INSA ROUEN

Génie Mathématique – 3ème année

Cours Langage C

(Volume 1)

KOTOWICZ JEAN-PHILIPPE

(jean-philippe.kotowicz@insa-rouen.fr)

VERSION 1.4

Table des matières

Chapitre 1 : Introduction

1. Bref historique

L'objectif était de développer un langage pour l'écriture d'un système d'exploitation le plus portable possible. Le langage C est donc lié à la conception du système UNIX par le laboratoire « Bell Laboratories » qui appartient à la société Western Electric Company filiale de AT&T.

Les langages ayant influencé son développement sont :

- le langage BCPL de M. Richards en 1967
- le langage B développé aux Bell-Labs en 1970.

Ces deux langages partagent avec le langage C :

- les structures de contrôle
- l'usage des pointeurs
- la récursivité

Ces deux langages étaient des langages non typés. Ils ne travaillaient que sur des données décrites par un mot machine et donc n'étaient absolument pas portables. Le langage C en introduisant des types de données tels que l'entier, ou le caractère évite ces écueils.

Les dates marquantes de l'histoire du langage C sont les suivantes :

- 1970 : diffusion de la famille PDP 11
- 1971 : début du travail sur le langage C, car le PDP 11 peut adresser un octet alors que son mot mémoire est de 2 octets, ce qui pose un problème vis-à-vis du langage B
- 1972 : la première version de C est écrite en assembleur par Brian W. Kernighan et Dennis M. Ritchie
- 1973 : Alan Snyder écrit un compilateur C portable (thèse MIT)
- 1975 : Steve C. Johnson écrit et présente le PCC (Portable C Compiler). C'était à l'époque le compilateur le plus répandu. Le PCC a longtemps assuré sa propre norme puisqu'il était plus simple de porter le PCC que de réécrire un compilateur C (25 % du code du PCC à modifier)
- 1987 : début de la normalisation du langage par l'IEEE avec constitution d'un comité appelé : X3 J-11
- 1989 : sortie du premier document normalisé appelé norme ANSI X3-159
- 1990 : réalisation du document final normalisé auprès de l'ISO : ISO/IEC 9899
- 1999 : première révision de la norme

Jusqu'en 1987, il n'y avait pas de norme, le livre "*The C Programming Language*" de B. W. Kernighan et D. M. Ritchie définit le langage. Ce livre contient une description précise du langage appelée "*C Reference Manual*".

2. Bibliographie sommaire

Kernighan & Ritchie, *Le Langage C*, Masson

Ph Dax, *Le langage C*, Eyrolles

J.M. Drappier, A. Mauffrey , *C par l'exemple*, Eyrolles

Ph Drix, *Langage C norme ANSI vers une approche orientée objet*, Masson

Claude Delannoy, *Programmer en langage C*, Eyrolles

Claude Delannoy, *Exercice en Langage C*, Eyrolles

3. Double aspect du C

3.1. Un langage de bas niveau

De par l'efficacité de son code, le langage C est un langage de bas niveau dons les instructions sont proches du langage assembleur (décalage à gauche, à droite, …).
Les formes syntaxiques sont aisément reconnaissables par le compilateur d'où un codage « immédiat » en langage machine.
De plus, par le biais du langage C, on a accès directement aux bits, octets, adresses, …

Le C exploite au maximum le système et notamment pour toutes les entrées/sorties qui font directement appel à la librairie système.

L'usage de pointeurs et une gestion de la mémoire absolument dévolue au programmeur complètent le tableau pour en faire définitivement un langage de bas niveau.

3.2. Un langage de haut niveau

Mais en même temps, plusieurs critères font que l'on peut également considérer le langage C comme un langage de haut niveau :

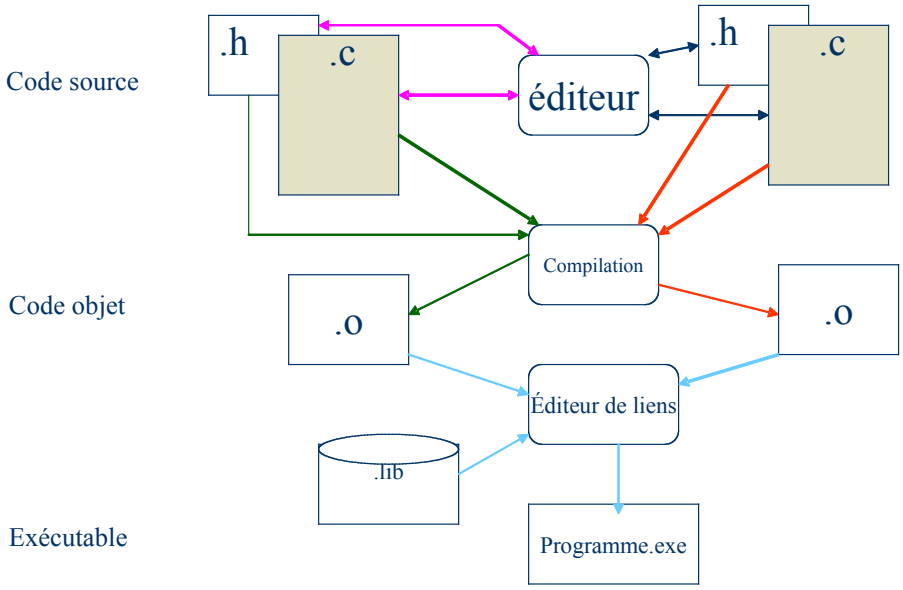
- Il est modulaire : il donne la possibilité de regrouper des fonctions/procédures/données dans des entités logiques et programmatiques.
- Il est portable : dans la mesure où les compilateurs ont tendance à respecter la norme ANSI
- Il est structuré en actions et dispose de toute la panoplie des instructions de la programmation structurée (si, tant que, etc.).
- Il est également structuré en données par l'utilisation de structures complexes telles que les enregistrements par exemple.
- Il est récursif

Et enfin, il est souple (voire permissif) : le compilateur ne fait aucun contrôle ou pratiquement pas. Il faut programmer en ayant beaucoup de rigueur.

Chapitre 2 : Présentation générale du C

1. Le programme C

Le langage C est un langage compilé.
Un programme C est donc un fichier exécutable résultant de la compilation d'un fichier source respectant la syntaxe du langage C.
En fait, un programme en C peut être constitué d'un ensemble de fichiers sources destinés à être compilés séparément et à subir une édition de liens commune. Ces fichiers sources sont aussi appelés modules et ce type de programmation est appelé programmation modulaire.



Chacun des fichiers source doit respecter la structure et la syntaxe du langage C.

2. Structure d'un programme C

Un fichier source est composé des éléments suivants :

- L'inclusion des fichiers d'entêtes
- Les déclarations de types
- Les déclarations de variables globales
- Les déclarations de fonctions
- le programme principal (fonction main)
- Les définitions des fonctions

La fonction main est le point d'entrée du programme et tout programme C doit en posséder un.
Toute variable utilisée dans un programme en C doit avoir été déclarée avant !

3. Généralités sur la syntaxe du C

3.1. Identifiants

Les identifiants sont utilisés pour donner des noms aux différentes entités utilisés dans le langage.

Un identifiant est construit à partir de l'alphabet : "a-z, A-Z, 0-9, _".
Il peut avoir jusqu'à trente et un caractères significatifs.
Mais il commence forcément par une lettre ou par le caractère *underscore* ('_').

3.2. Mots réservés

Les mots réservés sont les mots prédéfinis du langage C.
Ils ne peuvent pas être réutilisés pour des identifiants.

