Cours de C - ANSI
1ère partie (4 cours de 2 heures)
Bertrand LIAUDET

Qu'est-ce que la science ? La science c'est ce que le père enseigne à son fils. Qu'est-ce que la technologie? C'est ce que le fils enseigne à son père.

Michel Serres

SOMMAIRE DE LA PREMIERE PARTIE

1ère édition : octobre 2002 – mise à jour juillet 2007 – mise à jour septembre 2011

CHAPITRE 0 : PRESENTATION GENERALE

0.1 Organisation générale

1 Programmation structurée en langage C

L'objectif de ce cours est de vous apprendre un langage de programmation et, à travers cet apprentissage, de vous apprendre les principes de la programmation structurée.

C'est un cours de programmation en langage C-ANSI. Ce n'est pas un cours de C++ !!! Chaque chose en son temps. Avant d'apprendre la programmation objet, il faut d'abord connaître les bases de la programmation fonctionnelle (ou procédurale).

Ces bases permettent ensuite l'auto-apprentissage de tout autre langage procédural classique, et vous arment pour le passage au monde des objets.

Principaux éléments abordés :

- Présentation des éléments du langage C
- Les pointeurs
- Le mode de passage des paramètres dans les fonctions
- La programmation structurée

On part du niveau 0 et on monte vite !

2 Préceptes

Ce cours est parsemé de preceptes. Qu'est-ce qu'un pécepte : c'est une formule qui exprime un enseignement, une règle, une recette. En plus du vocabulaire, de la syntaxe et de la sémantique, ce cours donne les règles de l'art de la programmation.

Précepte n° 1

Relisez le manuel ! ! !

En effet, la routine nous entraîne à nous débrouiller avec toujours les mêmes outils alors qu'il en existe souvent d'autres qui sont très puissants.

Mieux vaut profiter du cours et des TD pour rafraîchir ses connaissances et lire de bons livres.

0.2 Bref historique de la programmation structurée et du langage C

•> 1960, ALGOL 60

ALGOL pour ALGOrithmic Language. L'un des tous premiers langages informatiques, comprenant des blocs, des procédures et des structures de contrôle. Ancêtre du Pascal, créé en 1960 par des chercheurs européens.

•> 1968, E.W. Dijkstra

Publication de l'article "GO TO Statements Considered Harmful" qui pose les bases de la programmation structurée en rejetant le "go to".

•> 1968, Niklaus Wirth

Création du langage Pascal.

•> 1972, Dennis Ritchie

Le langage C a été inventé en 1972 par Dennis Ritchie pour le système UNIX.

Il fait suite au langage B qui date de 1970. Le B n'était pas typé. Le C est typé (entiers, réels, caractères, enregistrements, etc.) et structuré (tests, boucles, fonctions).

Le langage C est donc un langage de troisième génération (type Pascal ou ADA).

Rappelons les différents types de langage

- **première génération** : le binaire. Le processeur est piloté directement par un langage machine binaire ayant un jeu d'instructions minimal (addition, comparaison, déplacement).
- **deuxième génération** : les assembleurs. Ces langages mettent en œuvre des instructions avec mnémoniques (MOV, ADD, etc.) qui permettent la manipulation d'informations plus complexes.
- **troisième génération** : les langages procéduraux. Ils proposent un jeu d'instructions plus riche et des données typées ce qui rapproche la programmation du langage naturel. De plus ils offrent la possibilité d'écrire des sous-programmes.

  Les langages de $1^{ère}$, $2^{ème}$ et $3^{ème}$ génération sont des langages impératifs. On donne des ordres : fait ceci, fait cela !

  On y range : le Fortran, le Cobol, l'Algol, Le Simula, le Pascal, le C, l'ADA, le PHP, etc.
- Les langages de **quatrième génération** sont les langages de base de données (le SQL particulièrement).
- Les langages de **cinquième génération** sont les langages adaptés à l'intelligence artificielle, aux systèmes experts et autres bases de connaissances. On peut distinguer deux grands types de langage adaptés à l'intelligence artificielle : les moteurs d'inférences classiques (le Prolog) et les systèmes à apprentissage (réseaux de neurones).

  Les langages de $4^{ème}$ et $5^{ème}$ génération ne sont plus impératifs : on ne donne plus uniquement des ordres. On va pouvoir poser des questions.

