Pour déclarer des champs d'un autre type, il est nécessaire de séparer les définitions par un point-virgule et de mettre un nouvel identificateur *type_champ*. A l'intérieur d'une structure, tous les noms des champs doivent être différents. Par contre, les identificateurs peuvent être les mêmes que ceux d'autres objets du programme sans qu'il y ait de collisions.

• ***ident_var*** est un identificateur de variable qui permet de définir une nouvelle variable du type `struct ident_struct`. Plusieurs définitions peuvent être spécifiées, à condition qu'elles soient séparées par des virgules. Il faut au moins une définition de variable si la structure n'est pas nommée, car sinon, on ne définit rien!

• la dernière partie ***= structure*** est facultative et permet d'affecter une valeur initiale à la variable nouvellement définie. Elle est consituée d'une paire d'accolades, entre lesquelles apparaissent une liste d'expressions constantes, séparées par des virgules, correspondant à la valeur de chacun des champs. Il est bon de savoir que certains compilateurs n'autorisent pas une telle affectation, que d'autres le permettent seulement-pour les variables globales et statiques mais pas pour des variables locales et que la norme ANSI semble avoir réglé le problème en acceptant toujours une telle initialisation[2].

• pour finir, on met un point-virgule, comme d'habitude.

1. Voir TYPES ENUMERES à la page 63.
2. Voir INITIALISATIONS DE STRUCTURES à la page 111.

DONNEES

Exemples de définitions possibles (ou impossibles entre /* */)

```
int a;

/* char a(); n'est pas autorise */
char b();

struct a {
      char a;
      float b;
};

/* enum a {x, y, z}; n'est pas autorise a cause du struct a */
/* enum b {a, y, z}; n'est pas autorise a cause du int a */
enum b {x, y, z};

/* float x; n'est pas autorise a cause du symbole z */

main() {
      int a; /* masque l'autre variable a */

      enum a {x, y, z};
      /*  struct  a  b;  impossible  car  enum  a  masque  le  definition  de
struct a */

      enum b truc;

      struct b {
            int a;
            float b;
      } machin;         /* masque la definition de enum b */

      /* enum b bidule; n'est plus possible */

      struct b b;/* Declaration de variable, possible*/
}
```

DONNEES

Accès à un champ

Les champs d'une structure peuvent être accédés directement par leur nom au moyen d'une notation "pointée". Il suffit en effet de mettre le nom de la variable de type structure, un point (opérateur!) et le nom du champ pour accéder à son contenu, y stocker une valeur, etc.

```
nom_variable.nom_champ
```

L'expression est une variable du type du champ, que l'on peut utiliser partout où l'on peut utiliser une variable habituellement.

Affectation

L'affectation globale au moyen de l'opérateur = entre deux variables de même structure est maintenant autorisée par la plupart des compilateurs, elle est même préconisée par la norme ANSI.

Comparaison

La comparaison (==) n'est par contre pas admise généralement. Il est nécessaire de tester chacun des champs l'un après l'autre. En général, une petite fonction fait très bien l'affaire.

Structures et fonctions

La plupart des compilateurs actuels, dont TURBO C autorisent le passage de structures par valeur et le retour de structures par les fonctions. Attention toutefois à la portabilité de ces opérations. D'une manière générale, un passage par référence évite des problèmes si un programme doit "voyager" dans d'autres environnements.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
#include <stdio.h>

enum propulsion {pedales, moteur, reacteur};

struct vehicule {
      char nom[20];
      int longueur, poids;
      enum propulsion mode;
} velo = {"Euroteam", 2, 5, pedales};

void imprime();
struct vehicule modif();

main () {
      static struct vehicule voiture = {
                              "Toyota",
                              5, 1500,
                              moteur };
      struct vehicule avion;
      imprime(velo);
      imprime(modif(velo,reacteur));
      strcpy (avion.nom, "Jumbo");
      avion.longueur = 60;
      avion.poids = 450000;
      avion.mode = reacteur;
      imprime(avion);
}

void imprime (s)                    /* reçoit une structure */
struct vehicule s;
{
      printf ("Nom: %s\n", s.nom);
      printf ("Longueur: %d, poide: %d\n", s.longueur, s.poids);
      printf ("Mode de propulsion: %d", s.mode);
      printf ("\n");
}

struct vehicule modif (p, mode)     /* retourne une structure */
struct vehicule p;
enum propulsion mode;
{
      p.mode = mode;
      return p;
}
```

Résultat

```
Nom: Euroteam
Longueur: 2, poide: 5
Mode de propulsion: 0
Nom: Euroteam
Longueur: 2, poide: 5
Mode de propulsion: 2
Nom: Jumbo
Longueur: 60, poide: 450000
Mode de propulsion: 2
```

DONNEES

Il existe un grand nombre de fonctions prédéfinies de manipulation de chaînes de caractères. Elles sont déclarées dans le fichier `string.h` qu'il est nécessaire d'inclure dans votre programme s'il utilise l'une d'entre-elles.

Copier une chaîne dans une autre

Les chaînes de caractères en langage C étant des tableaux, il n'est pas possible d'utiliser l'opérateur d'affectation '`=`' habituel pour les copier, il faut utiliser la fonction

```
strcpy (chaine_destination, chaine_source);
```

Cette fonction copie la chaine *chaine_source* dans la chaîne *chaine_destination*, y compris le marqueur de fin '`\0`'.

Exemple

```
char une_chaine[20] = ""; /* la chaine est vide */
strcpy (une_chaine, "Hola!"); /* une_chaine contient "Hola!" */
```

Mesurer la longueur d'une chaîne

Se fait au moyen de la fonction

```
int strlen (chaine);
```

qui retourne la longueur de la chaîne passée en paramètre,

Exemple

```
/* avec les définitions de l'exemple précédent */

int longueur;
longueur = strlen (une_chaine);     /* longueur prend la valeur 5 */
```

Comparer deux chaînes

On utilise la fonction

```
int strcmp (chaine_1, chaine_2);
```

qui retourne les valeurs suivantes, selon le contenu des deux chaînes

- `0` si les deux chaînes sont identiques.
- un nombre **négatif** si la première est alphabétiquement inférieure (avant) la seconde.
- un nombre **positif** si la première chaîne est supérieure à la seconde.

Exemple

```
/* toujours avec les définitions précédentes */
int compare;
compare = strcmp (une_chaine, "Hello!");
/* compare prend une valeur positive */
compare = strcmp (une_chaine, "Hola!");
/* compare prend la valeur 0 */
```

LIBRARIES

La fonction **`strcmpi()`** fonctionne comme la fonction `strcmp()` mais ne fait pas la distinction entre majuscules et minuscules.

Mettre deux chaînes bout à bout

`strcat (`*`chaine1`*`, `*`chaine2`*`)`

Ajoute une copie de *`chaine2`* à la fin de *`chaine1`*.

Conversions majuscules-minuscules

`strlwr (`*`chaine`*`)`

Convertit toutes les majuscules en minuscules sans changer les autres caractères.

`strupr (`*`chaine`*`)`

Convertit toutes les minuscules en majuscules sans changer les autres caractères.

Par ailleurs, les fonctions **`toupper()`** et **`tolower()`** retournent le caractère qu'on leur fournit comme paramètre en majuscules ou en minuscules respectivement.

Recherches

`strchr (`*`chaine`*`, `*`caract`*`)`

Retourne un pointeur[1] sur la première occurence du caractère dans la chaîne ou NULL (=0) si le caractère ne figure pas dans la chaîne.

`strstr (`*`chaine1`*`, `*`chaine2`*`)`

Retourne un pointeur sur la première occurence de *`chaine2`* dans *`chaine1`* ou NULL.

Attention

Bien que leur usage soit généralisé, ces fonctions ne se trouvent pas dans tous les environnements de programmation.

LIBRARIES

1. Dans bien des cas, un pointeur est équivalent à une sous-chaîne bien que ce ne soit pas la même chose puisque toute modification faite par le biais du pointeur se fait dans la chaîne originale. La relation entre pointeurs et tableaux est expliquée plus en détail à la page 113.

Les mémoires à semi-conducteurs utilisées dans les ordinateurs modernes sont capables de stocker à une certaine adresse un ensemble de données élémentaires appelées **bits**.

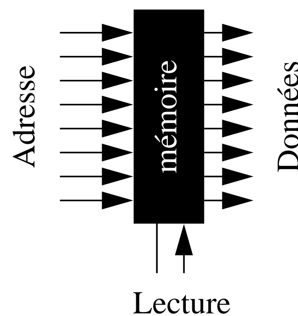
Chacun de ces bits est stocké physiquement par une tension électrique dans quelques transistors. Deux niveaux de tension différents permettent de stocker une information **binaire**, généralement représentée par un 0 ou un 1 à laquelle nous pouvons associer n'importe quelle signification.

Pour des raisons pratiques, nous ne pouvons en général pas accéder directement à chaque bit de la mémoire, mais seulement à un ensemble de bits ou **mot**. La taille des mots mémoire, ou nombre de bits est une caractéristique importante d'un ordinateur. Sur la plupart des micro-ordinateurs modernes, cette taille est de 16 ou 32 bits.

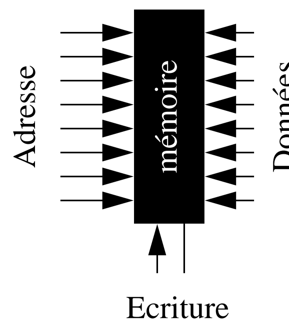
Grâce à un mécanisme d'**adressage**, le processeur central peut lire ou écrire un mot à une certaine adresse mémoire, elle aussi codée en binaire, et ainsi stocker et retrouver les données qu'il doit manipuler. Heureusement pour nous, ces mécanismes sont automatisés. L'adresse avec laquelle nous désirons travailler est elle-aussi codée en binaire. Le nombre de **bits d'adresse** est lui-aussi une caractéristique importante de l'ordinateur. Généralement, les PC ont un adressage sur 20 ou 32 bits.

Le fait que les adresses sont codées de manière similaire aux nombres est très important pour le programmeur C car il implique que les adresses peuvent être manipulées comme les données sous formes de variables ou de constantes. Une variable contenant une adresse mémoire est appelée **pointeur**.

Lecture d'un mot mémoire



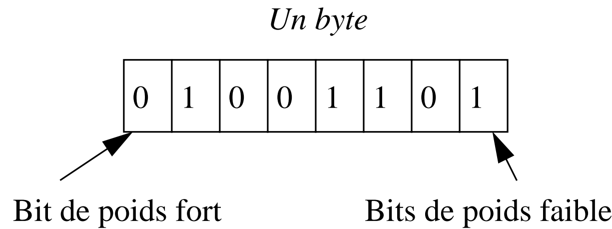
Ecriture d'un mot mémoire



Cette figure illustre les deux mécanismes de base d'accès à un mot mémoire de huit bits dans le cas d'un adressage sur huit bits. Chacune des petites flèches représente **un seul** bit, donc un 0 ou un 1 qui entre ou qui sort de la puce sous forme de tension électrique. C'est l'extrême simplicité de ce mécanisme qui nous oblige à établir des **codes**, afin de décrire exactement quelle doit être la représentation binaire de toutes les informations que nous désirons manipuler à l'aide de l'ordinateur.

Pour des raisons pratiques, des groupes de **huit bits** sont très souvent utilisés dans les

ordinateurs modernes. On parle dans ce cas de **bytes** ou d'**octets** en français.



Dans la suite du texte un indice après une donnée indique dans quelle base elle est exprimée. Les données sans indice sont exprimées en base 10.

Lorsqu'on exprime une donnée sous sa forme la plus élémentaire, donc en binaire, il est nécessaire de donner la valeur de chaque bit. Par exemple 0110100110011101_2 pour un mot mémoire de 16 bits. Il est par conséquent beaucoup plus pratique de faire usage d'autres codes, dans des bases plus élevées où on exprime la valeur de plusieurs bits par un seul chiffre ou une seule lettre.
Les codes les plus utilisés sont l'octal (base 8), le décimal (base 10) et l'hexadécimal (base 16). On donne à chacun des bits un **poids** dépendant de sa position, le bit le plus à **droite** obtient le poids le plus faible (1), et le bit le plus à **gauche** le poids le plus fort (2^{k-1} si le mot comporte k bits). La valeur résultante est la somme des poids de chaque bit à 1.

Exemple

0	1	1	0	1	1	0	0	bits du mot binaire
128	64	32	16	8	4	2	1	poids correspondants
	64 + 32 + 8 + 4 = 108							valeur décimale correspondante

Si la conversion de binaire en décimal n'est pas directe, les conversions en octal et en hexadécimal sont immédiates par regroupements de bits. La convention veut que l'on utilise les premières lettres de l'alphabet latin pour représenter les valeur au-delà de 9_{10}.

Exemple

6 | 9 | 9 | D	valeur en hexdécimal par regroupements de 4 bits
0 1 1 0 1 0 0 1 1 0 0 1 1 1 0 1	valeur binaire
0 | 6 | 4 | 6 | 3 | 5	valeur en octal par regroupements de 3 bits

Pour le codage des entiers, il est nécessaire de tenir compte de deux caractéristiques: la taille et le signe.

TAILLE DU NOMBRE

Elle détermine le nombre d'octets occupés en mémoire par une variable, les trois possibilités sont

- **`short int`** (entier court) qui correspond à une taille de 2 octets (16 bits) sur toutes les machines
- **`int`** qui est stocké sur un mot machine entier (dépend de la machine)
- **`long int`** qui correspond à une taille de 4 octets (32 bits) sur toutes les machines.

Pour les cas `short` et `long`, le mot réservé `int` n'est pas obligatoire (le type entier pris par défaut). Par ailleurs, la longueur des variables de type `int` dépend de la machine et peut correspondre soit à un `short`, soit à un `long`. Il faut donc être prudent lorsqu'un programme doit être utilisé sur plusieurs machines différentes.

Exemples

`short int a;`	et	`short a;`	sont des déclarations équivalentes.
`long int b;`	et	`long b;`	également.

DONNEES

SIGNE

Le codage est également différent si le nombre est signé (positif ou négatif) ou pas (seulement positif). Les nombres signés sont codés en complément à deux, les nombres non-signés sont codés en binaire "pur". Comme le nombre de données que l'on peut coder sur un k bits est constant (2^k), les nombres non-signés peuvent être jusqu'à deux fois plus grands que les nombre signés.

Exemple

Le tableau suivant montre comment seraient codés des entiers sur huit bits.

Valeur binaire	*Entier signé*		*Entier non-signé*	
00000000	0		0	
...				
01111111	$2^{(k-1)}-1$	(= 127)	$2^{(k-1)}-1$	(= 127)
10000000	$-2^{(k-1)}$	(= -128)	$2^{(k-1)}$	(= 128)
...				
11111111	-1		2^k-1	(= 255)

Par défaut, les entiers sont signés en C mais pour être vraiment sûr que le nombre est signé, on peut précéder sa définition du mot réservé **`signed`**. Pour spécifier un nombre non signé, on ajoute le mot réservé **`unsigned`** avant sa définition.

Exemples