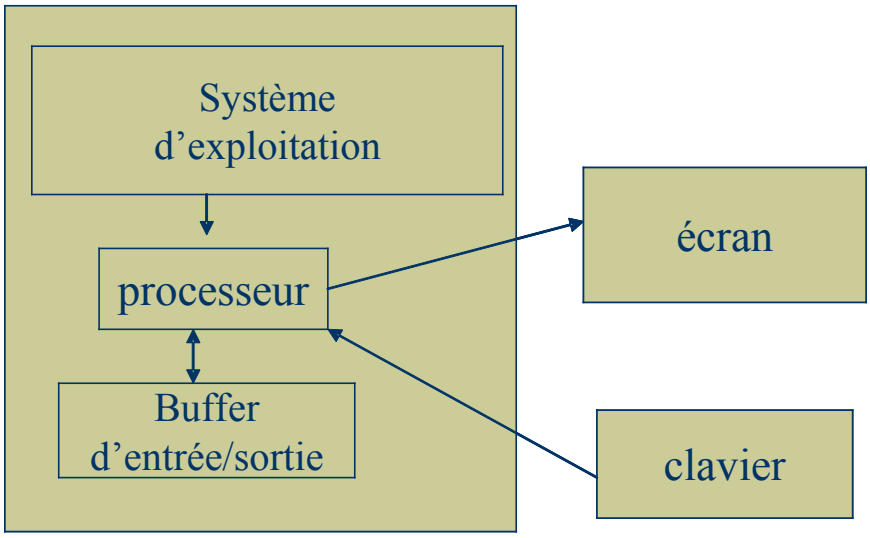
Il existe des mots réservés pour :

- La définition des données (*char const double float int long short signed unsigned void*)
- Les classes d'allocation (*auto extern register static*)
- Les constructeurs (*enum struct typedef union*)
- Les instructions de boucle (*do for while*)
- Les sélections (*case default else if switch*)
- Les ruptures de séquence (*break continue goto return*)
- Divers (*asm entry fortran sizeof*)

Chapitre 3 : Les entrées/sorties standards

1. Schéma général

Les entrée-sorties en langage C ne sont pas prises en charge directement par le compilateur mais elles sont réalisées à travers des fonctions de bibliothèque. Le compilateur ne peut pas faire de contrôle de cohérence dans les arguments passés à ces fonctions car ils sont de type variable. Ceci explique l'attention toute particulière avec laquelle ces opérations doivent être programmées.



L'écran est la sortie standard (*stdout* pour standard output), le clavier est l'entrée standard (*stdin* pour standard input).

2. Affichage à l'écran

La méthode C qui permet d'afficher quelque chose à l'écran est la fonction ***printf()***.
Cette fonction reçoit un nombre d'arguments variable qui seront transformés en une chaîne de caractères. Cette transformation fait appel à une notion de format.

printf(*chaîne de caractère comportant des formats****,*** *liste de variables à afficher****);***

Cette fonction affiche à l'écran la chaîne dans laquelle les formats ont été substitués aux valeurs correspondantes.

Le nombre de variables à afficher doit être égal au nombre de formats compris dans la chaîne de caractères.

Le format définit le type de la donnée et le mode de représentation de la donnée. Un format est indiqué par le signe **%** suivi du type de la donnée à afficher.

Les formats utilisables en C sont les suivants :

- Caractère ASCII : %c %6c
- Entier décimal : %d %6d %03d
- Entier non signé : %u %6u %03u
- Entier octal : %o %6o %03o
- Entier hexadécimal %x %6x %02x
- Virgule fixe : %.1f %10.2f
- Notation E : %.3e %10.6e

Lorsque % est suivi d'un entier x, l'élément à afficher sera affiché aligné à droite sur x caractères. Si cet entier est précédé d'un 0, l'alignement sera complété par des 0.
Dans le cas des réels, mettre un '**.**' suivi d'un entier après le % précise le nombre de décimales à afficher.

Exemple :
Si **x** vaut 3 et **car** est le caractère 'g',

```
printf ("j'affiche l'entier %d et le caractère %c." , x, car)
printf ("j'affiche l'entier %3d et le caractère %c." , x, car)
printf ("j'affiche l'entier %03d et le caractère %c." , x, car)
```

Affichera : **j'affiche l'entier 3 et le caractère g.**
j'affiche l'entier __3 et le caractère g.
j'affiche l'entier 003 et le caractère g.

Un certain nombre de caractères spéciaux peuvent également être insérés dans la chaîne de caractères du **printf** pour être affichés à l'écran.
Parmi ces caractères spéciaux, on trouve notamment :

- '\n' saut de ligne
- '\t' tabulation
- '\b' espace arrière
- '\r' retour chariot
- '\f' saut de page
- '\\' backslash
- '\" apostrophe

Deux autres fonctions permettent également un affiche à l'écran :

***fputs** (chaine);*

Cette fonction affiche à l'écran la chaîne de caractères ainsi qu'un saut à la ligne.

***int putchar (int** c);*

Cette fonction affiche à l'écran le caractère c.

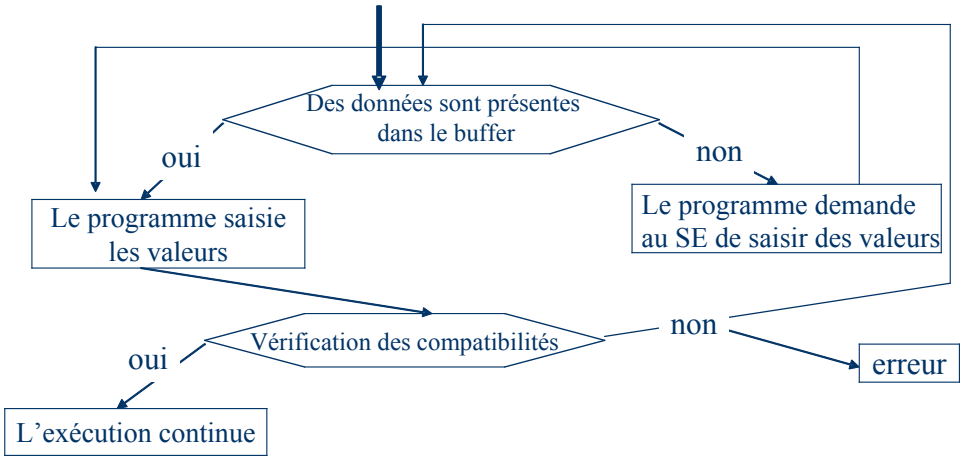
3. Saisie à partir du clavier

La fonction ***scanf()*** fait le pendant à la fonction ***printf()***. Elle permet de lire des valeurs sur le clavier.

***scanf** (format, liste des adresses où devrons être stockées les valeurs saisies);*

La saisie à partir du clavier se fait également par l'intermédiaire du *buffer*.

Ainsi, quand un programme désire saisir une donnée de l'utilisateur, le fonctionnement général de la saisie suit le schéma suivant.



Les formats du ***scanf*** sont les mêmes que pour le ***printf***, à savoir :

- %c lit un caractère
- %d, %hd, %ld lit un nombre décimal (entier)
- %o, %ho, %lo lit un nombre octal
- %x, %hx, %lx lit un nombre hexadécimal
- %f, %lf, %e, %le lit un nombre décimal (float)

Exemple :

```
scanf("%d", &i);  /* attend la saisie d'un entier */
scanf("%f", &j);  /* attend la saisie d'un réel */
scanf("%c", &c);  /* attend la saisie d'un caractère */
```

Le '**&**' qui précède le nom des variables indique que l'on travaille sur l'adresse de la variable et non directement sur sa valeur. Nous reviendrons plus précisément sur cet aspect dans le chapitre sur les pointeurs.