- L'**orientation objet** de la programmation concerne d'abord et essentiellement les langages de troisième génération : C objet, Pascal objet, etc. La représentation des données propre à l'objet concerne aussi les langages de quatrième génération (SQL objet) et de cinquième génération.

Le langage C, comme ses cousins Pascal et ADA, hérite de l'ALGOL 60.

Le langage C est un langage à la fois évolué et proche de la machine, qui allie à la fois haut niveau et bas niveau.

•> 1978, Kernighan et Ritchie

En 1978 est publié le premier manuel de référence du langage C.

•> 1988, C-ANSI

Fin 1988, après 5 années de travail, l'American National Standards Institute produit la norme ANSI / X3.159-1989.

L'événement s'accompagne de la publication de la deuxième édition du Kernighan et Ritchie.

Viendront ensuite les normes ISO (International Organisation for Standardisation) et AFNOR : le langage C s'appelle désormais **le langage C standard**. Il est adulte.

•> Les clés du succès

- La simplicité : c'est un langage de petite taille assez facile à assimiler.
- La puissance : c'est un langage universel à la fois de haut niveau et de bas niveau.
- La portabilité : les programmes (sources) C sont assez facilement portables sur des machines différentes et sur des systèmes d'exploitation différents.
- Ses relations avec le reste du monde informatique :

  UNIX, Internet,

  Objet (C++),

  Base de données (C et SQL)..

Le C s'est imposé dans tous les domaines de la programmation. Les applications systèmes (SE, bibliothèques, langages, etc.), les applications industrielles et aussi les applications de gestion.

0.3 Bibliographie de référence pour le langage C

K&R91 **Kernighan B.W. et Ritchie D.M., Le Langage C - ANSI, Masson - Prentice Hall, 1991. 280 p.**

C'est le manuel de référence (Ritchie est le créateur du langage C). Malheureusement, les premiers exemples du livre très orientés UNIX (ils concernant les getchar) sont particulièrement obscurs pour des programmeurs PC.

T&D90 **Clovis L. Tondo et Scott E. Gimpel, Exercices corrigés sur le langage C – Solutions des exercices du Kernighan et Ritchie, Dunod – Prentice Hall, 1990, 157 p.**

Pour s'entraîner et aller jusqu'au bout du K. & R.

Bra98 **Braquelaire Jean-Pierre, Méthodologie de la programmation en C, Masson, 1990, 1998. 556 p.**

C'est aussi un manuel de référence. Très orienté UNIX, il est plus difficile que le K&R.

0.4 Bibliographie pour le langage C

Les livres d'algorithmique en C donnent aussi de bons exemple de code C. Par exemple :

http://www.amazon.fr/Algorithmique-en-C-Jean-Michel-L%C3%A9ry/dp/2744074799/ref=sr_1_1?s=books&ie=UTF8&qid=1325518633&sr=1-1

Del97 **Delannoy Claude, Programmer en Langage C, Eyrolles, 1992, 1996, 1997. 282 p.**

Ce livre a l'avantage d'être assez facile à lire.

On peut aussi citer : les livres de poche en tout genre.

0.5 Sites web pour le langage C

Le site de référence de la bibliothèque standard

http://www.cplusplus.com/reference/clibrary/

Le site wiki

http://fr.wikiversity.org/wiki/Langage_C

Le site du zéro et le livre associé :

http://www.siteduzero.com/tutoriel-3-14189-apprenez-a-programmer-en-c.html

Un site avec de multiples références :

http://www.bien-programmer.fr/index.php

Un site de cours en forme de « faq »

http://www.levenez.com/lang/c/faq/

Evidemment, il y a de nombreuses autres sources sur internet.

0.6 Bibliographie pour la programmation structurée

Dij65 E.W. Dijkstra, "Programming Considered as a Human Activity", In Proc. IFIP Congress, pages 213-217, 1965.