```
signed short i;     est équivalent à     short i;
unsigned long c;
unsigned int xy;
```

ECRITURE DES CONSTANTES

Nous savons déjà comment mettre une constante entière dans un programme. Jusqu'à présent, nous nous sommes contentés d'écrire des valeurs entières dans les expressions.

Exemple

```
a = 3*b-14;
```

Mais nous pouvons aussi écrire des constantes en octal et hexadécimal dans un programme et également spécifier la taille de cette constante et si elle est signée ou pas.

Les constantes en base 8 (**octal**) s'écrivent avec un **`0`** comme premier chiffre. De plus, le nombre ne doit être constitué que de chiffres entre 0 et 7.

Les constantes **hexadécimales** s'écrivent avec un préfixe **`0x`** au début. Elles ne doivent être constituées que de chiffres entre 0 et 9 et de lettre A ou a (10) à F ou f (15).

Le suffixe **`L`** ou **`l`** peut être ajouté si la constante est "longue". Dans les autres cas, on considère que c'est une valeur de type "`short`".

Les suffixes **`U`** ou **`u`** peuvent être ajoutés si la constante est non signée. En l'abscence de ces suffixes, elle est considérée comme signée. Ces suffixes ne sont pas acceptés par tous les compilateurs. Mais lorsqu'ils le sont, on peut aussi les combiner avec un suffixe **`L`** ou **`l`**.

Exemples

```
a=1234;      /* constante signée, short et décimale */
a=01234;     /* constante signée, short et octale */
a=0815;      /* interdit à cause du 8 */
a=0x12F;     /* constante signée, short et hexadécimale */
a=123u;      /* constante non-signée, short et décimale */
a=123l;      /* constante signée, longue et décimale */
a=0x23ul;    /* constante non-signée, longue et hexadécimale */
```

DONNEES

Afin de prendre le minimum de place, les caractères sont codés sur **un seul octet**, qui permet 256 combinaisons de bits différentes. Ainsi, chaque caractère (y compris les caractères spéciaux comme ESC, CR, LF, etc.) possède une combinaison de huit bits pour le représenter.

C'est la correspondance entre les combinaisons de bits est les entiers binaires correspondants que l'on nomme code ASCII (American Standard Code for Information Interchange). Ainsi, un caractère correspond toujours à un nombre entier entre 0 et 255, qui correspond à un nombre binaire entre 00000000_2 et 11111111_2.

Exemples

Le caractère 'a' correspond à l'octet 01100001_2 qui correspond à l'entier 97.
Le caractère 'A' correspond à l'octet 01000001_2 qui correspond à l'entier 65.
Le caractère '2' correspond à l'octet 00110010 qui correspond à l'entier 50.

On comprend donc pourquoi les caractères peuvent être manipulés comme des entiers dans les programmes C. En fait, ce sont de petits entiers.

Tout comme les entiers, les caractères peuvent être signés ou pas, par défaut, ils sont signés, mais on peut également utiliser les mots réservés **`signed`** et **`unsigned`** dans les définitions de variables de type `char`.

Pour les variables de type **`unsigned char`**, l'intervalle des valeurs possibles est 0..255.
Pour les variables de type **`signed char`** ou **`char`** tout court, l'intervalle est -128..+127.

CONSTANTES

Les constantes de type caractères peuvent être entrée sous forme d'entiers (codes ASCII) décimaux, octal (octaux) ou hexadécimaux.

La forme la plus agréable est celle qui utilise les apostrophes. Dans cette forme, il est possible de donner des caractères spéciaux en préfixant un code octal par **`\0`** ou un code hexadécimal par **`\x`**. Enfin, un ensemble de caractères spéciaux courants ont une forme préfixée par un `\` que nous avons déjà rencontrée.

```
\b          backspace
\f          form feed
\n          new line
\r          carriage return
\t          tabulateur
\\          backslash
\'          apostrophe
\"          guillemets
\0          NULL
```

Exemples

```
char a = 65;
a = 'A';
a = '\x41';
a = '\0101';
```

DONNEES

Les variables du type "flottant" sont justement codées sous la forme dite "virgule flottante". Tout nombre codé de cette manière est de la forme

`signe * mantisse * base` $^{\texttt{exposant}}$

Les bits utilisés pour coder le nombre sont répartis en trois groupes:

- bit de signe, disant si le nombre est positif ou négatif
- bits de mantisse normalisée, contenant les chiffres significatifs du nombre
- bits d'exposant, contenant la "position de la virgule"

La base utilisée est généralement 2 pour le codage interne.

Il existe deux tailles de nombres flottants en C:

• **`float`** qui est utilisé pour des nombres codés sur 4 bytes (32 bits), ce qui permet trois intervalles approximatifs de

$-10^{+38}..-10^{-38}$, 0.0 et $+10^{-38}..+10^{+38}$.

• **`long float`** ou **`double`** qui correspondent à des nombres codés sur 8 bytes (64 bits) et offrent des valeurs possibles dans les trois intervalles

$-10^{+308}..-10^{-308}$, 0.0 et $+10^{-308}..+10^{+308}$.

CONSTANTES

Les constantes réelles s'écrivent

`signe mantisse` `E` *`signe_exposant exposant`*

Exemple

`-3.25e+15` représente le nombre `-3.25 * 10`^{15}

Si la constante est positive, on ne doit pas nécessairement mettre le **`+`**, de même si l'exposant est positif.
La mantisse est composée d'une partie entière, d'un point et d'une partie fractionnaire.
Selon son humeur du moment, on peut aussi mettre un **`e`** au lieu de `E`.
L'exposant est un nombre entier exprimé en base 10.

On peut omettre soit la partie entière, soit la partie fractionnaire (mais pas les deux), de plus, on peut omettre, dans le cas où il n'y a pas de partie fractionnaire soit le point décimal soit le 'e' et l'exposant (mais pas les deux).

Par ailleurs, les constantes doivent être considérées comme étant en double précision. Il est toutefois possible d'ajouter un suffixe **`f`** ou **`F`** pour que la constante soit considérée comme étant en simple précision et **`l`** ou **`L`** pour la double précision (facultatif).

La fonction `printf()` permet d'imprimer des nombres codés de manières particulières.

La lettre **`l`** avant un champ permet d'indiquer que la variable est de type long (entiers et flottants). La lettre **`h`** pour une valeur "short".

Les codes de types suivants peuvent également être utilisés:

- **`o`** pour indiquer que le nombre est **non-signé** et doit être affiché en **octal**.
- **`x,X`** pour afficher un nombre **non-signé** en **hexadécimal**.
- **`u`** pour les nombres entiers non-signés en **décimal**.

Exemple

```
#include <stdio.h>

main() {
      char c = 'A';
      unsigned short d = 65;
      printf ("%c %d %o %x \n", c, c, c, c);
      printf ("%c %hd %lo %lx \n", d, d, d, d);
}
```

Résultat

```
A 65 101 41
A 65 101 41
```

Important!

Ces modifications sont aussi valables pour la fonction de lecture usuelle **`scanf()`**.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

LIBRARIES

Cet opérateur est très utile si on veut travaille à un niveau assez bas car il permet de calculer la taille d'une donnée ou d'un type.

Syntaxe

```
sizeof ( type )
sizeof expression
```

Il retourne l'**occupation mémoire en octets** de type ou de l'expression donnée.

Noter que l'expression n'est pas évaluée lors de l'exécution.

On l'utilise ainsi si on travaille avec des types de données **dynamiques** dont la taille n'est pas fixée une fois pour toutes.

Exemple

```
#include <stdio.h>

struct essai {
      long a;
      char w, x, y, z;
      double d;
};

char tableau[20];

main () {
      struct essai variable;

      printf ("Taille de la structure \"essai\" : %d\n",
            sizeof (struct essai));
      printf ("Taille du champ \"d\" de la variable \"variable\" :
%d\n",
            sizeof variable.d);
      printf ("Taille du tableau de caracteres : %d\n",
            sizeof tableau);
      printf ("Taille d'une constante de type float : %d\n",
            sizeof 3.14e0f);
}
```

Résultat

```
Taille de la structure "essai" : 16
Taille du champ "d" de la variable "variable" : 8
Taille du tableau de caracteres : 20
Taille d'une constante de type float : 4
```

CONTROLE

Le langage C met à disposition du programmeur une gamme complète d'opérateurs de manipulations de bits. Ces opérateurs prennent des **arguments de type "entier"** (`char`, `int`, `short`, `long` dans leurs versions `signed` et `unsigned`).

Ces opérateurs sont

`&`	et logique bit à bit
`|`	ou logique
`^`	ou exclusif
`~`	complémentation à 1 (négation)
`>>`	décalage à droite
`<<`	décalage à gauche

Les opérateurs `&`, `|`, `^` prennent **deux arguments** d'un type listé ci-dessus et retournent une valeur du même type.

Les opérateurs `>>` et `<<` prennent **un opérade de type "entier"** et un **nombre**, le nombre indique le nombre de positions à décaler vers la gauche ou vers la droite.
Lors de l'utilisation de ces opérateurs avec des valeurs signées, le bit de signe reste en place, cela afin d'implémenter correctement les divisions et multiplications par une puissance de deux. On dit aussi que ce sont des décalages "**arithmétiques**".

L'opérateur `~` prend **un argument** de type "entier" et retourne une valeur du même type, avec tous les bits inversés.

Exemples

Soient les valeurs a = $1001\ 1000_2$ et b = $0111\ 1011_2$.

`a & b`	vaut	$0001\ 1000_2$			
`a | b`	vaut	$1111\ 1011_2$			
`a^b`	vaut	$1110\ 0011_2$			
`~a`	vaut	$0110\ 0111_2$ et	`~b`	vaut	$1000\ 0100_2$
`a >> 1`	vaut	$0100\ 1100_2$			
`b << 4`	vaut	$1011\ 0000_2$			

L'opérateur `&` est très utile pour faire un masque, si on ne s'intéresse qu'à certains bits d'un mot.

L'opérateur `<<` est très utile pour aller tester un certain bit: l'expression

```
mot & 1 << n
```

a la valeur VRAI (différent de 0) si le n$^{\text{ème}}$ bit (en partant de la droite en en commençant à compter à 0) du mot est 1 et FAUX (= 0) si le n$^{\text{ème}}$ bit du mot est 0.

Lors de l'évaluation d'une expression, des conversions implicites sont automatiquement appliquées sur les opérandes.

CALCULS

Lors de l'évaluation d'une expression mettant en jeu des opérateurs arithmétiques, les conversions suivantes sont réalisées automatiquement.

- une opérande `float` est convertie en `double`
- une opérande `char` ou `short` est convertie en `int`
- une opérande `unsigned char` ou `unsigned short` est convertie en `unsigned int`

Après ces conversions, si deux opérandes d'un même opérateur sont différentes, elles sont converties dans le type le plus "large".

- si une opérande est de type `double`, l'autre sera convertie en `double`
- si une opérande est `float`, l'autre est convertie en `float`
- si une opérande est `unsigned long`, l'autre sera convertie en `unsigned long`
- si une opérande est `long`, l'autre sera convertie en `long`
- si une opérande est `unsigned`, l'autre sera convertie en `unsigned`
- sinon, on a affaire à un calcul entre `int`

AFFECTATION

Lors d'une affectation, il y a conversion automatique du résultat de l'évaluation de l'expression à droite dans le type du membre de gauche.

APPEL DE FONCTION

Lors de l'appel d'une fonction, les paramètres effectifs sont convertis selon les règles suivantes.

- une valeur `float` est convertie en `double`
- une valeur `char` ou `short` est convertie en `int`
- une valeur `unsigned char` ou `unsigned short` est convertie en `unsigned int`

La norme ANSI définit certaines variantes pour les conversions lors du passage de paramètres.

DONNEES

On peut forcer explicitement la conversion d’un type vers un autre, il existe une opérateur dit "cast" qui permet de le faire.

Syntaxe

```
( type voulu ) expression
```

On précède simplement l’expression à convertir d’une paire de parenthèses entre lesquelles on indique dans quel type le résultat de l’évaluation de l’expression doit être converti.

Exemple

```
/* exemple de l'utilite des conversions explicites */
#include <stdio.h>

main() {

      printf ("%d\n", 2 * 3); /* une expression de type entier */
      printf ("%f\n", 2 * 3); /* probleme, l’expression est "int" */
      printf ("%f\n", (float) 2*3); /* conversion explicite */
      printf ("%d\n", 2.1*3); /* probleme, l'expression est "float" */
      printf ("%d\n", (int) 2.1*3); /* conversion explicite */
      printf ("%d\n", (int) (2.1*6)); /* arrondi, vers le bas */
}
```


Résultat

Voir les commentaires dans le texte du programme.

```
6
0.000000
6.000000
1075393331
6
12
```

PROBLEMES

Aucun problème n’est rencontré lors d’une conversion d’un certain type vers un type "plus large" (espace mémoire de représentation plus grand). En revanche, certains problèmes apparaissent lors de conversions vers un type "plus étroit". Ces problèmes peuvent provenir de valeurs trop élevées pour le type récepteur, auquel cas rien n’est prévu et le résultat est n’importe quoi. Des problèmes peuvent aussi découler des conventions de représentation interne des informations. Dans ce dernier cas, les valeurs sont généralement arrondies à la valeur représentable la plus proche.

Le langage C autorise la définition de nouveaux identificateurs de types grâce au mot réservé **`typedef`**.

Une défintion de type n'est pas aussi contraignante que dans d'autres langages, car les types définis par un `typedef` sont toujours compatibles avec le type de base et entre eux.

Elle a par contre l'avantage d'être **locale**, avec les mêmes règles de visiblité que les autres constructions de types comme les structures ou les énumérations.

Syntaxe

```
typedef type_existant identificateur;
```

Exception

Pour définir un type `tab` qui soit un tableau de `nb_elements` éléments d'un certain type de base, il faut écrire

```
typedef type_de_base tab[nb_elements];
```

Ici, les crochets ne signifient pas que la partie encadrée est optionnelle.

Utilité

On utilise parfois une définition de type pour raccourcir l'écriture des types structurés ou énumérés. Par exemple

```
typedef enum jours_de_la_semaine e_jour;
```

Permet d'écrire `e_jour` au lieu de `enum jours_de_la_semaine`. Ce qui est un gain assez appréciable si on doit le mettre plusieurs fois dans le programme.

On utilise aussi les définitions de type pour des raisons de lisibilité du programme, cet aspect documentaire étant essentiel dans les grands programmes.

La troisième utilité des définitions de type est la réutilisabilité des fonctions. En effet, si on définit que les paramètres d'une fonction sont de type `char`, cette fonction ne pourra être utilisée qu'avec des caractères. Si par contre, on met un type que l'on définit soi-même, on pourra réutiliser la fonction un autre fois juste en changeant la définition du type et sans rien changer à notre fonction si elle a été conçue soigneusement. Si on arrive à programmer de grandes portions de programme dans cette optique, un gain appréciable de temps peut être réalisé lorqu'une application semblable doit être écrite.

L'exemple suivant montre comment on peut définir un type booléen en C et comment on peut travailler avec (remarquer que les opérateurs logiques habituels fonctionnent correctement).

Noter par ailleurs que l'on a défini `FAUX` comme première valeur de l'énumération, ce qui lui donne automatiquement la valeur `0` conventionnelle tandis que le constante `VRAI` aura la valeur `1`.