Deux autres fonctions permettent également un affichage à l'écran :

int getchar();

Cette instruction est une macro qui retourne le prochain caractère lu.

char * gets(char * s);

Cette fonction lit une chaîne depuis le flux d'entrée standard et la place dans s. La lecture se termine à la réception d'un retour chariot, lequel est remplacé dans s par un caractère nul (\0).

Remarque : gets peut être utile pour vider le buffer

Chapitre 4 : Les types de base

1. Numération

1.1. Représentation

Le système décimal :

- système de numération à base 10
- symboles utilisés : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ecriture d'un nombre décimal quelconque :

- $N = a_n.10^n + a_{n-1}.10^{n-1} + \ldots + a_0.10^0$
- $123 = 1.10^2 + 2.10^1 + 3.10^0$

Système à base quelconque :

- $N = a_n.b^n + a_{n-1}.b^{n-1} + \ldots + a_0.b^0$ avec $0 <= a_i < b$
- ou encore $N = a_n.a_{n-1} \ldots a_0$

Dans le cas du système binaire, les symboles autorisés sont 0 et 1.
$N = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_0 2^0$ avec $0 <= a_i < =1$

Exemple : $1010_2 = 1.2^3+0.2^2+1.2^1+0.2^0 = 10_{10}$

En interne, toutes les informations sont codées sous forme de bits et plus particulièrement par l'intermédiaire de mots composés d'un certain nombre d'octets contenant eux même 8 bits.

1.2. Les entiers

Un mot binaire de n bits permet de représenter des entiers naturels de 0 à 2^{n-1}.
Donc on peut représenter les entiers soit sur un octet, on peut ainsi manipuler les valeurs 0 à 2^8-1 à savoir de 0 à 255. Ou sur 2 octets : 0 … 2^{16}-1 à savoir de 0 à 65 535.

En ce qui concerne les entiers négatifs, l'idée de base est d'utiliser le premier bit comme bit de signe. On choisit 0 pour un nombre positif et 1 pour un nombre négatif.
Ainsi, sur un mot de 8 bits, on n'utilise que 7 bits pour coder un nombre positif, on est donc limité aux nombres signés compris entre -2^7-1 et 2^7-1 à savoir de -127 à 127.

Par convention, on choisira que $1000\ 0000_2 = -2^7 = -128$ ceci afin d'avoir un zéro unique.

Un nombre entier codé sur 16 bits permet donc de représenter un nombre appartenant à l'intervalle -32 768 … 32767.

Pour calculer l'entier correspondant à un mot binaire, la technique utilisée est celle du complément à 2. Il s'agit de faire le complément à 1 auquel on ajoute 1.

Exemple : que représente l'entier 131 ? $1000\ 0011_2$

On retire 1	0000 0001	
	1000 0010	
Complément à 1	0111 1101	on obtient 125_{10}

Puisque le 1er bit de 131 en binaire est 1, le nombre qu'il représente est négatif.
La valeur binaire de 131 représente donc l'entier -125.

A l'inverse, comment représenter l'entier -125 en binaire :

On écrit 125_{10} en binaire	$0111\ 1101_2$
Le complément à 1 donne	1000 0010
On ajoute 1	0000 0001
	1000 0011 soit 131_{10}

1.3. Les réels

Pour les réels, on utilise une arithmétique à virgule flottante, c'est-à-dire une forme exponentielle telle que :

SIGNE MANTISSE x $Base^{Exposant}$

Par exemple :
En base décimale :

5748,6954	$= 57486954 \times 10^{-4}$
	$= 0{,}57486954 \times 10^{4}$
	$= 5{,}7486954 \times 10^{3}$

En binaire :

1011,0111	$= 1011\ 0111 \times 2^{-4}$
	$= 0{,}1011011 \times 2^{4}$
	$= 1{,}0110111 \times 2^{3}$

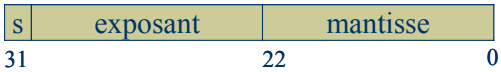
Pour utiliser au mieux la place disponible, la mantisse est normalisée de manière à éviter les zéros non significatifs, ce que l'on obtient en imposant 0,1<=mantisse<1, ce qui garantit que le premier chiffre significatif est toujours celui situé immédiatement après le point décimal.

Ainsi, la forme normalisée de l'exemple précédent est : $0{,}10110111 \times 2^4$

La représentation en machine sur 32 bits est la suivante :

- Un bit de signe : 0 pour + et 1 pour -.
- L'exposant sur 8 bits, des numéros 23 à 30, de -128 à +127.
- La mantisse est sur 23 bits, des numéros 0 à 22, et est normalisée (le premier bit étant obligatoirement à 1, il est donc ignoré).

Soit :



Exemple sur 32 bits :

Soit le nombre $37{,}625_{10}$, la forme binaire correspondante est 100101,1010
Après normalisation : $0{,}100101101 \times 2^6$
Si on décompose, on obtient : signe : 0
exposant : 6
mantisse : 100101101
L'exposant est $6 + 128 = 134_{10} = 10000110_2$ (afin d'éviter l'utilisation d'un bit de signe)

Le plus grand nombre réel que l'on peut obtenir est : 2^{127} pour l'exposant et la mantisse est proche de 1 d'où $2^{127} \sim 10^{38}$.

2. Les types simples

2.1. Les entiers

Les types simples concernant les entiers ont donc été définis en fonction du système de numération décrit précédemment et du nombre d'octets sur lesquels ils vont être codés.

On distingue donc les entiers (**int** en C) signés et non signés (**unsigned**) et les entiers courts (**short**) ou longs (**long**).
Le type caractère (**char**) est également codé en tant qu'entier. En C un caractère est en fait associé à son code ASCII (i.e. une valeur entière comprise entre 0 et 255).

Le tableau suivant recense les différents types du C basés sur les entiers et les valeurs qu'ils peuvent prendre. Les valeurs données ici sont valables pour des processeurs travaillant sur un mot machine de 16 bits.
La taille standard des entiers (**int**) est de 2 octets, mais dans les machines d'aujourd'hui, elle est codée sur 4 octets (32 bits).

Type	Taille	Rang		
unsigned char	1 octet	0	à	255
char	1 octet	-128	à	127
short	2 octets	-32 768	à	32 767
unsigned short	2 octets	0	à	65 535
unsigned int	4 octets	0	à	4 294 967 295
int	4 octets	-2 147 483 648	à	2 147 483 647
unsigned long	4 octets	0	à	4 294 967 295
long	4 octets	-2 147 483 648	à	2 147 483 647
unsigned long long	8 octets	0	à	18 446 744 073 709 551 615
long long	8 octets	-9 223 372 036 854 775 808	à	9 223 372 036 854 775 807

2.2. Les réels

En C, les types associés aux réels sont les types **float** et **double**.

Pour un processeur travaillant sur un mot machine de 16 bits, un **float** sera codé sur 4 octets et un double sur 8 octets.

2.3. Redéfinir un type simple

On peut redéfinir tous les types simples au moyen du mot clé *typedef*
Il s'agit en fait simplement de donner un nouveau nom à un type.
Bien entendu, après un **typedef**, l'ancien nom du type existe toujours.