Dij68 E.W. Dijkstra : "GO TO Statements Considered Harmful", Communications of the ACM, 11(3):147-148, Mars 1968.

http://www.acm.org/classics/oct95/

Par72 D. Parnas : "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, 14(1): 221-227, 1972.

http://www.acm.org/classics/may96/

Tre85 Jean-Paul Tremblay, Richard B. Bunt et Paul G. Sorenson, "L'art de programmer", tiré de "Logique de programmation. Initiation à l'approche algorithmique", Montréal, McGraw-Hill, pp. 297-336.

http://www.teluq.uquebec.ca/expl_inf1200/module2_3/mod2_3_act1.htm

Wir71 N. Wirth : "Program Development by Stepwise Refinement", Communications of the ACM, v.14, n.4, Apr. 1971, pp.221-227

http://www.acm.org/classics/dec95/#3

Wir74 N. Wirth: " On the composition of well-structured programs", ACM Computing Surveys, 6(4):247-259, December 1974.

Autres références Internet :

http://www.acm.org/classics

http://www.adelphi.edu/sbloch/class/adages

CHAPITRE 1 : PREMIERS PROGRAMMES ET PREMIERES NOTIONS DE PROGRAMMATION ET DE C

Ce chapitre présente les premières notions du langage C à partir de quelques petits programmes simples.

1.1 Affichage

Un programme c'est quoi : ce sont des ordres qu'on donne à une machine. Pour qu'on puisse se rendre compte de quelque chose, il faut que la machine « montre » les résultats : c'est ce que permet l'affichage.

1 Sujet

Ecrire un programme qui affiche Bonjour à l'écran.

2 Analyse

Il n'y a pas de calcul, mais seulement une chose à faire : afficher bonjour.

3 Solution

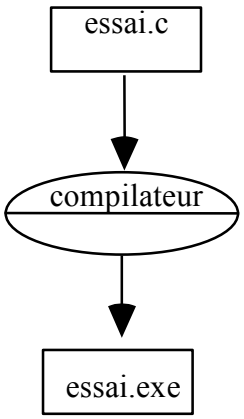
```
# include <stdio.h>
void main(void)
{
    printf ("Bonjour \n");
}
```

essai.c

4 Compilation et exécution

Le fichier "essai.c" est un programme C.

Avec le compilateur, on va créer un fichier exécutable appelé "essai.exe" (sous DOS) à partir du **fichier source** "essai.c". Le fichier "essai.exe" est un programme, un logiciel, un exécutable (d'où l'extension ".exe") contenant des instructions compréhensibles par la machine.



Le compilateur c'est un traducteur et vérificateur de syntaxe. C'est comme si on passait du français à l'anglais sauf que là on passe du C au langage machine.

On s'intéressera au compilateur en T.P. (sous DOS ou sous UNIX). Il produit un **fichier exécutable**.

Exemple d'exécution du programme (ici sous UNIX)

```
$ essai ↵
bonjour
$
```

5 Explications

Exp. 1 : printf

La troisième ligne, "printf…", est la seule **instruction** du programme, c'est-à-dire le seul moment où est dit explicitement ce qui est à faire : ici afficher « Bonjour ». Tout le reste n'est que le « cadre » du programme.

printf est une fonction qui permet d'afficher du texte et le contenu de variables à l'écran.

printf permet d'afficher ce qui ce trouve entre guillemets.

Le " \n " est un caractère qui signifie qu'on passe à la ligne après l'affichage.

On aurait pu écrire :

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

Avec cet exemple on voit qu'il n'y a pas qu'une seule solution possible.

Exp. 2 : accolades et notion de bloc

Les instructions du programme principal sont comprises entre " { " et " } ". Cet ensemble d'instructions forme un bloc d'instructions. L'accolade ouvrante est un marqueur de début de bloc, l'accolade fermante un marqueur de fin de bloc.

Exp. 3 : void main (void)

La deuxième ligne "void *main* (void)" marque le début du programme principal. Le *main* est une fonction, comme printf : on peut lui passer des paramètres et elle peut renvoyer une valeur. Mais le printf est une fonction ici utilisée, tandis que le *main* est une fonction qu'on est en train de définir.

Toute définition de fonction est constituée d'une en-tête (void *main* (void) et d'un corps (le bloc d'instructions).