```
#include <stdio.h>

typedef enum {FAUX, VRAI} BOOL;

void printb(); /* impression d'une valeur de type BOOL */

main () {
      BOOL v1 = VRAI, v2 = FAUX;
      printb (v1);
      printb (v2);
      printb (v1 && v2);
      printb (v1 || v2);
      printb (! v1);
}

void printb (v)
BOOL v;
{
      printf ("Le resultat est %s.\n", v ? "VRAI" : "FAUX");
}
```

Résultat

```
Le resultat est VRAI.
Le resultat est FAUX.
Le resultat est FAUX.
Le resultat est VRAI.
Le resultat est FAUX.
```

DONNEES

Un pointeur est une **variable** suceptible de contenir l'**adresse en mémoire** d'un objet du programme.

Lorsqu'un pointeur contient l'adresse d'un objet, on dit qu'il **pointe** sur cet objet. Et on parle d'objet **pointé** ou **référencé** par le pointeur.

Comme toutes les variables, les pointeurs ont un **identificateur** qui permet de les nommer lors de leur utilisation.

Comme les objets référencés peuvent être de types très divers, et pour éviter des confusions lors des manipulations de pointeurs, on doit définir pour chaque pointeur quel est le **type** des objets qu'il peut référencer.

Cette contrainte est un facteur de sécurité important, vu la puissance des opérations autorisées sur les pointeurs. Elle aide par ailleurs le programmeur à s'y retrouver lorsqu'il travaille avec de nombreuses variables de ce type.

Syntaxe d'une définition de pointeur

```
type_objet_pointé *identificateur [= adresse];
```

Le type de l'objet pointé peut être n'importe quel type déjà défini, l'identificateur est construit selon les règles habituelles, la partie entre crochets (valeur initiale) est optionnelle et permet d'affecter au pointeur l'adresse d'un objet du bon type immédiatement après sa définition.

Lorsqu'on ne spécifie pas de valeur initiale, la règle est la même que pour les autres variables, s'il s'agit d'une variable globale, le pointeur est initialisé à zéro, s'il s'agit d'une variable locale, l'adresse référencée est indéfinie.

Il convient d'être particulièrement prudent lors de l'utilisation de pointeurs, les pointeurs mal initialisés peuvent être sources d'interminables plantées de la machine, bien- sûr indétectables par le compilateur.

Il est donc très vivement conseillé de **toujours initialiser tous les pointeurs le plus tôt possible**.

Si la donnée à référencer n'est pas connue, ou que le pointeur ne "pointe sur rien", il est possible de lui affecter la constante prédéfinie `NULL` (qui vaut d'ailleurs `0`, ce qui est très utile pour les tests).

Il faut enfin toujours se souvenir qu'un pointeur est une variable en soi et qu'une définition de pointeur n'entraîne pas la définition (réservation mémoire) d'une variable du type correspondant.

Exemples

```
int *p1;    /* pointeur sur une variable de type entier */

float *p2;  /* pointeur sur une variable de type float */

struct exemple {        /* structure quelconque comme exemple */
      int a, b, c;
};

struct exemple *p3;     /* pointeur sur une variable de type "struct
                        exemple" */

typedef char ligne[80];       /* les variables de type ligne sont des
                              tableaux de 80 caractères */

ligne *p4;                    /* p4 est un pointeur sur un tableau de
                              80 caractères */

double **p5;      /* pointeur sur un pointeur sur un double! */
```

DONNEES

OPERATEUR ADRESSE &

L'opérateur `&`, placé devant une variable (ou une fonction) fournit l'adresse de cette variable en mémoire centrale.

Syntaxe

`&`*`variable`*

Le type résultant est "pointeur sur le type de la variable".

Il n'est donc pas nécessaire en C de se soucier des adresses effectives des variables, puisqu'on dispose d'un opérateur pour les calculer.

Il n'est pas possible de déplacer une variable à l'aide de cet opérateur, il sert uniquement à calculer la position de l'objet en mémoire.

Par ailleurs, la variable ne doit pas être de classe `register`.

Exemple

```
main () {
      int   a,    /* variable de type entier */
            *p;   /* pointeur vers un entier */

      p = &a;     /* p pointe sur a car on lui affecte
                  l'adresse de a*/
}
```

CONTROLE

OPERATEUR INDIRECTION *

L'opérateur `*`, placé avant une expression de type pointeur ou adresse, permet de retrouver la variable référencée.

Syntaxe

`*`*`expression`*

Le type résultant est le type de la variable référencée par le pointeur.

Cet opérateur peut se trouver dans la partie droite d'une affectation ou dans une expression arithmétique ou logique, afin de retrouver la valeur (la donnée) se trouvant à une certaine adresse.

Mais il permet également, s'il se trouve dans la partie gauche d'une expression d'affectation, d'affecter une valeur à une adresse (à une variable) donnée.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Exemple

```
main () {

       int a,*p;

       a = 10;
       p = &a;      /* p pointe sur a */
       printf ("Valeur de la variable a:%d\n", *p);
       /* valeur de la donnée à l'adresse p */
       *p = 30;
       /* affectation d'une valeur à une certaine adresse */
       printf ("Valeur de la variable a:%d\n", *p);
       *p += 4;
       /* Calcul complexe mettant les deux utilisations en jeu */
       printf ("Valeur de la variable a:%d\n", *p);
}
```

Résultat

```
Valeur de la variable a:10
Valeur de la variable a:30
Valeur de la variable a:34
```

Remarques

Il faut remarquer la forte cohérence dans les notations relatives au pointeurs, ainsi, la déclaration

```
int *pointeur;
```

peut être comprise comme la définition d'un pointeur sur un entier, mais elle indique aussi que `*pointeur` est une variable de type entier (en fait, il n'y a pas obligatoirement de telle variable, la définition d'un pointeur n'impliquant pas automatiquement qu'il existe une variable sur laquelle il pointe, mais c'est une manière agréable de se souvenir).

Ainsi, dès que l'on a fait pointer `p` sur la variable `a`, il existe deux manières d'accéder à la variable. Soit avec son nom, soit avec le pointeur.

AUTRES OPERATEURS

On peut bien-entendu **affecter** l'adresse d'un objet du bon type à un pointeur.
Tous les opérateurs de **comparaison** sont utilisables avec les pointeurs, qui ne peuvent toutefois être comparés qu'à un autre pointeur où à la constante `NULL`.
Seules l'addition, la soustraction d'une valeur entière et la soustraction de deux pointeurs sont des opérations **arithmétiques** sensées.
Un entier, additionné ou soustrait à un pointeur est d'abord multiplié par la taille du type pointé de façon à correspondre à un déplacement en **nombre d'éléments** de ce type. Réciproquement, la soustraction de deux pointeurs sur un même type retourne le nombre d'éléments de ce type situés entre les deux pointeurs.

CONTROLE

POINTEURS DE STRUCURES

On utilise fréquemment des pointeurs sur des structures, notamment lorsqu'on travaille avec des structures de données dynamiques. Le langage possède ainsi un opérateur spécial pour accéder aux champs d'une stucture pointée.

Si on a *ptr* un pointeur sur une variable de type structure avec un champ *champ*, l'expression

ptr->*champ*

remplace avantageusement

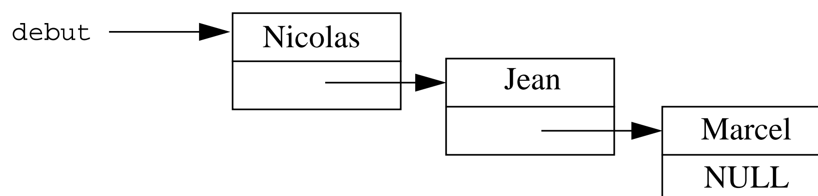
(*ptr).*champ*

Exemple

Imaginons qu'on ait les définitions suivantes

```
struct personne {
      char prenom[20];
      struct personne *pere;
};

struct personne   le_grand_pere = {"Marcel", NULL},
                  le_pere = {"Jean", NULL},
                  le_fils = {"Nicolas", NULL};
                  *debut;

le_pere.pere = &le_grand_pere;      /* Lier les enregistrements */
le_fils.pere = &le_pere;
debut = &le_fils;
```

Cela nous définit une **chaîne** d'enregistrements:



`debut` est un pointeur sur la variable `le_fils` de type `struct personne`
`*debut` = `le_fils`
`(*debut).prenom` = **`debut->prenom`** = `le_fils.prenom` = `"Nicolas"`
`(*debut).pere` = **`debut->pere`** = `le_fils.pere` est un pointeur sur la variable `le_pere` de type `struct personne`
`(*(*debut).pere).prenom` = **`debut->pere->prenom`** = `le_pere.prenom` = `"Jean"`
`(*(*debut).pere).pere` = **`debut->pere->pere`** pointe sur la variable `le_grand_pere` de type `struct personne`
`(*(*(*debut).pere).pere).prenom` = **`debut->pere->pere->prenom`** = `le_grand_pere.prenom` = `"Marcel"`

CONTROLE

Nous avions vu qu'en C, toutes les paramètres sont passés **par valeur** lors des appels de fonction. Ainsi, la fonction ne reçoit qu'une copie de la valeur, et pas la valeur originale. De cette manière, une fonction ne peut pas modifier la valeur d'une variable intempestivement.

Mais c'est une arme à double tranchant, car il serait parfois justement utile de pouvoir modifier les variables passées en paramètre. Les pointeurs permettent de briser l'étanchéité imposée par le passage par valeur. Le principe est simple, on passe l'**adresse** de la variable que la fonction doit modifier (autrement dit, une référence à cette variable). Cette adresse est obligatoirement passée par valeur, on a pas le choix, mais elle suffit à la fonction pour aller modifier la valeur de la variable.

Noter qu'il est parfois utile de modifier le pointeur lui-même. Dans ce cas, on a recours à des indirections multiples, en passant un pointeur sur un pointeur (un pointeur est une variable comme une autre et il a donc une adresse).

Exemple

```
#include <stdio.h>

void incrementer();

main() {
      int nombre=11;

      printf("%d\n",nombre);
      incrementer(&nombre);
      printf("%d\n",nombre);
}

void incrementer(int *nombre) {
      (*nombre)++;
}
```

Résultat à l'exécution

```
11
12
```

Comme le montre cet exemple, le passage par référence n'est pas une opération compliquée. La page suivante donne un exemple un peu plus intéressant.

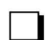
Exemple

```
void permute ();         /* passage par valeur */
void permuteref ();      /* fonction de permutation de deux variables
                         de type entier, passees par reference */

main () {
      int a=10, b=20;

      printf ("a vaut %d et b vaut %d.\n", a, b);
      permuteref (&a, &b);
      /* doit bien fonctionner */
      printf ("a vaut %d et b vaut %d.\n", a, b);
      permute (a, b);
      /* ne doit pas fonctionner */
      printf ("a vaut %d et b vaut %d.\n", a, b);
}

void permuteref (pa, pb)
int *pa, *pb;      /* deux pointeurs sur les entiers a permuter */
{
      int tampon;
      tampon = *pb;
      *pb = *pa;
      *pa = tampon;
}

void permute (a, b)
int a, b;          /* deux valeurs entieres */
{
      int tampon;
      tampon = b;
      b = a;
      a = tampon;
}
```

Résultat

```
a vaut 10 et b vaut 20.
a vaut 20 et b vaut 10.
a vaut 20 et b vaut 10.
```

Le passage par référence permet de permuter les valeurs de deux variables (fonction `permuteref()`) alors que la fonction `permute()` ne fonctionne pas puisquelle travaille sur des **copies** des variables.



FICHIERS ET LIBRAIRIE STANDARD

Un des rôles du **système d'exploitation** est d'offrir un jeu de primitives de base pour la manipulation des fichiers. Les primitives en question étant différentes d'un système d'exploitation à l'autre, accéder aux fichiers par l'intermédiaire de ces dernières pose de gros **problèmes de portabilité** des logiciels.

C'est pourquoi la **librairie standard** propose une autre panoplie de fonctions pour la manipulation des fichiers. Cet ensemble de fonctions est pratiquement identique d'une machine ou d'un compilateur à l'autre. Il est donc vivement conseillé de favoriser l'emploi de ce niveau d'accès plutôt que celui du système d'exploitation, les primitives de ce dernier restant toujours utilisables pour des applications plus spécialisées.

Au niveau de la librairie, un fichier est vu comme une **collection de bytes** n'ayant aucune structure particulière. Ces fichiers sont accédés principalement de manière **séquentielle** (tous les bytes sont lus l'un après l'autre), bien qu'un **accès aléatoire** (possibilité d'accéder à un byte précis directement en effectuant des "sauts") soit possible pour les dispositifs l'autorisant. On appelle souvent ce type de fichiers des **streams**, **flots** ou **flux**.

Les fonctions de la librairie possèdent la caractéristique d'être "**tamponnées**", c'est-à-dire que ces dernières gèrent un espace tampon intermédiaire entre la demande de lecture ou d'écriture et la réalisation proprement dite de cette demande. Les informations sont "écrites dans" ou "lues depuis" une mémoire tampon. Une fois ce tampon rempli ou consommé, il sera "vidé", respectivement "rempli", par un appel aux primitives de bas niveau (celles du système d'exploitation). Cet aspect tamponné est intéressant du point de vue des **performances** pour des fichiers traités sequentiellement; en revanche, pour des fichiers auxquels on accéde de manière aléatoire, les primitives de plus bas niveau risquent d'être plus performantes.

Un autre avantage des fonctions de bibliothèque, et non des moindres, est la présence de fonctions d'**entrée-sortie formatées**, semblables à celles qui sont utilisées pour lire au clavier ou écrire à l'écran.

Une caractéristique de la plupart des systèmes d'exploitation actuels est la "**banalisation** des entrée-sorties", c'est-à-dire qu'on cherche à ne plus faire de différences entre les périphériques et les fichiers. Tout périphérique est considéré et manipulé de la même manière qu'un fichier. Le système d'exploitation gère bien-entendu différemment les diverses unités périphériques mais, vu de l'extérieur, cette homogénéité offre une gestion beaucoup plus agréable. Toutefois, dûes à la nature différente des périphériques, certaines restrictions peuvent apparaître (lire sur une imprimante...).

Stdin, stdout et stderr

Les quelques fonctions d'entrée-sortie déjà vues lisaient ou écrivaient leurs informations sur ou depuis la console (clavier et écran); en réalité, ces opérations s'effectuent sur des fichiers pré-ouverts qui sont associés à la console. Il existe trois fichiers, pré-ouverts par le système, qui sont associés par défaut à la console:

- **`stdin`** (standard input) par défaut le canal d'entrée de la console.
- **`stdout`** (standard ouput) par défaut le canal de sortie de la console.
- **`stderr`** (standard error) un fichier sur lequel les messages d'erreur sont affichés; par défaut le canal de sortie de la console.

La redirection des entrée-sorties au niveau de l'interpréteur de commande met en évidence l'utilité du fichier **`stderr`**. Il est ainsi possible de dérouter ce fichier, par exemple vers un fichier sur disque, et d'être en mesure de consulter les messages d'erreur produits par l'application, une fois celle-ci terminée, sans être perturbé par ses affichages. Généralement, les systèmes dépourvus d'un système d'exploitation multi-tâches ou multi-utilisateurs offrent quelques fichiers pré-ouverts supplémentaires comme, par exemple, **`stdprn`** pour le canal imprimante ou **`stdaux`** pour un canal de liaison série.

Accès aux fichiers

La chaîne de caractères identifiant un fichier pour le DOS est un nom, dit, **externe**. Au sein d'un programme, cette référence ne suffit pas, il faut disposer d'informations supplémentaires comme l'adresse du buffer, l'avancement dans celui-ci, etc. En fait, un fichier sera caractérisé, à l'intérieur d'un programme, par un **pointeur sur une structure `FILE`** contenant toutes ces informations. On note souvent ce pointeur **`fp`** pour "file pointer".

Stdio.h

Tout fichier employant des fonctions d'entrée-sortie de la bibiliothèque standard doit inclure le fichier **`stdio.h`**. Ce fichier d'en-tête regroupe différentes déclarations de constantes et de fonctions, quelques macros et la définition de la structure **`FILE`**. Cette structure contient toutes les informations nécessaires à la gestion d'un flux et de son buffer.

Exemple