La syntaxe est la suivante :

typedef ancien_type nouveau_type

On peut ainsi par exemple définir un type **entierCourt** par l'instruction :

typedef short int entierCourt ;

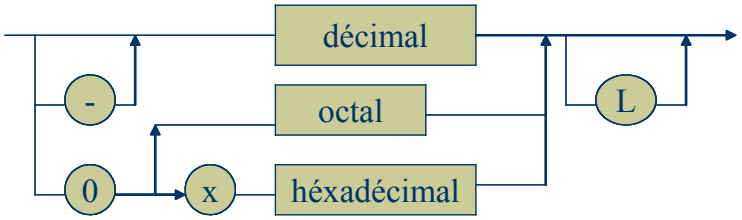
Puis ainsi déclarer des variables :

entierCourt x, y, z ;

3. Les constantes associées aux types simples

Les constantes entières acceptent tous les entiers de la forme suivante :

Constante_int



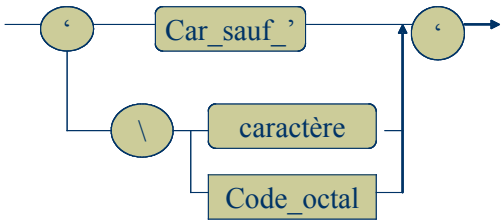
123 -12L

0377 0377L

0xFF 0XAB

Les constantes de caractères se construisent comme suit :

Constante_char



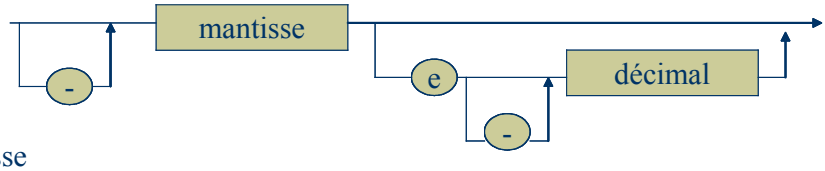
‘x’ ‘X’

‘\’’ ‘\\’ ‘\n’

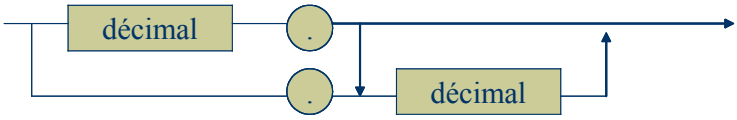
‘\0’ ‘\7’ ‘\33’

Et les constantes réelles peuvent prendre une des formes suivantes :

Constante_flottante



mantisse



-3.5e-6
.5
5.
3.e7

Chapitre 5 : Les opérateurs de base

Les opérateurs du C constituent un reflet fidèle des opérations machines effectuées lors des calculs de l'ordinateur. Ils fournissent au programmeur toutes les possibilités de l'assembleur sans pour autant se limiter à un matériel donné.
On peut classer les opérateurs en fonction de leur arité (unaires ou monadiques, binaires ou dyadiques, ternaires ou triadiques), c'est-à-dire en fonction du nombre d'opérandes qu'ils admettent. Mais on peut également les classer en fonction de leur utilisation comme on va le faire par la suite.

1. Les différentes classes d'opérateurs

1.1. Opérateurs arithmétiques

Les opérateurs sont ceux qui vont nous permettre d'effectuer des calculs mathématiques sur et entre les variables.
Parmi ces opérateurs, les plus communs sont l'addition (**+**), la soustraction (**-**), la multiplication (*****) et la division (**/**) entre deux expressions.
Chacun de ses opérateurs sont binaires et réalisent le calcul correspondant avec les valeurs des deux expressions se trouvant de part et d'autre de l'opérateur.

Exemple : a+b; 5-b; a/5; 3*4;

Il existe également l'opérateur modulo (%), qui calcule le reste de la division entière (ex : 10%3=1, 10%2=0).

L'opérateur unaire "moins" (-) donne la valeur négative d'un nombre (ex : si x vaut 3, -x vaut -3).

D'autres opérateurs permettent d'effectuer des opérations bits à bits sur les valeurs binaires des variables. Ainsi, on dispose des opérateurs suivants qui réalisent des opérations bit à bit :

- >> : décalage à droite
- << : décalage à gauche
- ~ : complément à 1

Le complément à 1 est un opérateur unaire. Les décalages sont des opérateurs dyadiques qui prennent en argument gauche un entier indiquant le nombre de décalage à effectuer.

1.2. Opérateurs logiques

On dispose également d'opérateurs logiques dont le résultat va être 0 si le test est considéré comme faux et toute autre valeur si il est vrai.
Ainsi, on peut tester l'égalité (**==**) ou la différence (**!=**) ; comparer deux valeurs (**<, >, <=, >=**).

Et combiner des tests par les opérateurs booléens ET (**&&**), OU (**| |**), OU EXCLUSIF (**^^**).
Ainsi qu'obtenir la valeur de vérité inverse par le NON (opérateur '**!**').

On dispose également d'opérateurs logiques bits à bits :

- & : et logique bit à bit
- | : ou logique bit à bit
- ^ : ou exclusif bit à bit

A l'exception de l'opérateur ' !' pour la négation qui est monadique, tous les autres opérateurs logiques sont dyadiques.

1.3. Opérateurs d'affectation

Ce n'est pas le tout de disposer d'opérateurs pour réaliser des tests ou des calculs, encore faut-il pouvoir les exploiter. Pour cela, il va falloir mémoriser le résultat de ces opérations dans des variables au moyen d'affectations.

L'opérateur élémentaire de l'affectation est l'opérateur dyadique '**=**'.
Il prend à gauche ce que l'on appelle une *lvalue*, c'est-à-dire une expression qui fasse référence à une mémorisation de la machine. Pour faire simple, une variable et donc surtout pas une constante, ce qui n'aurait de toute façon pas de sens.

Deux autres opérateurs, monadiques ceux-là, permettent une affectation simple, il s'agit de l'opérateur d'incrémentation (**++**) et de l'opérateur de décrémentation (**--**).
Ainsi, var++ revient à affecter var+1 à var ; soit var=var+1.
Ces deux opérateurs peuvent être utilisés en forme préfixée (var++) ou en forme post-fixée (++var).
Dans les deux cas, le résultat est identique mais nous verrons plus loin l'incidence que cela a par rapport à la précédence des opérateurs.

Le langage C dispose également d'opérateurs d'affectations combinés.
Des opérateurs d'affectation arithmétiques :

- a+=b équivaut à a=a+b
- a-=b équivaut à a=a-b
- a*=b équivaut à a=a*b
- a/=b équivaut à a=a/b
- a%=b équivaut à a=a%b

D'opérateurs de masquages :

- i&=8 équivaut à i=i&8 (i ET 8)
- i|=8 équivaut à i=i | 8 (i OU 8)
- i^=4 équivaut à i=i^4 (i OU exclusif 4)

Et d'affectations de décalages :

- i<<=4 équivaut à i=i<<4 (décale i à gauche de 4 positions)
- i>>=4 équivaut à i=i>>4 (décale i à droite de 4 positions)

2. Evaluation des opérateurs

2.1. Opérateurs et opérandes

Les opérandes sont les valeurs sur lesquelles on va appliquer les opérateurs.
Comme on l'a déjà signalé, pour les opérateurs d'affectation, on distingue les *lvalues* des *rvalues*, c'est-à-dire les valeurs qui doivent correspondre à une adresse mémoire (les *lvalues*, parties gauches des affectations) des valeurs qui peuvent indifféremment être des constantes ou des variables (*rvalues* ou parties droites des affections).

Tous les opérateurs arithmétiques et logiques vont pouvoir prendre comme opérandes indifféremment des *lvalues* ou des *rvalues* comme on le voit dans les exemples suivants.