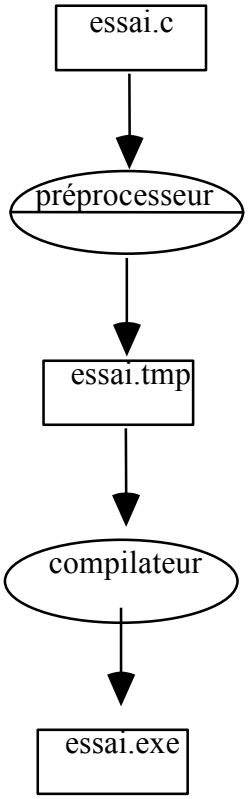
"void" signifie explicitement que le *main* ne renvoie pas de valeur et qu'il n'a pas de paramètres. Ceci peut s'appliquer à toutes les fonctions. On peut toujours ne pas écrire "void". écrire : *main*() tout court, mais il est préférable de préciser "void" plutôt que de ne rien mettre. On reverra tout ça dans le chapitre sur les fonctions.

Exp. 4 : include

La première ligne, # include <stdio.h> , est une instruction particulière : c'est ce qu'on appelle une **directive** qui sera prise en compte par le **préprocesseur** avant la traduction (compilation) du programme.

Cette directive demande d'inclure le fichier stdio.h dans le programme. C'est dans ce fichier qu'est déclarée la fonction "printf". Ainsi, elle sera reconnue par le compilateur. (Attention : « déclarer » une fonction ce n'est pas la même chose que la « définir », on y reviendra).

Le préprocesseur ajoute des éléments dans le fichier essai.c de départ. On appelle ces éléments des fichiers inclus : dans notre exemple, il s'agit du fichier stdio.h. Le prépocesseur produit un nouveau fichier (temporaire) qui est ensuite compilé pour former l'exécutable.



printf est une fonction prédéfinie dans la **bibliothèque standard** du langage C.

Les fichier ".h" sont appelés des **fichiers d'en-tête** (h est l'initial de "header" qui signifie en-tête).

Exp. 5 : majuscules et minuscules

Le langage C distingue les majuscules et les minuscules. Il ne faut pas écrire Printf, ni PRINTF. Par contre le texte entre guillemets du printf est affiché tel quel.

Exp. 6 : point virgule

Toute instruction se termine par un point virgule. Ici il n'y a qu'une instruction : le printf.

Exp. 7 : forme générale d'un programme

Dans un premier temps, disons et retenons que tout programme C a généralement cette forme :

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

1.2 Variables et affectation

1 Sujet

Ecrire un programme qui transforme une valeur donnée en degrés Celcius (°C) en degrés Fahrenheit (°F).

La formule c'est : °F = 9 / 5 * °C + 32

2 Analyse

Le problème est un peu plus compliqué : il ne faut pas seulement afficher, il faut d'abord faire un calcul à partir d'une valeur récupérée au clavier. Il y a donc 3 étapes :

1) récupérer la valeur en °C
2) transformer cette valeur en °F
3) afficher le résultat.

3 Solution

```
# include <stdio.h>
void main(void)
{
    int fahr, celcius ;
    printf("entrez une valeur en degré celcius : ") ;
    scanf("%d", &celcius) ;
    fahr = celcius*9/5 + 32 ;
    printf ("%d F = %d C \n", fahr, celcius);
}
```

essai.c

4 Exécution

```
$ degre ↵
entrez une valeur en degré celcius : 30 ↵
86 F = 30 C
$
```

5 Explications

Exp. 8 : déroulement du programme

Les instructions sont exécutées les unes après les autres, en partant de l'accolade ouvrante après le *main* jusqu'à l'accolade fermante du *main*.

Exp. 9 : forme générale

On retrouve la forme générale présentée précédemment : include, *main* { instructions }

Exp. 10 : variables

fahr et celcius sont des variables

La notion de variable est la première notion fondamentale de l'informatique et de la programmation.

Une variable informatique ce n'est pas la même chose qu'une variable mathématique.

La définition d'une variable est une chose difficile qui relève de l'épistémologie et pas des mathématiques. C'est pourquoi on trouve plusieurs définitions.

En mathématique, la variable est un symbole, généralement une lettre, défini de telle sorte que ce symbole peut être remplacé par **0, 1 ou plusieurs valeurs**. Les valeurs possibles d'une variable peuvent être **déterminées ou pas**. Les exercices d'algèbres consistent souvent à déterminer les valeurs de variables.