```
#include <stdio.h>

main() {
      FILE *fichier;
      ...
}
```

fopen()

```
FILE *fopen(filename, mode)  /* declaration dans stdio.h */
char filename[], mode[];
```

Cette fonction établit le lien entre un fichier de nom externe ***filename*** et une nouvelle structure **FILE**, crée automatiquement, et qui sera utilisée par toutes les opérations relatives à celui-ci.

On parle alors d'opération d'**ouverture** de fichier.

En plus du nom de fichier, **fopen()** requiert un deuxième argument: le mode d'ouverture de ce dernier qui déterminera les **opérations applicables** au fichier. L'argument ***mode*** est une chaîne de caractères pouvant prendre les valeurs suivantes:

- **"r"** ouverture en lecture d'un fichier existant.
- **"w"** ouverture en écriture (création si le fichier n'existe pas ou écrasement si celui-ci existe déjà).
- **"a"** ouverture en ajout (création si le fichier n'existe pas ou ajout en fin de fichier si celui-ci existe déjà).
- **"r+"** ouverture en modification d'un fichier existant (lecture et écriture) en commençant en début de fichier.
- **"w+"** ouverture comme pour "w" en autorisant la modification.
- **"a+"** ouverture comme pour "a" en autorisant la modification.

Pour une ouverture en modification, le flux peut être utilisé en lecture et en écriture.

La norme ANSI propose, pour n'importe quel mode mentionnés plus haut, d'ajouter en fin de chaîne le caractère **'b'** ou **'t'** pour traiter le flux en **mode binaire** ou en **mode texte**. Le mode binaire permet, sur certaines machines, de ne pas étendre le caractère **'\n'** en un retour de chariot **'\r'** suivi du caractère d'interligne. Le mode texte effectue cette transformation par défaut sur les PC.

La fonction fopen() retourne **un pointeur sur la structure FILE** du fichier nouvellement crée si l'ouverture s'est déroulée convenablement; dans le cas contraire elle retourne le pointeur **NULL** (fichier inexistant, protégé, ...).

Exemple

```
#include <stdio.h>
main () {
      FILE *fichier_a_lire;

      fichier_a_lire = fopen ("blabla.txt", "r");
      ...
}
```

LIBRARIES

fclose()

```
int fclose(stream)            /* declaration dans stdio.h     */
FILE *stream;
```

Opération de "fermeture" d'un fichier ***stream***. Les mémoires tampon de ce dernier seront vidées et leur espace libéré. Le lien établi par **fopen()** entre le nom du fichier proprement dit et sa structure **FILE** sera ainsi "cassé".

La fonction **fclose()** retourne **0** si aucune erreur n'est intervenue, **EOF** (constante pré-définie) dans le cas contraire.

exit()

Un appel à la fonction **exit()** a pour effet, non seulement de terminer le programme, mais aussi de vider tous les tampons et de fermer tous les fichiers ouverts.

Il est généralement nécessaire d'inclure le fichier de décalarations **stdlib.h** pour pouvoir utiliser cette fonction.

LIBRARIES

Fonctions d'entrée-sortie de haut-niveau

Parallèlement à **`printf()`** et **`scanf()`**, il existe des fonctions de haut-niveau pour écrire et lire dans des fichiers. Il s'agit de **`fprintf()`** et **`fscanf()`** qui s'utilisent exactement de la même manière, en ajoutant seulement le pointeur de fichier comme premier argument.

```
int fprintf(stream, format [ , arguments... ] )
FILE *stream;
char format[];
```

```
int fscanf(stream, format [ , pointeurs... ] )
FILE *stream;
char format[];
```

Quand on utilise ces fonctions, on parcourt le fichier simplement comme un flux.

Exemple

```
#include <stdio.h>
main()
{
  FILE *fp;
  char filename[20];
  printf("Entrer un nom de fichier : ");
  gets(filename);

  /* ouverture du fichier en écriture uniquement */
  if ((fp = fopen(filename,"w")) == NULL) {
    printf("Impossible d'ouvrir le fichier %s\n", filename);
    exit(1);
  }

  /* écriture dans le fichier */
  fprintf(fp, "\nCe fichier se nomme: %s\n",filename);

  /* fermeture du fichier */
  if (fclose(fp) == EOF) {
    printf("Fermeture impossible\n");
    exit(1);
  }
}
```

Remarque

Il est très important, étant donné que l'on travaille avec des pointeurs, et le risque de planter le système pendant des opérations de manipulation de fichiers, de **toujours vérifier que les opérations d'ouverture et de fermeture se sont bien passées**.

LIBRARIES

Entrées-sorties de caractères

Il est bien souvent possible ou nécessaire de traiter un fichier caractère par caractère. La bibliothèque propose donc une fonction de lecture, une fonction d'écriture mais aussi une fonction de remise en mémoire tampon d'un caractère. Cette dernière est utile lorsqu'il est nécessaire de connaître la valeur du caractère qui suit le caractère courant. On peut donc lire le caractère suivant, puis, le remettre artificiellement dans le flot.

fgetc(), getc(), getchar()

```
int fgetc(FILE *stream);       /* ANSI, prototype dans stdio.h */
int getc(FILE *stream);        /* ANSI, definie dans stdio.h   */
int getchar(void);             /* ANSI, definie dans stdio.h   */


int fgetc(stream)              /* declaration dans stdio.h     */
FILE *stream;

int getc(stream)               /* definition dans stdio.h      */
FILE *stream;

int getchar()                  /* definition dans stdio.h      */
```

La fonction **`fgetc()`** retourne le caractère lu depuis le flot ***`stream`***, et la valeur **`EOF`** en cas de lecture de la marque de fin de fichier ou en cas d'erreur.

La fonction **`getc()`** est identique à **`fgetc()`** excepté que c'est une macro définie dans le fichier **`stdio.h`**.

La fonction **`getchar()`**, quant à elle, est aussi une macro définie dans ce fichier d'en-tête comme l'abréviation de **`getc(stdin)`**. Elle ne permet donc que de lire dans l'entrée standard.

Ces trois fonctions possèdent la particularité de retourner une information de type entier et non caractère. Cette particularité est nécessaire pour pouvoir différencier un caractère quelconque (une combinaison 8 bits) de la valeur **`EOF`**. Si **`EOF`** est définie comme la valeur -1 (0xFFFF) il est alors possible de la différencier d'un caractère exprimé sous forme 16 bits (0x00??).

LIBRARIES

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

fputc(), putc(), putchar()

```
int fputc(char c, FILE *stream); /* ANSI, proto. dans stdio.h */

int putc(char c, FILE *stream);  /* ANSI, def. dans stdio.h   */

int putchar(char c);          /* ANSI, definie dans stdio.h    */

int fputc(c, stream)          /* declaration dans stdio.h      */
char c;
FILE *stream;

int putc(c, stream)           /* definition dans stdio.h       */
char c;
FILE *stream;

int putchar(c)                /* definition dans stdio.h       */
char c;
```

La fonction **fputc()** est une fonction de la bibliothèque qui écrit le caractère ***c*** sur le flot ***stream***.

Comme **getc()**, **putc()** est une macro définie dans le fichier **stdio.h**. Son le comportement est identique à **fputc()**.

La fonction **putchar()**, quant à elle, est aussi une macro définie dans ce fichier d'en-tête comme l'abréviation de **putc(c, stdout)**. Elle ne permet donc que d'écrire dans la sortie standard.

Ces trois fonctions retournent le caractère écrit, ou **EOF** en cas d'erreur.

ungetc()

```
int ungetc(int c, FILE *stream) /* ANSI, proto. dans stdio.h */

int ungetc(c, stream)         /* declaration dans stdio.h     */
int c;
FILE *stream;
```

La fonction **ungetc()** remet le caractère **c** dans la mémoire tampon associée au flot ***stream***. Le prochain appel à la fonction **fgetc()**, **getc()** ou **getchar()** sur ce flot, retournera alors ce caractère. Un seul caractère peu ainsi être remis dans le tampon. **ungetc()** retourne le caractère *c* si celui-ci a pu être remis correctement dans le tampon sinon, elle retourne **EOF**. La remise en mémoire tampon de la valeur **EOF** n'a aucun effet et, dans ce cas, la fonction retourne cette valeur.

LIBRARIES

Comme pour les caractères, il existe quelques fonctions bien pratiques pour lire et écrire des chaînes de caractères dans des fichiers.

fgets(), gets()

```
/* ANSI, prototype dans stdio.h */
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);


char *fgets(s, n, stream)     /* declaration dans stdio.h     */
char s[];
int  n;
FILE *stream;

char *gets(s)                 /* declaration dans stdio.h     */
char s[];
```

fgets() lit une séquence de caractères depuis le flux ***stream*** et la stocke dans la chaîne ***s***. La lecture s'effectue jusqu'à concurence de ***n***-1 caractères, cela afin de ne pas dépasser la longueur de la chaîne; ou lorsque le caractère d'interligne **'\n'** est lu. Ce caractère d'interligne est **copié** dans la chaîne **s**, puis celle-ci est automatiquement terminée par le caractère de terminaison **'\0'**.

La fonction **gets()** lit une séquence de caractères depuis l'entrée standard en la stockant dans la chaîne ***s***. Le nombre maximum de caractères à lire n'étant pas fixé, la lecture cesse lorsque le caractère **'\n'** intervient, ce caractère n'est **pas copié**; finalement la chaîne ***s*** est terminée par **'\0'**.

fgets() et **gets()** retournent un pointeur sur la chaîne lue, ou **NULL** en cas d'erreur ou de détection de fin de fichier.

fputs(), puts()

```
/* ANSI, prototype dans stdio.h */
int fputs(const char *s, FILE *stream);
int puts(const char *s);


int fputs(s, stream)          /* declaration dans stdio.h     */
char s[];
FILE *stream;

int puts(s)                   /* declaration dans stdio.h     */
char s[];
```

La fonction **fputs()** copie sur le flot ***stream*** la chaîne ***s*** terminée par le caractère **'\0'**. Le caractère **'\n'** n'est pas ajouté. **puts()**, par contre, copie la chaîne ***s*** sur la sortie standard en ajoutant le caractère d'interligne.

Les deux fonctions retournent le dernier caractère écrit, ou **EOF** en cas d'erreur.

LIBRARIES

feof()

```
int feof(FILE *stream);       /* ANSI, prototype dans stdio.h */

int feof(stream)              /* declaration dans stdio.h     */
FILE *stream;
```

Teste pour le fichier ***stream*** si la marque de fin de fichier a été lue.

Retourne une valeur **non nulle** (vraie) si celle-ci a été lue, **0** (fausse) sinon.

Cette fonction permet de distinguer l'erreur de la lecture de fin de fichier pour les fonctions qui retournent **EOF** dans les deux cas (par exemple `getc()`).

L'indicateur de fin de fichier ne sera effacé que par un appel à **clearerr()**, **rewind()**, **freopen()** ou **fclose()**.

ferror()

```
int ferror(FILE *stream);     /* ANSI, prototype dans stdio.h */

int ferror(stream)            /* declaration dans stdio.h     */
FILE *stream;
```

Retourne une valeur **non nulle** (vraie) si une erreur s'est produite lors d'une entrée/sortie sur le fichier ***stream***, sinon renvoie **0**.

L'indicateur d'erreur reste positionné jusqu'à la fermeture du fichier ou un appel à la fonction **clearerr()**.

clearerr()

```
void clearerr(FILE *stream); /* ANSI, prototype dans stdio.h */

void clearerr(stream)         /* declaration dans stdio.h     */
FILE *stream;
```

Remet à zéro l'indicateur d'erreur et celui de fin de fin de fichier pour le flux ***stream***.

LIBRARIES

freopen()

```
/* ANSI, prototype dans stdio.h */
FILE *freopen(const char *name, const char *mode, FILE *stream);
```

```
FILE *freopen(name, mode, stream) /* declaration dans stdio.h */
char name[], mode[];
FILE *stream;
```

Le flux ***stream*** est fermé puis, le fichier de nom ***name*** est ouvert en lui assignant la mémoire tampon ***stream***.

Le mode d'ouverture ***mode*** est indique par le même code que pour l'ouverture de fichier normale[1].

En cas d'ouverture du fichier ***name*** réussie **freopen()** retourne le flux ***stream*** passé en argument, sinon retourne le pointeur **NULL**.

Cette fonction est souvent employée pour rediriger les entrées/sorties standard **stdin**, **stdout** et **stderr** vers, ou depuis, un fichier de nom ***name***.

Exemple

```
freopen("enreg.txt", "w", stdout);
```

Toute écriture sur **stdout** aura donc lieu, après cette instruction, sur le fichier **"enreg.txt"**.

fflush()

```
int fflush(FILE *stream);     /* ANSI, prototype dans stdio.h */

int fflush(stream)            /* declaration dans stdio.h     */
FILE *stream;
```

Force le contenu de la mémoire tampon associée au flux ***stream*** d'être transféré sur le média.

Ainsi vidé, de nouvelles écritures pourront avoir lieu dans ce tampon.

L'ordre **fflush(stdout)** provoque l'affichage du tampon associé au flux **stdout** et vide celui-ci.

1. Voir fopen(), page VII.C.3

LIBRARIES

Outre **fflush()**, deux situations provoquent l'affichage: un tampon plein et la présence du caractère **'\n'**.

L'instruction **fflush(stdin)** vide, quant à elle, le tampon associé à l'entrée standard et permet ainsi d'ignorer les caractères jusque-là non consommés.

La fonction **fflush()** retourne **0** si aucune erreur n'est intervenue, **EOF** sinon.

rewind()