Exemples :
circonference = 2 * pi * rayon
calcul = 3 + x++ * 5 >> 2

Le problème va alors être : comment évaluer une expression, quels opérateurs appliquer en premier, quel opérateur l'emporte sur tel autre ?
Pour cela, une priorité et un ordre d'évaluation on été associé à chaque opérateur du langage C.

2.2. Priorité et associativité des opérateurs

Pour évaluer une expression, on va donc se rapporter à la table suivante et considérer en premier les opérateurs les plus prioritaires. Dans cette table, les opérateurs les plus prioritaires se trouvent dans le haut du tableau, les moins prioritaires sont en bas.
Pour chaque opérateur, on va regarder son associativité, c'est-à-dire voir s'il faut d'abord évaluer la partie gauche de l'expression puis la droite, ou l'inverse.

opérateurs	Associativité	description
! ~ ++ -- - +	droite-gauche	unaires préfixés
* / %	gauche-droite	multiplicatifs
+ -	gauche-droite	Addition
>> <<	gauche-droite	décalages
< <= > >=	gauche-droite	relations d'ordre
= = !=	gauche-droite	égalité
& ^ |	gauche-droite	binaire
&& ||	gauche-droite	logique
? :	gauche-droite	conditionnel (ternaire)
= += -= *= etc.	droite-gauche	affectation

On va pouvoir dans tous les cas forcer une évaluation allant contre les priorités définies en utilisant le parenthésage des expressions à évaluer en premier.
Par exemple !a && b == c sera interprété comme (!a)&&(b==c).
On peut changer cette évaluation par exemple par !(a&&b)==c qui n'aura plus la même valeur.

Le cas des opérateurs d'incrémentation et de décrémentation est intéressant en ce qui concerne l'ordre d'évaluation. On a vu que ces opérateurs pouvaient être sous la forme préfixée (++a) ou post-fixée (a++) et que le résultat était le même dans les 2 cas, à savoir l'incrémentation de la variable.
Mais lorsque l'opérateur d'incrémentation ou de décrémentation est placé avant son opérande ce dernier est incrémenté avant d'être utilisé dans l'expression, et lorsqu'il est placé après son opérande ce dernier est incrémenté après son utilisation dans l'expression.
Ainsi, après l'instruction x=++y; y a été incrémenté de 1 et x a la même valeur.
Alors que après l'instruction x=y++; y a été incrémenté de 1 mais x et y ont une différence de 1.

2.3. Les conversions de type

L'opération de conversion d'un type en un autre s'appelle le ***cast***. Cette conversion explicite se fait en faisant précéder la variable à convertir du nom du type dans lequel on veut la convertir entre parenthèses :

(type) nomVar

Par exemple, on peut forcer une variable réelle (de type float) a être évaluée en tant qu'entier par l'instruction suivante : (int) varReelle

Mais il existe également en C des règles de conversions implicites qui s'appliquent lorsque des opérandes de types différents sont utilisés dans une même expression.

En ce qui concerne les affectations, la règle de conversion implicite est la suivante :

- Convertir les éléments de la partie droite d'une expression d'affectation dans le type de la variable ou de la constante le plus riche
- Faire les opérations de calcul dans ce type
- Puis convertir le résultat dans le type de la variable affectée (partie gauche de l'affectation)

Il faut donc définir la notion de richesse d'un type, la règle est la suivante :

1. si l'un des deux opérandes est du type **long double** alors le calcul doit être fait dans le type **long double**
2. sinon, si l'un des deux opérandes est du type **double** alors le calcul doit être fait dans le type **double**
3. sinon, si l'un des deux opérandes est du type **float** alors le calcul doit être fait dans le type **float**
4. sinon, appliquer la règle de promotion en entier, puis :
    1. si l'un des deux opérandes est du type **unsigned long int** alors le calcul doit être fait dans ce type
    2. si l'un des deux opérandes est du type **long int** alors le calcul doit être fait dans le type **long int**
    3. si l'un des deux opérandes est du type **unsigned int** alors le calcul doit être fait dans le type **unsigned int**
    4. si l'un des deux opérandes est du type **int** alors le calcul doit être fait dans le type **int**

Chapitre 6 : Expressions et instructions

1. Définitions

Une ***expression*** est une suite syntaxiquement correcte d'opérateurs et d'opérandes. Le nombre d'opérandes doit être cohérent avec le nombre et l'arité des opérateurs (unaires, binaires ou ternaires).
Une expression ramène toujours une valeur même si celle-ci n'est pas utilisée.

Une ***instruction*** est soit une instruction simple soit un bloc soit une instruction de contrôle.

Une ***instruction simple*** étant une expression suivie d'un « **;** » ou une instruction de contrôle. L'instruction vide est un simple « ; » seul.

Un ***bloc*** est composé d'une accolade ouvrante « { », suivi optionnellement d'une liste de déclaration de variables locales, suivi d'une suite d'instructions (simples, blocs ou de contrôle) et se termine par une accolade fermante « } ».

Exemple de bloc :

```
{ int x = 2;
        { float x = 1.5;
            /* x est un float */
        }
        /*x est un int */
        { char x = 'c';
            /* x est un char */
        }
        /* x est un int
}
```

Parmi les ***instructions de contrôle*** de contrôle on retrouve :

- Les instructions conditionnelles
- Les instructions répétitives
- Les ruptures de séquences (nous ne les aborderons ici que lorsqu'elles sont parties prenantes d'une instruction)

2. Les instructions de contrôle

2.1. Les instructions conditionnelles

2.1.1 Le test

Le test simple est l'instruction :

if (expression) instruction

L'*expression* testée n'est pas forcément un test au sens booléen du terme. En C il n'y a pas de type booléen. Cette expression peut donc être une affectation ou un calcul quelconque. La règle étant que si *expression* retourne la valeur 0, le résultat du test sera considéré comme faux ; il sera considéré comme vrai pour toute autre valeur de retour (différente de 0).

L'*instruction* peut-être une instruction simple ou un bloc.
En fait, la forme générale de l'expression conditionnelle est la suivante :

***if** (expression) instruction1*
***else** instruction2*

Mais la partie ***else** instruction2* est optionnelle.

Dans tous les cas, il est toujours possible d'imbriquer des conditionnelles, la règle étant que le ***else*** se rapporte au ***if*** le plus proche. Bien entendu, chacune des instructions peuvent elles-mêmes contenir des conditionnelles.

Exemple :

```
if ( i == 1 ) inst1
else      if ( i == 2 ) inst2
          else      if ( i == 3 ) inst3
```

Pour forcer un ordre d'imbrication des instructions conditionnelles, il suffit d'utiliser des blocs.

2.1.2 L'instruction de branchement (ou « selon »)

L'instruction de branchement (***switch*** en C) est une façon simplifiée d'écrire des ***if … else*** imbriqués en cascade.
Sauf que **le *switch* ne permet de tester que des expressions constantes**.

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

```
switch (expression) {
        case expression-constante1:  instruction1; break;
        case expression-constante2:  instruction2; break;
        …
        default: instructions
}
```

Avec *instruction1*, *instruction2*, *instructions* des instructions simples ou des blocs.
Le ***break*** est un « débranchement » qui permet de quitter le ***switch*** (il est facultatif).
La partie ***default*** est également facultative.