```
void rewind(FILE *stream);     /* ANSI, prototype dans stdio.h */

void rewind(stream)            /* declaration dans stdio.h     */
FILE *stream;
```

Cette fonction a pour effet de "rembobiner" le flux ***stream***.

D'une manière analogue au rembobinage d'une bande magnétique, on revient au tout premier byte du fichier, ce qui rend une nouvelle lecture du fichier possible sans avoir à l'ouvrir une seconde fois.

Par ailleurs, cette fonction remet à zéro les indicateurs d'erreur et de fin de fichier pour le fichier ***stream***.

LIBRARIES

On dit d'un fichier qu'il est utilisé en **accès direct** ou **aléatoire** s'il est possible de lire ou d'écrire des infomations dedans dans un ordre quelconque. Le contraire est un parcours **séquentiel**, qui ne permet que d'accéder les données dans l'ordre.

En C, la distinction n'est pas faite sur les fichiers, on suppose que le système d'exploitation est capable de fournir les modes d'accès séquentiel et direct. Ce sont les **fonctions** qui font la différence entre accès direct et séquentiel.

Toutes les fonctions que nous avons vu jusqu'ici travaillent en "mode séquentiel", elles avancent du nombre de caractère qu'elles ont pu lire ou écrire.

Les deux fonctions suivantes permettent à tout moment de se positionner n'importe où dans le programme et de connaître la position actuelle.

fseek()

```
/* ANSI, prototype dans stdio.h */
int fseek(FILE *stream, long offset, int method);
```

```
/* declaration dans stdio.h */
int fseek(stream, distance, method)
FILE *stream;
long distance;
int  method;
```

Après ouverture d'un fichier, l'emplacement de lecture ou d'écriture est positionné au début de celui-ci, ou en fin pour le mode ajout. Les entrées/sorties accèdent séquentiellement les streams, c'est-à-dire que l'opération de lecture ou d'écriture a lieu à l'emplacement courant puis, cet emplacement est “avancé” du nombre de bytes transférés par l'opération.

La fonction `fseek()` est donc là pour déplacer la position courante de lecture ou d'écriture. L'argument ***distance (=offset)*** représente le déplacement (positif ou négatif) en bytes à effectuer dans le fichier ***stream***. Il faut se méfier avec ce paramètre car il doit être de type `long`, donc si une constante est utilisée, elle doit avoir le suffixe `L`.

Trois méthodes de déplacement sont à disposition. Ce choix est fixé par le paramètre ***method***; suivant sa valeur le déplacement aura lieu :

- `0` (ou la constante prédéfinie `SEEK_SET`) : par rapport au début du fichier
- `1` (`SEEK_CUR`) : par rapport à l'emplacement courant
- `2` (`SEEK_END`) : par rapport à la fin du fichier

La fonction `fseek()` retourne une valeur non nulle (vraie) si le déplacement n'a pu avoir lieu, `0` dans le cas contraire.

A remarquer que `SEEK_END` et une distance négative a pour effet de reculer depuis la fin.

LIBRARIES

Ainsi, l’appel **`rewind(fp)`** est identique à **`fseek(fp, 0L, 0)`**, excepté que **`fseek()`** ne met pas les indicateurs d’erreur à zéro.

ftell()

```
long ftell(FILE *stream);     /* ANSI, prototype dans stdio.h */

long ftell(stream)            /* declaration dans stdio.h     */
FILE *stream;
```

La fonction **`ftell()`** retourne la position courante de lecture/écriture du fichier ***`stream`*** mesurée en bytes depuis le début du fichier. Cette fonction est utile pour mémoriser une position afin de s’y repositionner ultérieurement.

Ainsi, si on désire trouver la longueur d’un fichier, on peut le faire facilement par les deux instructions

```
FILE *fich;
int longueur;

fseek (fich, 0L, 2); /* aller après le dernier byte du fichier */
longueur = ftell(fich); /* numéro du dernier byte */
```

LIBRARIES

Ces fonctions de la librairie standard permettent de lire ou d'écrire des données de type quelconque. Elle sont typiquement utilisées pour effectuer des entrées/sorties de scalaires sous forme binaire, de tableaux ou de structures. Il faut généralement ouvrir le fichier en mode binaire (b) pour que ces opérations se déroulent correctement.

fread()

```
/* ANSI, prototype dans stdio.h */
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);


int fread(ptr, size, n, stream) /* declaration dans stdio.h  */
char *ptr;
unsigned size;
int  n;
FILE *stream;
```

Cette fonction lit depuis le flot **`stream`** un nombre d'informations **`n`** dont chacune est de taille **`size`**. Le résultat de cette lecture est stocké à partir de l'adresse **`ptr`**.

La fonction retourne le nombre d'informations lues si aucune erreur n'est intervenue. Ce nombre peut être inférieur au nombre **`n`** si la fin de fichier a été rencontrée ou lorsqu'une erreur intervient.

Les fonctions **`feof()`** et **`ferror()`** peuvent être employées pour distinguer une erreur d'une fin de fichier lorsque la valeur retournée est inférieure à **`n`**.

fwrite()

```
/* ANSI, prototype dans stdio.h */
size_t fwrite(const void *ptr,size_t size,size_t n,FILE *stream);

int fwrite(ptr, size, n, stream) /* declaration dans stdio.h  */
char *ptr;
unsigned size;
int  n;
FILE *stream;
```

`fwrite()` transfère, sur le flot **`stream`**, **`n`** informations de taille **`size`** stockées à partir de l'adresse **`ptr`**.

La fonction retourne le nombre d'informations écrites, une erreur a eu lieu si ce nombre est inférieur à **`n`**.

Ces deux fonctions s'utilisent en général avec les fonctions de déplacement décrites en page précédente pour effectuer des entrées/sorties d'enregistrements en accès direct.

Les tableaux tels que nous avons vu jusqu'ici n'ont qu'une seule dimension (un seul indice). Ils se prêtent bien à de nombreux problèmes, mais il est parfois utile d'avoir des tableaux de dimensions supérieures (2, 3, ou plus) pour contenir des données qui ont justement une forme naturelle de tableau (pensez à un échiquier).

Pour définir un tableau à **plusieurs dimensions**, il faut mettre une taille entre crochets pour chacune des dimensions du tableau.

Exemples

```
int   tab2dim[100][50];        /* 100 lignes de 50 entiers */
float tab3dim[10][10][10];     /* "cube" de nombres flottants */
char  ecran[25][80];           /* ce n'est pas une chaîne
                               25 lignes de 80 caractères */

enum couleur {noir, blanc};
enum couleur echiquier[8][8];
```

Pour **accéder à un élément** d'un tableau donné, il faut mettre autant d'indices entre crochets qu'en comporte la déclaration de ce tableau.

Exemples

```
tab2dim[4][4] = 14;
tab3dim[7][8][1]++;

char a = ecran[10][0];
```

Remarques

Comme pour les tableaux à une dimension, les **indices** vont entre **0** et **taille-1** dans chacune des dimensions.

Il est possible d'interpréter le tableau à trois dimensions

```
int t[2][7][4];
```

comme un tableau de tableau de tableau d'entiers. Ainsi, avec **i** entre **0** et **1**, **j** entre **0** et **6** et **k** entre **0** et **3**,

- **t[i][j][k]** est un entier,
- **t[i][j]** est un tableau de **4** éléments,
- **t[i]** est un tableau de **7** tableaux de **4** éléments,
- **t** est un tableau de **2** tableaux de **7** tableaux de **4** éléments;

N'importe laquelle de ces expressions est valable si le contexte requiert une variable de ce type (par exemple comme paramètre d'une fonction).

DONNEES

Nous avons vu que la définition ou la déclaration d'un tableau à une dimension ne doit pas forcément définir la **taille** du tableau, puisqu'il est possible de laisser les crochets vides. Pour les tableaux à plusieurs dimensions, le calcul des indices impose que seule la première dimension puisse ne pas avoir de spécification de taille. Toutes les autres dimensions doivent avoir une taille fixée par une **expression constante** lors de la définition ou de la déclaration du tableau.

On comprend cette restriction si on se souvient que le type de base d'un tableau doit être **complet** et qu'un tableau dont la taille n'est pas fixée n'est pas complet.

Pour les mathématiciens, il sera peut-être important de noter que si on utilise la convention habituelle d'attribuer le premier indice au numéro de ligne et le second au numéro de colonne, alors les matrices sont **stockées par lignes** en C. Car un tableau

```
float mat[nb_lignes][nb_colonnes];
```

à deux dimensions étant en fait un tableau de `nb_lignes` tableaux de `nb_colonnes` éléments.

Enfin, on peut regretter que contrairement à d'autres langages, C ne fournisse pas de notation abrégée pour les indices. En effet, il faut obligatoirement mettre des crochets autour de chaque indice séparément.

DONNEES

En C, toute variable peut peut recevoir une valeur initiale. Les tableaux ne font pas exception à cette règle.

Une **valeur initiale** peut être affectée à un tableau en faisant suivre sa définition d'un signe = et d'une liste de valeurs initiales, entre accolades ({ et }) et séparées par des virgules.

```
int tab[3] = { 24, 120, 720 };
```

Les éléments de la liste doivent être des **expressions constantes**, donc ne contenant ni variables ni appels de fonctions.

Si la taille du tableau est **fixée** par une expression entre les crochets, la liste ne doit pas avoir plus d'éléments que le tableau ne peut en contenir. Elle peut par contre être plus courte est dans ce cas, les valeurs restantes seront initialisées à zéro.

```
int tab[10] = { 1, 1, 2, 6 }; /* complete par des 0 */
int tab[4] = { 1, 2, 3, 4, 5, 6, 7, 8 }; /* est interdit */
```

Si la taille du tableau n'est **pas fixée** par une expression entre crochets, alors la taille de la liste fixe la taille du tableau.

```
float tab[] = { 10, 20, 30, 40 }; /* fixe la taille à 4 */
```

Les tableaux (chaînes) de **caractères** peuvent être initialisées par une liste, mais aussi par une chaîne de caractères constante entre guillemets. Attention à la convention du caractère nul si on utilise une liste pour initialiser la chaîne, il n'est pas rajouté automatiquement.

```
char string[] = "Hello";
char string[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Lorsqu'on a affaire à des tableaux à plusieurs dimensions, il est possible de mettre des **sous-listes** dans la liste, contenant chacune les valeurs des "sous-tableaux".

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

Il est aussi possible de mettre les valeurs à la suite, sans que la structure du tableau n'apparaisse dans la liste. Dans ce cas, le tableau est rempli dans l'ordre, ligne par ligne et complété par des zéros si nécessaire.

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

DONNEES

Remarques

Bien que les initialisations de tableaux soient autorisées dans la norme ANSI, certains compilateurs n'autorisent cette opération que pour des tableaux de classe d'allocation **static** ou variables globales.

Il ne faut pas oublier que l'opérateur = est ici pris comme **initialisateur**et que l'**affectation** entre tableaux n'est jamais autorisée.

Tableaux de chaînes

Il est souvent utile de regrouper un ensemble de messages (par exemple des messages d'erreur) dans un tableau de chaînes de caractères. Un seule procédure à laquelle on passe le 'numéro' du message désiré permet de tous les afficher. Un tel tableau peut être défini et initialisé simplement par

```
[static] char messages[][constante] = {
      "Message0",
      "Message1",
      ...         };
```

Les crochets autour du mot réservé `static` indiquant qu'il n'est pas toujours nécessaire, *messages* est le nom du tableau et *constante* doit être au moins aussi grande que la plus longue des chaînes.

On peut utiliser un type énuméré pour définir des constantes correspondant aux différentes erreurs possibles.

Exemple

```
#define LONGMSG 50
enum fxy_err {FXY_OK, FXY_TROP, FXY_PAS_ASSEZ};

char fxy_msg[][LONGMSG]={
      "Tout va bien",
      "Paramètre trop grand",
      "Paramètre trop petit"  };

fxy_err fxy (param) int param; {
      if (param>10) return FXY_TROP;
      if (param<4) return FXY_PAS_ASSEZ;
      ...
      return FXY_OK;
}

main() {
      fxy_err valeur;
      int a;
      ...
      valeur = fxy (a);
      if (valeur!=FXY_OK) printf ("%s\n", fxy_msg[valeur]);
}
```

DONNEES

Comme les tableaux, les structures peuvent aussi être **initialisées** par des listes d'expressions constantes, entre accolades et séparées par des virgules.

Elles peuvent par-contre aussi être initialisées par une **expression non-constante** si elles ne sont pas de classe statique. Le type de l'expression devra bien-entendu être correct. Cela permet par exemple d'affecter à une structure la valeur retournée par une fonction.

Si la liste d'expressions comporte moins de champs que la structure elle-même, alors les champs restant sont **remplis par des zéros**. Une liste plus longue que le nombre de champs de la structure est interdite.

Si certains champs de la structure sont eux-aussi des structures ou des tableaux, alors il est possible de mettre des **sous-listes** comme valeurs initiales pour les structures internes.

Exemple

```
struct logement {
      int nombre_de_pieces, superficie, etage;
      struct {
            char nom[20], prenom[20];
      } locataire;
      char rue[20], description[4][50];
};

struct logement home_sweet_home =
      {
            3, 40, 6, {"Queloz", "Pierre-Antoine"},
            "Rue des Pâquis",
            "Bien situé avec jolie vue et bon ensoleillement"
      };
```

Les unions sont de nouvelles structures de données assez proches des structures. Elles permettent de stocker dans une variable des données de **types différents**. Par contre, elles ne peuvent contenir qu'**une seule donnée à la fois**.

En fait, la seule différence entre les unions et les structures est la manière d'organiser les champs en mémoire. Les structures stockent leurs champs **séquentiellement** et la taille d'une structure est donc la somme de la taille de chacun de ses champs, plus quelques bytes éventuellement. Les unions stockent tous leurs champs à la **même adresse**, c'est-à-dire que les champs d'une union se recouvrent, ce qui explique qu'il ne puisse y avoir qu'une seule donnée à la fois dans une union. La taille d'une union est alors la taille du composant le plus "volumineux" faisant partie de l'union.

```
union exemple {
      char  car_val;          /* contient un caractère */
      int   ent_val;          /* ou un entier */
      float flo_val;          /* ou un flottant */
      char  tex_val[10];      /* ou du texte */
};
```

Bien-sûr, au niveau le plus bas, seuls des bits sont stockés. Le type de chacun des champs est donc nécessaire si l'on veut savoir comment interpréter la donnée contenue dans l'union.

Ainsi, il n'y a pas de sens à relire un autre champ que celui que l'on a écrit précédemment.

```
union exemple data;
data.flo_val = 123.425;
printf ("%f", data.flo_val);  /* est valide */
printf ("%d", data.ent_val);  /* n'a pas de sens car l'union
                                 contient un float.*/
```

Une variable de type union pouvant avoir à priori une valeur de type quelconque, il faut en général associer à des variables un **indicateur** (par exemple de type énuméré) qui permette de trouver quel champ est pertinent à un moment donné. En général, on associe cet indicateur et l'union elle-même dans une **structure**.

Par exemple

```
enum type_valeur {caractere, entier, flottant, chaine};

struct donnee_complexe {
      enum type_valeur type_val;
      union {
            char car_val;
            int ent_val;
            float flo_val;
            char tex_val[10];
      } valeur;
} nd;

nd.typ_val = entier;
nd.valeur.ent_val = 6;        /* affectation d'une valeur */
```

DONNEES

Comme les éléments d'un tableau sont stockés à la suite dans la mémoire centrale, il est possible de se déplacer dans un tableau à l'aide d'un pointeur.

Exploitant à fond cette propriété, le langage C va même plus loin en convertissant toutes les expressions de type **tableau d'éléments de type T**, avec T un type quelconque en une expression du type **pointeur sur donnée de type T**, avec comme adresse celle du premier élément du tableau.

Cette conversion a lieu systématiquement, sauf quand l'expression est une opérande de
- l'opérateur adresse `&`,
- les opérateurs d'incrémentation et de décrémentation `++` et `--`,
- l'opérateur de taille `sizeof`.

Ou qu'elle est opérande gauche
- d'un opérateur d'affectation
- ou de l'opérateur de sélection de champ `"."`.

On comprend à présent pourquoi les tableaux sont passés par **référence** dans les fonctions. Chaque occurence du nom de tableau est transformée en pointeur automatiquement et le mécanisme est tout à fait transparent.

Il est également possible de mettre comme paramètre formel un pointeur et comme paramètre réel un tableau de données de type correspondant.

Comme on s'en doute, la sélection d'un élément du tableau passe aussi par un pointeur, ainsi, l'opérateur `[]` est défini de la manière suivante, il prend une expression de type "pointeur sur quelque-chose" (`E1`) et une expression entière (`E2`) et il est automatiquement transformé selon la règle

```
E1[E2] <=> *( (E1) + (E2) )
```

La somme dans la partie droite est donc une somme entre un pointeur et un entier, donc qui multiplie l'entier par la taille de la donnée pointée afin d'avoir un déplacement en éléments du tableau.

De plus, dans le cas où il y plusieurs indices, l'indice le plus à droite est traité en premier.

L'utilisation des pointeurs est particulièrement élégante et efficace dans le cas du traitement des chaînes de caractères. Par exemple, la fonction qui recopie une chaîne dans une autre peut s'écrire

```
char *strcpy (dest, source) char *dest, *source;
{
      char *p=dest;
      while (*dest++ = *source++);  /* affecter, avancer, tester fin */
      return p;
}
```

DONNEES

Nous avions vu que tout programme C doit contenir une fonction particulière nommée **`main`**. Ce sont les instructions contenues dans cette fonctions qui sont exécutées lorsque le programme est lancé.

Comme les autres fonctions, `main()` peut recevoir des paramètres au moment où le programme est lancé. Ces paramètres se trouvent sur la ligne de commande tapée par l'utilisateur lorqu'il utilise un programme sous DOS. Par exemple

```
dir /w
```

Dans ce cas, le programme lancé se nomme "`dir`" et il a un paramètre, la chaîne de caractères "`/w`".

Le principe est simple, le programme reçoit **un entier** qui est le nombre de "mots" séparés par des espaces que l'utilisateur a tapés. Ce paramètre est généralement appelé **`argc`** pour "compte d'arguments".

Le second paramètre est un **tableau de pointeurs sur des chaînes** contenant chacune l'un des "mots" que l'utilisateur a tapés. Il est ainsi possible pour le programme de tester quels étaient les paramètres. La convention veut que ce paramètre soit appelé **`argv`** pour "vecteur d'arguments".

Le **premier** élément de `argv` pointe sur une chaîne qui contient le nom du programme, il est ainsi possible de connaître le nom du programme, même s'il a été renommé.

Le **dernier** élément de `argv` est un pointeur **`NULL`**.

Exemple

```
main (argc, argv)
int argc;
char *argv[];     /* tableau de pointeurs sur des caractères */
{
      int i=1;

      printf ("Ce programme s'appelle %s.\n", argv[0]);

      if (argc==1) printf ("Il n'a pas d'arguments.\n");
      else {
            printf ("Voici ses arguments:\n");
            while (i<argc) {
                  printf ("\t%s\n", argv[i]);
                  i++;
            }
      }
}
```

Résultats

```
C:\>a.out aha bebe coucou          tapé par l'utilisateur sous DOS
Ce programme s'appelle a.out.
Voici ses arguments:
      aha
      bebe
      coucou
```

CONTROLE

Préprocesseur

Le préprocesseur est un programme qui traite le code source avant son passage par le compilateur C. Ce prétraitement permet des facilités pour le programmeur telles 7que définition de constantes, de macros, compilations conditionnelles et inclusions de fichiers.

En fait quand on lit un programme C, un grand nombre d'instructions n'est pas destinée au compilateur, mais au préprocesseur, qui transformera le source selon les directives le concernant, et passera ensuite un source C «pur» au compilateur.

Le préprocesseur s'occupe aussi de supprimer les commentaires.

Ces directives, tout d'abord, sont précédées du caractère '#' et ne se terminent pas par un point-virgule. Détaillons-les:

#define

L'instruction **#define** permet de définir des constantes.

```
#define     PI    3.14

      float x = PI;
```

En fait le préprocesseur en rencontrant la directive ci-dessus remplacera dans le source toutes les occurences de la chaîne de caractère «PI» par la chaîne «3.14». On voit qu'aucun contrôle de type ou autre ne pourra être fait sur «PI» par le compilateur car il ne verra même pas cette chaîne. C'est réellement une réécriture.

De ce fait **#define** ne sert pas qu'a définir des constantes mais aussi des macros, en effet on peut écrire (avec un backslash pour continuer sur la ligne suivante):

```
#define LONGUEINSTRUCTION     for (i=0; i<100; i++) \
                              {      \
                                     printf(«........»); \
                                     scanf(«....»,&x);\
                                     traite(x);\
                              }
```

Ensuite, on peut inserer la longue instruction n'import où dans notre programme

```
      instruction();
      LONGUEINSTRUCTION
      instruction();
```

La directive **#define** peut également s'utiliser avec des paramètres, on parle alors de macros avec paramètres.

```
#define     max(x,y)    (x<y) ? y : x

long x1,x2;
float y1,y2;
      long x = max(x1,x2);
      float y = max(y1,y2);
```

On constate alors que la magie de la réécriture permet d'utiliser la pseudo-fonction «max» avec plusieurs types différents sans devoir la réécrire soi-même pour chaque type.

La réécriture peut toutefois se révéler dangereuse, en effet avec:

```
#define     carre(x)    x * x
```

l'instruction

```
      while(i<10) printf(«%d\n»,carre(i++));
```

sera réécrite comme

```
      while(i<10) printf(«%d\n»,i++ * i++);
```

et l'on voit que l'incrémentation se fait deux fois par boucle, ce qui n'est surement pas l'effet désiré.

Un autre effet de bord se produit si l'on défini:

```
#define     suivant(x)  x + 1
```

l'instruction

```
      x = y * suivant(2);
```

sera réécrite

```
      x = y * 2 + 1;
```

ce qui n'est pas non plus ce que l'on désirait. Pour éviter ce genre d'erreur il très vivement conseillé de rajouter des parenthèses et de définir les macros comme suit:

```
#define     suivant(x)  (x + 1)
```

ce qui ne coûte rien et supprime les problèmes listés ci-haut.

#undef

Il est possible de supprimer une définition que l'on utilisait que dans une partie du code avec l'instruction **#undef**.

#ifdef

L'instruction **#ifdef** est très utile pour compiler un programme dans des conditions changeantes, ou sur des plateformes différentes.

```
#define     TEST
#ifdef      TEST
      instructions();
#endif
```

#ifdef teste si la définition de `TEST` existe auquel cas le préprocesseur inclut dans le code destiné au compilateur les instructions suivantes jusqu'à un **#endif.**

On constate aussi que **#define** peut être utilisé sans valeur associée.

Il est possible aussi d'utiliser **#elseif** :

```
#define TEST
#ifdef TEST
      instructions();
#elseif
      autre_instructions();
#endif
```

Enfin, ces inclusions conditionnelles peuvent être m
.

#ifndef

On peut également tester l'inexistance d'une définition avec **#ifndef**.

DECOMPOS

Exemples:

1) Pour maintenir un programme en mode debug ou final, on peut mettre les instructions concernant le debugging entre **#ifdef** et **#endif** et on compilera en définissant ou non le mot-clé associé:

```
#define _DEBUG

      instructions_du_programme();
#ifdef _DEBUG
      instructions_de_debugging();
#endif
      instructions_du_programme();
      .
      .
      .
```

2) Pour maintenir du code pouvant tourner sur différentes machine on place entre **#ifdef** et **#endif** les instructions concernant la machine désirée:

```
#defineAMIGA

      instructions_du_programme();
#ifdefTURBOC
      instructions_turboC();
#endif
#ifdefAMIGA
      instructions_amiga();
#endif
#ifdefMACINTOSH
      instructions_macintosh();
#endif
      instructions_du_programme();
      .
      .
      .
```

#include

L'instruction **#include** que vous avez sûrement déja utilisée est en fait une directive pour le préprocesseur, qui ne fait qu'inclure le contenu du fichier désiré dans le code destiné au compilateur.

Si le nom de fichier est entouré de 'crochets pointus' (`<...>`), le fichier est recherché dans un répertoire spécial contenant tous les fichiers '.h' du système.

Si le nom est entre guillemets doubles (`"..."`), le fichier est cherché dans le répertoire courant.

DECOMPOS

Modularisation

La modularisation est une technique vivement conseillée quand on développe de gros programmes, ou quand on veut réutiliser des portions de code. En fait le langage C est conçu pour être utilisé avec des modules. En effet, contrairement à Pascal qui possède ses procédure d'entrée sortie (**writeln**, **readln**), C ne propose qu'un jeu d'instructions très restreint. Toutes les fonctions que l'on utilise proviennent d'une multitude de modules stockés en librairies.

Quand vous utilisez **printf**() vous devez d'abord inclure le fichier 'stdio.h' dans lequel est déclarée la fonction **printf**(), puis à l'édition de lien le module contenant **printf**() est lié au programme exécutable.

Un module est en fait un programme C dans lequel on ne trouve pas la fonction **main**(). C'est une collection de fonctions que tout autre module peut utiliser.

Quand on développe une application on compile les modules séparément, ce qui permet de gagner du temps, et ils sont réunis à l'édition de lien en un seul programme. Un seul module doit alors posséder une fonction **main**().

Comme les fonctions sont alors séparées, si l'on veut utiliser une fonction provenant d'un autre module, il faut alors inclure un fichier de définition pour le module désiré à l'endroit où on en a besoin, à l'aide d'une directive **#include** afin que le compilateur sache que cette fonction existe et charge l'éditeur de liens de «lier» le code de la fonction à son appel. On comprend alors le terme d'édition de lien.

Conventions

On stocke les fichiers source contenant les définitions des fonctions du module avec un nom se terminant par '.c'.

On stocke les déclarations de ces fonctions dans un fichier du même nom mais se terminant par '.h' (header = en-tête en anglais).

Si le module travaille avec des structures de données privées que le reste du programme n'a pas besoin de connaître, elles peuvent être définies dans le fichier d'extension '.c'. Si par contre les structures de données doivent être connues du reste du programme, elles seront définies dans le '.h'.

Le fichier d'en-tête '.h' sera inclus dans le fichier '.c' par une directive `#include` si une structure de données doit être partagée ou si les fonctions contiennent des références circulaires.

Le compilateur produit alors des fichiers de code relogeable se terminant par '.o'.

L'éditeur de liens prend alors tout les '.o' et les réunit en les modifiant pour qu'ils communiquent correctement et produit un '.exe'.

Exemples

On a besoin dans plusieurs modules des fonctions **carre**() et **cube**(), on crée alors un module que l'on nomme *mathmod.c* dans lequel on implante nos fonctions:

```
float carre(x)
float x,
(
      return( x * x );
)

float cube(x)
float x,
(
      return( carre(x) * x );
)
```

Puis on crée un fichier nommé *mathmod.h* dans lequel on déclare (de la même manière que pour une référence en avant) uniquement les fonctions qui devront être visible depuis l'extérieur du module.

```
float carre();
float cube();
```

Maintenant, un autre module voulant utiliser **carre**() ou **cube**() devra inclure au début le fichier *mathmod.h* et lier le module *mathmod.o* produit par le compilateur avec son programme.

```
#include "mathmod.h"

      float x = cube(2);
```

Librairies

Généralement, quand on a une grande quantité de modules ne nessécitant plus d'être recompilés, on regroupe tout les '.o' dans une librairie (fichier.lib) et l'on indique à l'éditeur de lien d'inclure cette librairie. Toutes les fonctions standard en C sont regroupées dans la librairie 'c.lib' qui est automatiquement inclue par l'éditeur de lien, ainsi on a l'impression que ces fonctions sont prédéfinies dans le langage, c'est la grande force de C.

DECOMPOS

tcc est un compilateur C équivalent à celui qui est intégré dans l'environnement Turbo C. Il peut être appelé à l'invite du DOS.

tcc permet également de faire l'édition de liens et la création de code exécutable.

Les fichiers à compiler (et à lier) sont tapés sur la ligne de commande du DOS, accompagnés par les très nombreuses options qui permettent de paramétrer la compilation. En fait, tous ce qu'il est possible de faire dans les menus de Turbo C est aussi possible avec les options de tcc.

Syntaxe de l'appel de tcc

```
tcc {options} {fichiers}
```

Les accolades signifiant qu'il peut y avoir un nombre quelconque d'options et de fichiers.

La liste de toutes les options peut être obtenue en tapant siplement "tcc" à l'invite du DOS.

Fichiers à compiler

Les fichiers dont l'extension est

- **.obj** et **.lib** sont inclus lors de l'édition de liens.
- **.c** sont compilés puis liés si nécessaire.
- **.asm** sont assemblés puis liés.

Options

Les options sont spécifées par un '-', suivi d'une lettre et éventuellement d'une valeur. Elles sont regroupées en plusieurs classes

- **modèle de mémoire**, petit, grand, ...
- **macro-définitions**, semblables aux directives #define du préprocesseur
- **caractéristiques du code généré**, type d'arithmétique, processeur
- **optimisation** de la taille, de la vitesse, etc.
- **caractéristiques du code source**, ANSI, commentaires imbriqués,...
- **gestion des erreurs**, nombre, type, etc.
- gestion des segments
- contrôle de la compilation

Dans la plupart des cas, les valeurs par défaut utilisées par le compilateur sont correctes pour compiler des programmes simples. Voici toutefois celles qui peuvent être intéressantes.

Définition d'une constante
Une option de la forme

`-D`*`ident`*`[=`*`chaîne`*`]`

est équivalente à la ligne

`#define` *`ident chaîne`*

au début du premier fichier. Cette option est très utile avec le mécanisme de compilation conditionnelle, elle permet par exemple de mettre un

`-Ddebug`

en phase de déverminage et de ne plus en mettre quand le programme est fini. Si on prend soin de mettre les parties spéciales pour le déverminage (`printf()`, etc.) entre `#ifdef debug` et `#endif`, le programme final ne contiendra plus les bouts de code exécutable correspondants, mais les instructions seront toujours dans le code source, au cas ou des opérations de maintenance seraient nécessaires.