L'exécution du ***switch*** se passe comme suit :

- *expression* est évaluée comme une valeur entière ;
- les valeurs des ***case*** (*expression-constante*) sont évaluées comme des constantes entières ;
- l'exécution se fait à partir du ***case*** dont la valeur correspond à l'expression. Elle s'exécute en séquence jusqu'à la rencontre d'une instruction ***break*** ;
- les *instructions* qui suivent la condition ***default*** sont exécutées lorsque aucune constante des ***case*** n'est égale à la valeur retournée par *expression* ;
- l'ordre des ***case*** et du ***default*** n'est pas prédéfini par le langage mais par les besoins du programme ;
- l'exécution à partir d'un ***case*** continue sur les instructions des autres case tant qu'un ***break*** n'est pas rencontré ;
- plusieurs valeurs de ***case*** peuvent aboutir sur les mêmes instructions ;
- le dernier ***break*** est facultatif. Il vaut mieux le laisser pour la cohérence de l'écriture, et pour ne pas avoir de surprise lorsque l'on ajoute un ***case***.

2.2. Les instructions répétitives

2.2.1 La boucle *Tant que* (*while*)

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

while *(expression) instructions*

Où, comme dans le cas des conditionnelles, *expression* sera évaluée comme fausse si elle retourne 0 et comme vraie dans tous les autres cas.
Tant que *expression* sera vraie, on réitérera *instructions* (qui peut être une instruction simple ou un bloc).

Exemple :

```
printf("saisir un entier positif :") ;
scanf("%d",&nb);
while (nb<0) {
                printf("un entier positif !!!\n");
                scanf("%d",&nb);
                }
```

2.2.2 La boucle *Répéter…tant que* (*do…while*)

La syntaxe du ***do-while*** est :

do *instructions*
while *(expression)* ***;***

Les *instructions* sont effectuées tant que l'*expression* est vraie (non nulle).
Donc, contrairement au ***while***, les instructions sont effectuées au moins une fois.

Exemple :

```
do {    printf("saisir un entier positif :");
        scanf("%d",&nb);
  } while (nb>0);
```

2.2.3 La boucle *Pour* (*for*)

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

for *(expression1* ***;*** *expression2* ***;*** *expression3)* *instructions*

expression1 est la condition de départ (initialisations).
expression2 est la condition de fin.
expression3 est l' incrément de boucle.

Les 5 étapes du déroulement d'une boucle ***for*** sont :

1. *expression1* est évaluée avant d'entrer dans le ***for***
2. *expression2* est évaluée
3. si *expression2* est vrai, *instructions* est exécuté, sinon, on passe après la fin de la boucle et l'exécution de la boucle est finie
4. *expression3* est évaluée après l'exécution de *instructions*
5. on revient en 2

Exemples :

```
for (i=0 ; i<10 ; i++)
    printf("%d ",i);

for (i=0, j=0 ; i<N ; i++, j++)

f=1; s=0;
for (i=NB ; i>0 ; i--)
        {
                f*=i;
                s+=i;
        }
```

Remarque : il existe une équivalence entre ***while*** et ***for***

```
for (exp1; exp2; exp3) {
        instructions
}
```

est équivalent à

```
exp1
while (exp2) {
        instructions
        exp3
}
```

2.3. Rappels importants

Deux particularités du C sont sources de nombreuses erreurs de programmation :

- l'affectation est une expression
- il n'y a pas de type booléen

Il est en effet important de se souvenir que l'affectation est une **expression**, et non une **instruction**, et peut donc être utilisée à chaque fois que l'on peut mettre une expression.

On peut ainsi rencontrer, par exemple, dans les parties test des boucles ou des conditionnelles des instructions du type :

```
x=y=z=0
while ((y=f(x))!=0) ...
while ((c=getchar())!=EOF) ...
while (n-) ...
if (x-y) ...
```

Dans tous les cas, ces expressions seront évaluées et prendrons une valeur quelconque. Si cette valeur est 0, le résultat du test sera considéré comme faux, pour toute valeur différente de 0, il sera considéré comme vrai.

Bien faire également attention de distinguer l'affectation (=) du test d'égalité (==).
En effet, une erreur très classique est : ***if (x=0)***, toujours faux (valeur booléenne 0) et qui affecte la valeur 0 à *x*, au lieu de ***if (x==0)***.

Chapitre 7 : Les tableaux

1. Les tableaux à une dimension

1.1. Déclaration

La déclaration d'un tableau se fait au moyen de l'opérateur **[ ]**.
Cet opérateur se place à la suite de l'identificateur du tableau à déclarer.
Tous les éléments d'un tableau sont de même type, et ce type doit être précisé à la déclaration.
Le nombre d'éléments du tableau est fixé par la constante entière située entre les crochets (la **taille**).

Type nomTab[TAILLE]

Exemple :

```
int tab1[10];     /* déclaration d'un tableau de 10 entiers */
float tab2[5];    /* déclaration d'un tableau de 5 réels */
char tab3[50];    /* déclaration d'un tableau de 50 caractères */
```

La déclaration d'un tableau à une dimension réserve un espace de mémoire contiguë dans lequel les éléments du tableau peuvent être rangés. La taille réservée sera égale au nombre d'éléments du tableau multiplié par la place qu'occupe un élément du type du tableau. Par exemple, un tableau de 10 entiers occupera 10*4=40 octets, un tableau de 50 caractères, occupera 50*1=50 octets.
Le nom du tableau seul est une constante dont la valeur est l'adresse du début du tableau.

Chaque élément du tableau est identifié par son indice, c'est-à-dire sa position dans la suite des éléments. Le premier élément est l'élément d'indice 0, le dernier est l'élément d'indice **taille-1**.

1.2. Initialisation

On peut initialiser les tableaux à la déclaration

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

On peut n'initialiser que le début

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

Par contre, il faut bien faire attention aux tableaux partiellement remplis. En effet, si une case d'un tableau n'a pas été remplie explicitement avec une valeur, on ne sait pas ce qu'elle contiendra. On pourra tout à fait accéder normalement à cette case du tableau, mais la valeur stockée sera un entier aléatoire !
Dans le cas de variables locales, le tableau est initialisé avec des 0, mais n'étant pas le cas général, mieux vaut se rapporter à la règle précédente et s'habituer à initialiser complètement un tableau avant de l'utiliser.

Autre possibilité : fixer une taille par défaut

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

Avec une telle déclaration, la taille du tableau **tab** sera arbitrairement fixée au nombre d'éléments donné lors de l'initialisation. Dans l'exemple, un tableau de 3 entiers.

Exemple :
int arr[10] = { 9,8,7,6,5,4,3,2,1,0};

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]
9	8	7	6	5	4	3	2	1	0

1.3. Accès aux éléments d'un tableau

Les éléments sont accessibles par : le nom du tableau, un crochet ouvrant, l'indice de l'élément et un crochet fermant : *nomTab[indice]*

On se retrouve alors exactement dans le cas d'une variable de type simple que l'on peut utiliser en tant qu'opérande de tous les opérateurs définis en C.

Attention : ne pas oublier que les indices vont de 0 à taille-1 !!!