Suppression de l'édition de liens
L'option **`-c`** permet de supprimer l'édition de liens, on l'utilise pour compiler une librairie ne contenant pas de fonction `main()`.

Spécifier un nom pour le programme objet
Une option de la forme

`-o`*`nom_fichier`*

indique au compilateur que le fichier objet généré doit se nommer *`nom_fichier`*`.obj`.

Création de code assembleur
Une option instructive, qui permet de voir le code assembleur généré à partir du programme C originale est **`-S`**.

Spécifier un nom pour le programme exécutable
Une option de la forme

`-e`*`nom_fichier`*

indique au compilateur que le fichier exécutable généré doit se nommer

`nom_fichier``.exe`.

Options d'optimisation

- **`-G`** permet de demander une optimisation de la vitesse d'exécution, parfois au détriment de la taille du code exécutable généré.
- **`-O`** permet d'optimiser la taille du code exécutable.

Maintenance de programmes

Quand on a déjà quelques modules sur lesquel on travaille simultanément il devient vite difficile de se souvenir de toutes les modifications que l'on effectue et donc des modules devant être recompilés. Pour cela TurboC propose une gestion de projet qui est en fait basé sur un utilitaire courant dans Unix: **make**.

L'utilitaire **make** est un programme gérant des dépendances entre fichiers (qui peuvent être de toutes sortes) et permet d'exécuter des actions quand un fichier à été modifié. **make** fonctionne selon le principe qu'un fichier dépendant d'un autre doit être retraité (pour nous recompilé) si sa date de dernière modification est plus ancienne que celle du fichier dont il dépend.

Les définitions décrivant les dépendances et les actions se regroupent dans un fichier couramment nommé 'makefile' dans le répertoire où se trouve le projet en question.

Exemple

On a un module 'mathmod' comprenant les fichiers 'mathmod.c' et 'mathmod.h', un module 'blabla' comprenant les fichiers 'blabla.c' et 'blabla.h' et utilisant des fonctions de 'mathmod' et un fichier 'main.c' contenant un programme utilisant les fonctions de 'blabla' et de 'mathmod'.

Pour exprimer que 'main.o' dépend de 'main.c' et qu'il doit être regénéré si 'main.c' change on écrit dans le makefile:

```
main.o : main.c          # main.o dépend de main.c
      tcc -c main.c      # l'action à effectuer si main.c change
```

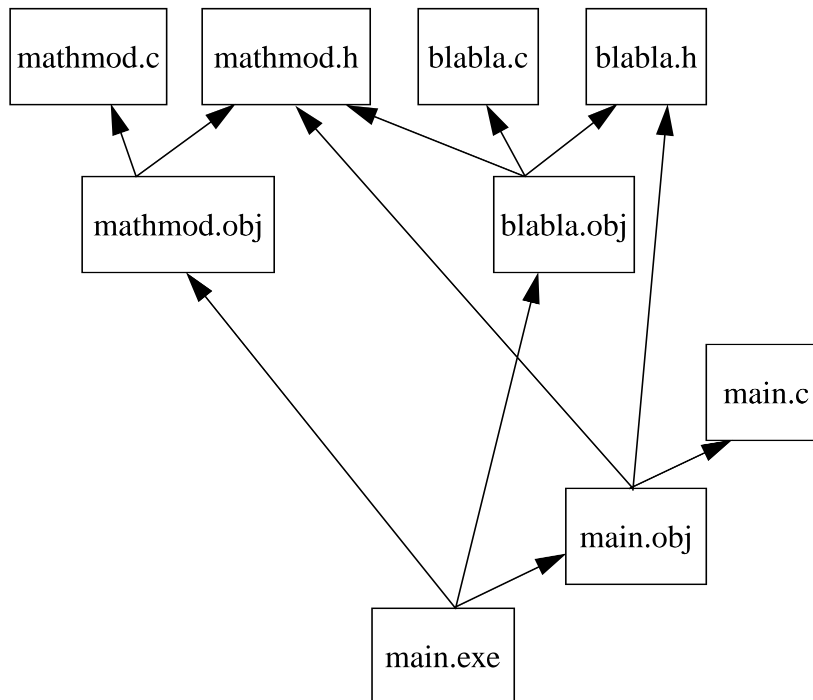
Une dépendance doit commencer sur le premier caractère d'une ligne et la ou les actions à exécuter doivent suivre directement cette ligne et commencer par une tabulation.

Pour notre exemple nous écrirons donc le makefile suivant:

```
main.exe : main.obj blabla.obj mathmod.obj
      tcc -emain.exe main.obj blabla.obj mathmod.obj

main.obj : main.c blabla.h mathmod.h
      cc -c main.c

blabla.obj : blabla.c blabla.h mathmod.h
      cc -c blabla.c

mathmod.obj : mathmod.c mathmod.h
      cc -c mathmod.c
```

DECOMPOS

Il correspond au **graphe de dépendances** suivant:



Les sommets de ce graphe sont les fichiers composant l'application, les arcs indiquente que le sommet de départ doit être régénéré si le sommet d'arrivée a été modifié.

On constate que chaque module dépend de son '.c' et de son '.h' mais qu'un module ne dépend que du '.h' d'un autre, en effet si on modifie le contenu d'un module sans toucher à sa définition d'interface il est inutile de recompiler les modules qui en dépendent. Si, par contre, on modifie un '.h' il est alors nessécaire de recompiler tout les modules l'utilisant afin de voir si l'utilisation qui en est faite est cohérente.

Il est possible d'utiliser des dépendances sans actions ou avec des actions autres que des compilations, ou même des dépendance ne dépendant de rien, dont les actions seront toujours exécutées:

```
nettoyer : enlever_objets enlever_executables

enlever_objets :
      del *.o
      del *.obj

enlever_executables :
      del *.exe
```

DECOMPOS

Pour utiliser un makefile il suffit de taper ‘make’ dans le DOS et la première dépendance trouvée dans le makefile sera testée. On peut aussi tester n’importe quelle dépendance en tapant:

```
make nettoyer
```

L’utilitaire ‘make’ permet aussi de déclarer des variables que l’on utilise avec un ‘$’ devant:

```
MAIN=mon_programme_principal

$MAIN.obj : $MAIN.c $MAIN.h
      cc -c -o$MAIN.obj $MAIN.c
```

Les projets de Turbo C

Turbo C propose un autre moyen, plus simple mais plus limité de gérer une application composée de plusieurs fichiers.

Il suffit de créer un fichier d’extension ‘**`.prj`**’ contenant la liste des noms des fichiers composant l’application (un nom de fichier par ligne).

Ensuite, il faut donner le nom du projet, correspondant au nom du fichier d’extension ‘**`.prj`**’ dans le menu

```
Project -> Project name
```

Pour compiler l’application, il suffit de sélectionner le menu

```
Compile -> Build all
```

une fois que tous les fichiers sont prêts. En cours d’élaboration, les fichiers peuvent être compilés sans que l’édition de liens n’ait lieu avec le menu

```
Compile -> Compile to OBJ
```

DECOMPOS

PRIORITES DES OPERATEURS

Nous avons vu que le langage C dispose d’un jeu très complet d’opérateurs, pour simplifier l’écriture d’expressions complexes, chaque opérateur possède une **priorité** et une **associativité**. Ainsi, on peut souvent se passer de parenthèses si on connaît les règles du tableau suivant.

Table 1: Priorités est associativités des opérateurs

<table>
<tr><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td><code>() [] -> .</code></td><td>⇒</td></tr>
<tr><td><code>! ~ ++ -- -(unaire) *(indirection)</code>
<code>&(adresse) ()(conversion) sizeof</code></td><td>⇐</td></tr>
<tr><td><code>* / %</code></td><td>⇒</td></tr>
<tr><td><code>+ -</code></td><td>⇒</td></tr>
<tr><td><code><< >></code></td><td>⇒</td></tr>
<tr><td><code>< <= > >=</code></td><td>⇒</td></tr>
<tr><td><code>== !=</code></td><td>⇒</td></tr>
<tr><td><code>&</code></td><td>⇒</td></tr>
<tr><td><code>^</code></td><td>⇒</td></tr>
<tr><td><code>|</code></td><td>⇒</td></tr>
<tr><td><code>&&</code></td><td>⇒</td></tr>
<tr><td><code>||</code></td><td>⇒</td></tr>
<tr><td><code>?:</code></td><td>⇐</td></tr>
<tr><td><code>= += -= *= /= %= >>= <<= &= |= ^=</code></td><td>⇐</td></tr>
<tr><td><code>,</code></td><td>⇒</td></tr>
</table>

Les opérateurs les plus **haut** dans la table sont évalués en premier dans l’expression.

Lorsqu’il y a plusieurs opérateurs de même niveau à évaluer, l’**associativité** indique dans quel ordre ils seront évalués.

⇐ l’évaluation se fait de droite à gauche
⇒ l’évaluation se fait de gauche à droite

Classification de caractères:

`isalnum`	vrai si c est une lettre ou un chiffre
`isalpha`	vrai si c est une lettre
`isdigit`	vrai si c est un chiffre entre 0 et 9
`isgraph`	vrai si c est un affichable (mais pas un espace)
`islower`	vrai si c est une lettre minuscule
`isprint`	vrai si c est affichable
`isspace`	vrai pour espace, tab, CR, LF, vtab, FF
`isupper`	vrai si c est une majuscule
`isxdigit`	vrai si c est un chiffre ou une lettre entre A et F ou a et f

Manipulations de répertoires

`chdir`	change le répertoire courant
`findfirst, findnext`	trouver un fichier sur disque
`fnmerge`	fabriquer un nom de fichier
`fnsplit`	decomposer un nom de fichier en disque, repertoires, nom et extension
`getcurdir`	donne le répertoire courant pour un certain disque
`getcwd`	donne le répertoire courant
`getdisk`	donne le lecteur courant
`mkdir`	crée un nouveau répertoire
`mktemp`	crée un nouveau fichier, de nom inexistant (commence avec AAA.AAA)
`rmdir`	supprime un répertoire vide, pas utilisé en ce moment
`searchpath`	cherche un fichier dans le répertoire courant et dans les répertoires de recherche du DOS (path)
`setdisk`	change le lecteur courant

Commandes de processus

`abort`	termine le programme en cours, avec la valeur 3 comme code de sortie
`exec...`	famille de fonctions permettant d'exécuter un autre programme, le programme en cours se termine
`exit`	termine le programme en cours, avec une valeur au choix

LIBRARIES

`raise`	envoye un signal, une gestion de signaux doit être installée par le programme en cours
`signal`	spécifie une fonction à exécuter lorsqu'un certain signal est "levé" (= interruption)
`spawn...`	famille de fonctions permettant d'exécuter temporairement un autre programme, la valeur retournée est la valeur donnée par le "fils" à la fonction exit()
`system`	permet de lancer une commande DOS depuis l'intérieur du programme

Conversions

`atof`	chaîne -> nombre float
`atoi`	chaîne -> nombre int
`atol`	chaîne -> nombre long
`ecvt, fcvt, gcvt`	nombre float -> chaîne
`itoa`	nombre int -> chaîne (base à choix)
`ltoa`	entier long -> chaîne (base à choix)
`strtod`	chaîne -> nombre double
`strtol`	chaîne -> entier long (base à choix)
`strroul`	idem mais unsigned
`tolower`	caractère -> minuscules
`toupper`	caractère -> majuscules
`ultoa`	entier long non signé en chaîne avec choix de la base

Dépistage d'erreurs

`assert`	évalue une condition, montre l'expression et indique le fichier et la ligne avant de s'arrêter si l'expression n'est pas vérifiée
`perror`	imprime une chaîne de caractères donnée et ensuite un message d'erreur du système
`strerror`	retourne un pointeur sur une chaîne contenant un message d'erreur

Entrées-sorties

`access`	teste les possibilités d'accès à un fichier donné
`chmod`	change les droits d'accèes à un fichier
`fileno`	retourne l'identificateur de fichier (int) d'un flux (FILE *)

`filelength`	retourne la longueur d'un fichier
`getftime`	permet de connaître la date de création ou de modification d'un fichier
`getpass`	affiche une chaîne et lit un mot sans le montrer à l'écran
`ioctl`	gestion de périphériques
`kbhit`	teste si un caractère a été frappé, n'attend pas
`remove, unlink`	supprime un fichier
`rename`	renomme un fichier
`setftime`	modifie la date et l'heure de création d'un fichier
`setmode`	place un fichier en mode binaire ou en mode texte
`sprintf, sscanf`	écriture et lecture dans une chaîne de caractères
`tmpfile`	crée un fichier qui est automatiquement effacé lorsqu'on le ferme ou que le programme se termine
tmpnam	crée un nom de fichier nouveau

Interface avec DOS, BIOS, 8086

`absread, abswrite`	lire ou écrire directement sur un disque
`bdos, bdosptr`	appel système au DOS
`bioscom`	I/O sur port série
`biosdisk`	I/O sur disque
`biosequip`	teste le matériel présent
`bioskey`	interface clavier
`biosmemory`	taille de la mémoire
`biosprint`	gestion des imprimantes
`biostime`	gestion de l'horloge interne
`ctrlbrk`	permet de spécifier une fonction lancée quand la combinaison de touches CTRL+BREAK est frappée, avant de termier le programme
`disable, enable`	interdit ou autorise les interruptions
`geninterrupt`	génère une interruption
`getcbrk, setcbrk`	teste ou décide si le CTRL+BREAK est possible ou pas
`getdfree`	donne l'espace libre sur un disque
`getvect`	lire un vecteur d'interruption
`getverify, setverify`	vérification des écritures sur disque
`inport, inportb`	lit un mot ou un byte sur un port d'entrée matériel
`int86, int86x, intdos, intdosx`	interruptions

`keep`	termine le programme et le laisse résident
`outport, outportb`	ecrit un mot ou un byte sur un port matériel
`peek, peekb`	lit un mot ou un byte dans la mémoire
`poke, pokeb`	écrit un mot ou un byte en mémoire
`setvect`	initialise un vecteur d'interruption
`sleep, delay`	suspend l'exécution pendant un certain nombre de secondes ou millisecondes

Manipulation de chaînes et de mémoire

`memccpy, memcpy, mmove`	copier un bloc de n octets
`memchr`	rechercher un caractère dans un bloc de n octets
`memcmp`	comparer deux blocs de n octets
`memicmp`	comme memcmp, mais ignore la différence entre majuscules et minuscules
`memset, setmem`	initialise un bloc de n octets avec une valeur donnée
`movedata, movmem`	copier un bloc de n octets
`strcpy, stpcpy`	recopier une chaîne dans une autre
`strcat`	mettre deux chaînes bout à bout
`strchr`	chercher la première occurence d'un caractère dans une chaîne
`strcmp`	comparer deux chaînes
`strcmpi, stricmp`	comparer deux chaînes sans tenir compte des majuscules et des minuscules
`strcspn`	chercher un segment de chaîne qui ne contienne pas un ensemble de caractères donné
`strdup`	duplication d'une chaîne (réserve l'espace nécessaire à la nouvelle chaîne)
`strlen`	retourne la longueur d'une chaîne
`strlwr`	met une chaîne en minuscules
`strncat`	comme strcat, mais avec un nombre fixe de caractères qui est ajouté
`strncomp`	comme strcmp mais seulement sur les n premiers caractères
`strncmpi, strnicmp`	fusion de strncmp et strcmpi
`strncpy`	copie un nombre donné de caractères d'une chaîne dans une autre
`strnset`	initialise un certain nombre de caractères d'une chaîne par un caractère donné
`strpbrk`	cherche dans une chaîne le premier caractère appartenant à un ensemble donné
`strrchr`	cherche dans une chaîne la dernière occurence

	d'un caractère donné
`strrev`	inverse une chaîne
`strset`	remplace tous les caractères d'une chaîne par un caractère donné
`strspn`	cherche la première sous-chaîne qui ne contienne que des caractères d'un ensemble donné
`strstr`	cherche la première occurence d'une chaîne dans une autre
`strtok`	cherche des mots dans une chaîne, séparés par un ensemble de séparateurs donné
`strupr`	met la chaîne en majuscules

Fonctions mathématiques

Tout ce que qu'il faut pour rendre un mathématicien heureux...