Exemple :

```
char tab[5];      /* déclaration d'un tableau de 5 caractères */

tab[0] = 'a';             /* affectation du caractère 'a' dans la case d'indice 0 du tableau tab */
tab[1] = 'b';             /* affectation du caractère 'b' dans la case d'indice 1 du tableau tab */
tab[2] = tab[0];          /* affectation dans la case d'indice 2 du contenu de la case d'indice 0 */
if (tab[0] == tab[2]) printf("Mêmes éléments\n"); /* comparaison des cases d'indices 0 et 2 */
```

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]
'a'	'b'	'a'		

```
tab[2]++ ;                /* incrémentation du contenu de la case d'indice 2 */
```

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]
'a'	'b'	'b'		

Exemple d'utilisation : lecture et affichage des valeurs d'un tableau

```
#include <stdio.h>
#define NLMAX 5
void main ()
{
        int i, tab[NLMAX];
        for (i=0; i< NLMAX; i++) {
                printf ("entrer tab[%2d] : ", i);
                scanf ("%d", &tab[i]);
        }

        for (i=0; i< NLMAX; i++) {
                printf ("tab[%2d] = %d \n", i, tab[i]);
        }
}
```

2. Les tableaux à plusieurs dimensions

2.1. Déclaration

On peut appliquer l'opérateur **[ ]** plusieurs fois et ainsi créer des tableaux à 2 dimensions ou plus.

Les tableaux à deux dimensions sont en fait des tableaux de tableaux.
Les tableaux à n dimensions sont des tableaux de tableaux à n-1 dimensions.
Les indices de droite correspondent aux indices des tableaux les plus internes.

Type nomTab[TAILLE1] [TAILLE2]… [TAILLEn]

Exemple :

```
int matrice[4][3] ;        /* déclaration d'une matrice de 4 lignes par 3 colonnes */
char espace[3][5][2];      /* déclaration d'un tableau à 3 dimensions de caractères */
float tropDeDimensions[2][3][4][5][6] ; /* tableau à 5 dimensions ! */
```

Graphiquement on représente les matrices (2 dimensions) sous la forme d'un tableau lignes, colonnes. En réalité, pour le langage C, toutes les données sont à la suite les unes des autres.
D'où l'importance du nombre de colonnes déclarées, car c'est lui qui permet d'accéder aux éléments du tableau.
Et c'est de même pour des tableaux à plusieurs dimensions, il faut connaître le nombre d'éléments à sauter pour arriver dans la dimension suivante.

2.2. Initialisation

On va pouvoir initialiser les tableaux à plusieurs dimensions soit en marquant les différentes dimensions en ajoutant un niveau d'accolades ({}), soit en donnant directement les éléments sous forme d'une liste (comme ils seront stocké en mémoire). Dans tous les cas, il faut bien fixer toutes les dimensions « intérieures ».

Soit une matrice de deux lignes et 3 colonnes, on peut l'initialiser de 2 manières différentes

```
int mat [2][3] = {{1, 2, 3}{4, 5, 6}};/* initialisation d'une matrice 2 lignes 3 colonnes */
ou   int  mat[2][3] = { 1, 2, 3, 4, 5, 6 };    /* idem */
```

Les possibilités de déclaration vues pour les tableaux à une dimension s'appliquent aussi, mais, il faut au moins préciser la taille d'une ligne (le nombre de colonnes)

```
int mat [][3] = { 1, 2, 3, 4, 5 }; /* initialisation d'une matrice de 3 colonnes, le nombre de
                                      lignes sera fixé à 2 à cause du nombre d'éléments
                                      le dernier élément ne sera donc pas initialisé */
```

2.3. Accès aux éléments

Les éléments sont accessibles par : le nom du tableau, un crochet ouvrant, l'indice de l'élément dans la 1ère dimension, un crochet fermant, un crochet ouvrant, l'indice dans la 2nde dimension, un crochet fermant, … :

nomTab[indice1][indice2]	pour une matrice
nomTab[indice1][indice2][indice3]	pour un tableau à 3 dimensions
…	

Comme pour les tableaux à une seule dimension, les indices de chaque dimension varient de 0 à taille-1.

Exemples :
int mat[4][3];
mat est un tableau de 4 * 3 = 12 éléments (par convention on dit 4 lignes et 3 colonnes)

mat[3][2]=5; on met 5 dans la 3 * 3 + 2 = 11ème case du tableau,
en partant de 0

char espace[3][5][2];
espace est un tableau de 3 * 5 * 2 = 30 éléments

espace[2][3][1]='a' on met 'a' dans la (2*5+3)*2+1=27ème case
en partant de 0

Exemple d'utilisation : Affichage d'une matrice NxM

```
for (i=0;i<N;i++)
        for (j=0;j<M;j++)
                printf("%5d\n",mat[i][j]);
```

3. La redéfinition d'un type tableau

3.1. Tableaux à une dimension

typedef int typTab[3];

typTab est le type tableau de 3 entiers

typTab tab;

Cette déclaration est équivalente à int tab[3];

3.2. Tableaux à deux dimensions

typedef char typMat [10][5];

typMat est le type d'une matrice de 10 lignes et 5 colonnes de caractères

typMat mat;

Cette déclaration est équivalente à
char mat[10][5];

Bien entendu, on peut redéfinir des types tableaux à n dimensions. Le principe reste exactement le même.

Chapitre 8 : Les Structures

Une structure est une réunion d'objets de types différents.
Les structures s'appellent aussi des enregistrements.
Les éléments d'une structure sont des champs (ou membres).

struct *nomStruct* **{**
liste de déclaration de champs
};

1. Déclaration et initialisation

La déclaration d'une variable de type structuré se fait comme pour toute autre variable, le type étant ***struct nomStruct*** (attention de ne pas oublier le mot-clé **struct** !).

L'initialisation d'une structure se fait soit champs par champs, soit lors de la déclaration.
L'initialisation à la déclaration se fait en précisant les valeurs de chacun des champs entre guillemets et séparés par des virugles. Les valeurs doivent être données dans l'ordre de la déclaration des champs et respecter leurs types.

Exemples :

```
struct nbcomplexe
{
   double pr;
   double pi;
};
struct point
{
   double abs,ord;
   int code_couleur;
};
struct eleve
{
   char nom[20];
   int age;
   float note;
};

struct eleve e1, e2, mec = {"Dupont", 20, 10.5};
struct point p1, pt;
struct nbcomplexe z1,z2,z3;
```

2. Accès aux champs

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se fait en faisant suivre le nom de la variable structurée par l'opérateur "**.**" suivi du nom du champ.

Exemples :

```
scanf ("%s", &e1.nom)
scanf ("%d", &el.age)
e1.note = 10;
e1.nom[3]='a';
```

L'opérateur "**.**" a une priorité très élevée, supérieure à celle des autres opérateurs

On peut affecter de manière globale une structure dans une autre. Ainsi, on peut écrire :

```
e2 = e1;
```

Cela équivaut à affecter chacun des champs e1 dans ceux de e2. Bien sur cela n'est possible que si les structures ont le même type.

Remarque : on ne peut pas faire le même type d'affectation avec un tableau.