Allocation mémoire

`malloc, calloc`	allocation dynamique de mémoire
`coreleft`	retourne la taille de la mémoire disponible pour le programme inutilisée
`free`	libérer la mémoire allouée dynamiquement
`realloc, setblock`	réajuster un bloc de mémoire

Et les mêmes avec le prefixe far pour des modèles de mémoire étendus

Son

`sound`	biiiiiiiip!
`nosound`	à faire une fois après avoir appelé la fonction sound()

Tri et recherche

`bsearch`	recherche dichotomique dans un tableau
`lfind, lsearch`	recherche linéaire dans un tableau
`qsort`	tri (très efficace)

Nombres aléatoires

`rand, random`	retourne un nombre entier aléatoire

`randomize, srand`	initialise le générateur de nombres aléatoires

Date et heure

`asctime, ctime`	convertit une date et une heure en chaîne de caractères
`clock`	permet de détermier un intervalle de temps entre deux événements (depuis le début du programme)
`difftime`	calcule le temps qui s'est écoulé entre deux instants
`dostounix, unixtodos`	passer des formats d'heure UNIX <-> DOS
`getdate`	retourne la date du système
`gettime`	retourne l'heure du système
`setdate`	fixe la date
`settime`	fixe l'heure
`stime`	fixe la date et l'heure
`time`	donne la date et l'heure

Aussi une gestion compliquée des décalages horaires (gmtime, localtime, tzset)

Graphisme

`arc`	tracer un arc de cercle
`bar`	tracer un rectangle (barre)
`bar3d`	tracer une barre en 3 dimensions (perspective)
`circle`	tracer un cercle
`cleardevice`	efface tout l'écrangraphique
`clearviewport`	efface la fenêtre graphique courante
`closegraph`	referme le système graphique
`detectgraph`	tester le matériel graphique installé
`drawpoly`	tracer une ligne polygonale
`ellipse`	tracer un arc d'ellipse
`fillellipse`	tracer une ellipse remplie
`fillpoly`	trace un polygone et le remplit
`floodfill`	remplit une zone fermée
`getarccoords`	trouver les coordonnées du dernier arc tracé avec la fonction arc()
`getaspectratio, setaspectratio`	gestion de la "forme" des pixels
`getbkcolor, setbkcolor`	couleur courante du fond
`getcolor`	retourne la couleur de tracé courante
`getdefautpalette`	définition de palette par défaut
`getdrivername`	nom du gestionnaire courant
`getfillpattern, setfillpattern`	motifs de remplissage

`getfillsettings`	motifs et couleur de remplissage courants
`getgraphmode`	donne le mode graphique courant
`getimage`	sauvegarde en mémoire une portion d'écran
`getlinesettings`	style, motif, épaisseur des lignes courants
`getmaxcolor`	plus grand numéro de couleur pouvant être passé à setcolor()
`getmaxmode`	plus grand numéro de mode pour le gestionnaire graphique courant
`getmaxx, getmaxy`	plus grands x et y possibles dans l'écran courant
`getmodename`	nom d'un mode graphique donné
`getmoderange`	modes graphiques disponibles pour un gestionnaire donné
`getpalette, setpalette`	gestion de la palette de couleurs
`getpalettesize`	nombre de couleurs de la palette dans le mode courant
`getpixel`	donne la couleur d'un certain pixel
`gettextsettings`	paramètres courants du texte en mode graphique: fonte, direction, taille, justification
`getviewsettings`	fenêtre graphique courante
`getx, gety`	position courante en x et y
`graphdefaults`	réinitialise les paramètres aux valeurs par défaut
`grapherrormsg`	chaînes de caractères associées aux erreurs de graphresult()
`_graphfreemem, _graphgetmem`	gestion de la mémoire graphique
`graphresult`	gestion des erreurs
`imagesize`	nombre d'octets nécessaires à la mémorisation d'une image
`initgraph`	initialisation du système graphique, avec détection possible du matériel à disposition
`installuserdriver`	gestionnaire d'écran personnalisé
`installuserfont`	police non inclue dans le système BGI
`line`	trace une ligne
`linerel, lineto`	trace une ligne par rapport au point courant
`moverel, moveto`	déplace le point courant
`outtext`	affiche du texte en mode graphique, à la position courante
`outtextxy`	affiche du texte n'importe où dans l'écran graphique
`pieslice`	trace et remplit un secteur angulaire
`putimage`	copie une image de la mémoire sur l'écran
`putpixel`	allume un pixel à la couleur voulue
`rectangle`	trace un rectangle (vide)
`registerbgidriver`	gestion fine des pilotes d'écran

`registerbgifont`	gestion fine des polices de caractères
`restorecrtmode`	revient avant initgraph
`sector`	trace et remplit un secteur angulaire d'ellipse
`setactivepage, setvisualpage`	gestion des pages d'écran multiples
`setallpalette`	change toutes les couleurs de la palette
`setcolor`	choisir une couleur
`setfillstyle`	motif de remplissage
`setgraphbufsize`	gestion du tampon interne
`setgraphmode`	sélectionne un mode graphique et efface l'écran
`setlinestyle`	épaisseur et motif de tracé des lignes
`setrgbpalette`	gestion des couleurs
`settextjustify`	justification du texte (haut, bas, gauche, droite, centré)
`settextstyle, setusercharsize`	caractéristiques du texte
`setviewport`	fenêtre graphique courante
`setwritemode`	mode de superposition des lignes
`textheight, textwidth`	hauteur et largeur du texte

Remarques

Les fonctions présentées plus en détail allieurs dans le cours ne sont par répétées ici, par exemple:

Gestion de fichers, entrées-sorties, gestion de l'écran en mode texte...

En général, il faut consulter un manuel, pour connaître le fonctionnement exact de la fonction qui semble faire l'affaire, ou alors, utiliser l'aide (F1) de TURBO C.

Sous UNIX, les pages de manuel peuvent être obtenues en tapant

```
man nom_fonction
```

Par ailleurs, quand vous utilisez une de ces fonctions (ou macros), réfléchissez bien au problèmes de portabilité qui pourraient avoir lieu si votre programme doit changer de système ou de compilateur...

TURBO C dispose de quelques fonctions spéciales de gestion de l'écran en mode texte. Ces fonctions sont les mêmes que celles de TURBO PASCAL.

Attention lors de l'utilisation de ces fonctions, elles ne font pas partie de la plupart des libraires standard de C. Un programme qui en fait usage ne pourra pas être compilé dans un autre environnement que TURBO C. Ces fonctions nuisent donc à la portabilité de vos programmes si vous les utilisez. Par contre, si votre programme n'est destiné qu'à tourner sur un PC, elles sont assez pratiques.

Souvenez-vous d'autre part qu'il faut toujours mettre des parenthèses lors d'un appel de fonction, même si elles sont vides. Ainsi, la procedure `clrscr` de TURBO PASCAL correspond à la fonction `clrscr()` de TURBO C qu'on appelle toujours avec ses parenthèses.

Pour utiliser les fonctions qui suivent, il faut inclure le **fichier de définitions** `conio.h` dans votre programme à l'aide l'instruction `#include <conio.h>`.

Définir une fenêtre: fonction window()

Cette fonction permet de définir une zone rectangulaire de l'écran en mode texte comme la fenêtre courante. Certaines actions qui peuvent être effectuées par la suite seront limitées à cette fenêtre. L'utilité est de définir plusieurs zones de travail indépendantes, par exemple le haut de l'écran pour lire des entrées de l'utilisateur et le bas pour écrire des résultats. Une telle séparation peut apporter une meilleure clarté à votre application.

Syntaxe

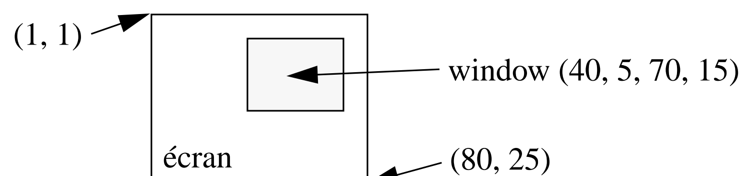
```
void window (int gauche, int haut, int droite, int bas);
```

Ce qui signifie que la fonction window prend quatre paramètre de type entier et ne retourne pas de valeur. Les quatre nombres sont les positions du coin supérieur gauche et du coin supérieur droit de la fenêtre par rapport aux coordonnées l'écran.

Le coin supérieur gauche de l'écran a les coordonnées (1, 1) et le coin inférieur droit (80, 25). Elle correspondent aux dimensions de la fenêtre par défaut.

Si les paramètres sont incorrects, par exemple largeur ou hauteur plus petite que 1 ou fenêtre dépassant les coordonnées de l'écran, l'appel de cette fontion est ignoré.

Exemple



LIBRARIES

Effacer la fenêtre texte: fonction clrscr()

Cette fonction efface tout ce qui est affiché dans le fenêtre texte courante et positionne le curseur dans son coin supérieur gauche (position (1,1) de la **fenêtre** courante).

Pour l'utiliser, il est aussi nécessaire d'inclure le fichier `conio.h`.

Attendre qu'une touche soit frappée: getch() et getche()

Habituellement, toutes les entrées au clavier doivent être validées par la touche RETURN. Il est parfois utile de lire une entrée de l'utilisateur directement, sans qu'il faille taper de RETURN. Ces fonctions servent à cela.

Syntaxe

```
char getche (void);
char getch (void);
```

Ces fonctions ne prennent pas de paramètre en entrée, elles attendent que l'utilisateur frappe une touche au clavier et retournent le caractère correspondant.

La différence entre les deux est que **`getche`** fait également l´**écho** dans la fenêtre courante alors que `getch()` ne le fait pas.

Exemple

```
char encore;
printf ("Encore une fois (o/n)? ");
encore = getche();
```

Lorsqu'une touche spéciale est frappée, comme par exemple une flèche du clavier ou une touche de fonction, **deux caractères** doivent être lus; le premier reçoit la valeur '`\0'`. Le second caractère reçoit un code correspondant à la touche pressée.

Codes pour les flèches

72	haut
77	droite
80	bas
75	gauche

Ce code se retrouve dans le secons caractère lu.

Pour tester si la touche RETURN a été pressée, tester `if (c=='\r') ...` .

LIBRARIES

Exemple

```
char c;
c=getch();
if (c=='\0') {
      c=getch();
      if (c==72) { la flèche vers la haut a été pressée }
}
```

Déplacer le curseur: gotoxy()

Cette fonction permet de placer le curseur à l’endroit désiré dans la fenêtre courante. Sa syntaxe est

```
void gotoxy (int colonne, int ligne);
```

Si la position spécifiée est incorrecte, par exemple en dehors de la fenêtre, l’appel est ignoré. Ce comportement est très utile, car il assure que l’on ne sort pas de la fenêtre inopinément.

Position du curseur: wherex() et wherey()

Ces deux fonction permettent au programme de connaître exactement la position du curseur dans la fenêtre de texte courante.

Syntaxe

```
int wherex (void);
int wherey (void);
```

`wherex()` retourne le numero de la colonne dans laquelle se trouve le curseur; `wherey()` le numéro de la ligne.

Exemples

```
printf ("Le curseur est à la position (%d, %d)\n", wherex(), wherey());

gotoxy (wherex(), wherey() - 1); /* monte le curseur d'une ligne */
```

Régler l’intensité de l’affichage

Les trois fonctions **`highvideo()`**, **`normvideo()`** et **`lowvideo()`** permettent de spécifier avec quelle intensité un caractère doit être affiché dans la fenêtre courante.

Syntaxe

```
void highvideo(void);
void normvideo(void);
void lowvideo (void);
```

Après un appel à `highvideo()`, tous les caractères seront affichés en **surbrillance** dans

LIBRARIES

la fenêtre de texte courante. La fonction `normvideo()` revient au **mode normal**. Enfin, `lowvideo()` sélectionne l'affichage des caractères en **faible intensité**.

Ecrire dans une fenêtre texte

L'écriture dans une fenêtre texte ne se fait pas avec les fonctions `printf()` et `puts()` habituelles. En effet, elles ne respectent ni les fenêtres de texte spécifiées, ni les modes d'affichage. Ces fonctions reviennent à la ligne au bout de l'écran et jusqu'à la marge. Par contre, elles respectent la position du curseur spécifiée par `gotoxy()`.

Il faut utiliser d'autres fonctions qui sont **`cprintf()`** et **`cputs()`**. Ces fonctions ont la même syntaxe que les fonctions habituelles; elles respectent par ailleurs les fenêtres, reviennent à la ligne automatiquement au bord, font un "scrolling" (défilement) seulement dans la fenêtre. Par contre, elles ne transforment pas le caractère d'interligne '`\n`' en séquence CR/LF "`\r\n`", ce qui a pour effet de faire descendre le curseur d'une ligne sans le ramener sur la marge de droite de la fenêtre texte courante.

Par ailleurs, j'ai pu constater une plus grande vitesse d'exécution pour la fonction `cprintf()` que pour `printf()`. Ce gain est probablement obtenu par un accès direct à la mémoire vidéo du PC.

La fonction

```
char putch (char c);
```

affiche le caractère que l'on lui passe en paramètre dans la fenêtre standard. Elle retourne le caractère si tout s'est bien passé et la constante prédéfinie `EOF` (End Of File) sinon. EOF est généralement le caractère de fin de fichier en C. Dans ce cadre, cela ne signifie pas grand chose, mais cette constante est retournée par quelques fonctions comme signal d'erreur.

La fonction `putch()` ne transforme pas non-plus le caractère '`\n`' en "`\r\n`". A cette différence près, elle est identique à la fonction **`putchar()`** qui est une fonction C standard compatible avec d'autres environnements et déclarée dans `stdio.h`.