3. La redéfinition d'un type structuré

```
typedef struct eleve {
       char  nom[20] ;
       int   age;
       float note;
 } typEleve;
```

Cette écriture renomme le type *struct eleve* en *typEleve*
Ainsi on pourra déclarer les variables :

```
typEleve el1, el2;
```

4. Imbrication de structures

Toute structure peut avoir comme champs une autre structure, voire une structure identique à elle-même.

```
typedef struct personne {
      char  nom[20] ;
      char  prenom[30];
      int   age;
} typePersonne;

typedef struct infoNote {
      float note;
      matiere char[20];
      char date[9];
} typeInfoNote;

typedef struct eleve {
      typePersonne etat_civil;
      typeInfoNote notes[5];
} typeEleve;

typeEleve e1;

printf("quelle matière ? ");
scanf("%s",&e1.notes[2].matiere);
e1.etat_civil.age=20;
e1.notes[0].date[8]=3;
```

Chapitre 9 : Les types énumérés et les unions

1. Les énumérations

Les énumérations servent à offrir des possibilités de gestion de constantes énumérées dans le langage C Ces énumérations sont un apport de la norme ANSI et permettent d'exprimer des valeurs constantes de type entier en associant ces valeurs à des noms.
Les énumérations offrent une alternative à l'utilisation du pré-processeur.
On définit une énumération par :

```
enum  nom_de_énumération {
        énumérateur1,
        énumérateur2,
        etc.
        énumérateurN
      };
```

1.1. Déclaration et initialisation

Les valeurs associées aux différentes constantes symboliques sont, par défaut, définies de la manière suivante : la première constante est associée à la valeur 0 ; les constantes suivantes suivent une progression de 1.
Dans l'énumération suivante, blanc correspond à la constante 0, rouge à 1, vert à 2 et bleu à 3 :

```
enum couleurs {blanc, rouge, vert, bleu};
```

Il est possible de fixer une valeur à chaque énumérateur en faisant suivre l'énumérateur du signe égal et de la valeur entière exprimée par une constante ; la progression pour l'énumérateur suivant reprend à 1. Par exemple, violet sera associé à 4, orange à 5 et jaune à 8 :

```
enum autrescouleurs {violet=4, orange, jaune=8};
```

1.2. Utilisation

Les énumérateurs peuvent être utilisés dans des expressions du langage à la même place que des constantes du type entier
Ils peuvent se situer dans des calculs, pour affecter des variables et pour réaliser des tests :

```
enum couleurs couleur1 = vert;
enum autrecouleurs couleur2;
couleur1++;
if (couleur1==bleu) printf("couleur bleue\n");
couleur2=bleu+rouge;
if (couleur2==violet) printf("couleur violette\n");
```

1.3. Redéfinition d'un type énuméré

```
typedef enum j_semaine {lundi, mardi, mercredi, jeudi,
      vendredi, samedi, dimanche} jours_semaine;
```

Cette écriture renomme le type *enum j_semaine* en *jours_semaine*
Ainsi on pourra déclarer les variables énumérées suivantes :

```
jours_semaine jour1, jour2;
```

2. Les unions

Les unions permettent à des variables de types différents de partager un même emplacement mémoire.
La syntaxe de l'union est proche de celle des structures :

union *nom_de_union* **{**
type1 nom_champ1;
type2 nom_champ2;
etc.
typeN nom_champ_N;
};

2.1. Déclaration

```
union zone {
      int entier;
      long entlong;
      float flottant;
      double flotlong;
};

zone z1, z2;
```

Le compilateur réserve l'espace mémoire nécessaire pour stocker le plus grand des champs appartenant à l'union. Dans l'exemple précédent, la place pour un double sera réservée.

2.2. Accès aux champs

La syntaxe d'accès aux champs d'une union est identique à celle pour accéder aux champs d'une structure (opérateur '**.**').
Une union ne contient qu'une donnée à la fois.
L'accès à un champ de l'union pour obtenir une valeur, doit être fait dans le même type qui a été utilisé pour stocker la valeur.
Les différents champs d'une union sont à la même adresse physique.

2.3. Redéfinition d'un type union

```
typedef union test {
            long entier;
            double reel;
            char ch[10];
      } test;
```

Cette écriture renomme le type *union test* en *test*.
Ainsi on pourra déclarer les variables :

```
test t1, t2;
```

Chacune de ces variable sera une union comme définit précédemment.

Remarque : utilisation déconseillée !

Chapitre 10 : Les fonctions

Les fonctions sont des parties de code source qui permettent de réaliser le même traitement plusieurs fois ou sur des variables différentes.
En C, on ne distingue pas vraiment procédure et fonction : une procédure est une fonction qui ne retourne rien (void).
Toutes les fonctions sont indépendantes les unes des autres et sont compilées séparément.

1. Le prototype d'une fonction

1.1. Déclaration

Toutes les fonctions doivent être déclarées avant leur définition au moyen d'un prototype.
Le nom de la fonction est précédé d'un type indiquant le type de la valeur qu'elle retourne et est suivi de la liste de ses arguments ou paramètres formels.

Type nom_de_fonction(TypeArg1, TypeArg2, …, TypeArgn);

Les types qui peuvent être retournée par une fonction sont :

- void (c'est-à-dire rien)
- int, float, double, char
- une structure
- un pointeur

Une fonction peut ne pas avoir de paramètre

Type nom_de_fonction();

1.2. Définition

Type nom_de_fonction(TypeArg1 Arg1, TypeArg2 Arg2, …, TypeArgn Argn)
{
instructions;
}

La définition (ou implémentation) d'une fonction reprend le prototype de celle-ci.
Les paramètres doivent être nommés (en plus d'être typées).
Un bloc constitue le corps de la fonction.

2. Le corps de la fonction

Type nom_de_fonction(TypeArg1 Arg1, TypeArg2 Arg2, …, TypeArgn Argn)
{ *déclaration de variables locales à la fonction*
instructions;
return *expression;* }

Le corps est un bloc.
Il peut commencer par une liste de variables locales et comporte les instructions de la fonction.

Le retour d'une valeur se fait par l'instruction ***return*** suivie d'une expression du type de la fonction
Une fonction de type ***void*** n'a pas besoin de l'instruction ***return***.

3. Les paramètres de la fonction

Les arguments sont passés soit par valeur soit par adresse.

- Si les arguments sont transmis par valeur :
  - ils peuvent être modifiés par la fonction sans être modifiés dans la fonction appelante
  - les variables de la fonction sont alors locales et un nouvel espace mémoire est alloué à chaque appel de la fonction
  - elles ne gardent donc pas leur valeur d'un appel à l'autre
- Si les paramètres doivent être modifiés par la fonction, ceux-ci devront être transmis par adresse

3.1. Passage par valeur

La fonction appelante fait une copie de la valeur passée en paramètre et passe cette copie à la fonction appelée.
Le paramètre est alors géré comme une variable locale de la fonction.
Cette variable est accessible de manière interne par la fonction à partir de l'argument formel correspondant.
En sortie de fonction, les paramètres reprennent leur valeur initiale (avant appel de la fonction).

Exemple :

```
int plus(int a, int b)
{     a=a+b;
      return a; }

void main() {
      int a; b=4;
      int c=5;
      a=plus(1,2);      /* résultat 3 */
      b=plus(a, b);     /* résultat 7, après l'appel a=3, b=4 */
      plus(c, b);       /* résultat 12, après l'appel b=7, c=5 */
}
```

3.2. Les tableaux en paramètre

On peut passer un tableau en paramètre d'une fonction :

Type nom_de_fonction(int tab[TAILLE]) **{ … }**

Dans ce cas, il n'y a pas de recopie du tableau, la fonction va travailler directement sur le tableau initial.
Résultat : en sortie, les valeurs du tableau auront changées.

En fait, un tableau est l'adresse de la première case du tableau, il s'agit donc en réalité d'un passage par adresse.

Exemple :

```
int som(int t[TAILLE], int nbelem)
{
      int i,som;
      for(i=0,som=0; i<nbelem; i++)
            { som+=t[i];
               t[i]++; }
      return som;
}
```

```
void main()
{
      int tab[4]={1,5,8,10};
      int som;
      som=som(tab, 4);
}
```

Résultat :

som vaut 24
tab contient {2, 6, 9, 11}

3.3. Passage par adresse

Pour qu'une fonction puisse modifier la valeur d'un de ses paramètres, il faut donc passer en argument l'adresse de la variable (et non sa valeur).

Pour cela, on a besoin d'utiliser des pointeurs…