En informatique, une variable a **une valeur** et une seule, valeur toujours **déterminée**. Cette valeur peut être modifiée au fur et à mesure du déroulement du programme.

Une variable c'est un contenant (une boite, une ardoise, etc.) qui contient une valeur (une information).

Une variable est caractérisée par :

- un nom : celcius : c'est l'identificateur de la variable. Le nom peut faire référence à la valeur ou au contenant selon son usage.
- une valeur (ou contenu): 30 : c'est la traduction en fonction du type de l'état physique réel de la variable.
- un contenant : c'est le bout de mémoire dans lequel la valeur est stockée. Je le symbolise par un carré.
- une adresse : ad_celsius, c'est la localisation du contenant.
- un type : Il permet de distinguer différents ensembles de valeurs possibles pour la variable (entier, réel, lettre, etc.).
- une signification (ou sens, attention à la polysémie) : par exemple, celcius c'est la température en °C. La signification est conventionnelle (c'est le programmeur qui la choisit). Mais elle est très importante. C'est l'occasion d'un nouveau précepte de programmation:

Précepte n° 2

Nommez significativement les variables !

En effet, dans le programme précédent, on pourrait appeler fahr et celcius x et y, ou même celcius et fahr ! Comme ça, on aurait toutes les chances de ne plus rien y comprendre ! Pensez à la relecture!

Schématiquement, on représentera les variables ainsi :

RETENIR : tout!

Il faut se représenter clairement ce qu'est une variable (un tiroir, une page blanche, une ardoise). C'est quelque chose où je peux mettre une info. Cette chose est fixe. Elle a une position géographique : c'est son adresse. Elle a un nom qui me facilite la vie (c'est pour cela qu'il vaut mieux qu'il soit significatif, sinon on pourrait la nommer par son adresse!). Elle a toujours un contenu. Le nom c'est "n". Le nom fait référence à (on peut dire c'est) soit la valeur, soit le contenant.

Exp. 11 : déclaration des variables et types

L'instruction "int fahr, celcius;" est une déclaration de variables.

Quand on écrit un programme, on commence par dire quelles variables on va utiliser, autrement dit sur quoi on va travailler.

Ici on dit que fahr et celcius sont deux variables de type int, c'est-à-dire entier.

Quand on déclare une variable, il faut préciser son type.

En C, il y a **4 types de base : int, float, double et char** qui correspondent aux entiers, aux réels et aux caractères. Il y a deux types réels : le double est plus précis que le float (il y a plus de chiffres après la virgule).

Déclarer une variable c'est créer une variable dont on pourra se servir ensuite. Elle a un nom, un type et une adresse.

Quand on déclare une variable, sa valeur c'est : n'importe quoi.

Exp. 12 : affectation

L'instruction « fahr = 9 / 5 celcius + 32 ; » est une affectation.

La notion d'affectation est la deuxième notion fondamentale de la programmation.

L'affectation est ce qui permet de donner une valeur à une variable.

Ici, l'affectation permet de donner à fahr le résultat de l'évaluation de l'expression : 9 / 5 celcius + 32. Au départ, fahr avait n'importe quelle valeur. Ensuite il a la valeur calculée.

Si j'écrivais ensuite fahr = 5 ; alors la valeur de fahr deviendrait 5.

le double usage du nom d'une variable

Le **nom** "celcius" est utilisée dans une expression à évaluer. "celcius" c'est la **valeur** de "celcius". "celcius“ est utilisé en entrée de l’affectation : il est utilisé mais pas modifié.

Le **nom** fahr est utilisé à gauche de l'affectation. fahr c'est le **contenant** de fahr. On parle aussi de "lvalue" ("left value", "valeur à gauche" de l'affectation, ou "valeur_g"[1]). fahr est utilisé en sortie de l’affectation : il est modifié.

Exp. 13 : les opérateurs

Dans l’expression « : 9 / 5 celcius + 32 », on utilise des opérateurs arithmétiques.

Le langage C offre les opérateurs arithmétiques classiques :

opérateurs arithmétiques binaires

+, -, *, /, et l'opérateur de modulo %

L'opérateur de modulo donne le reste de la division entière : 12 % 5 vaut 2. Il ne s'applique qu'aux entiers (et aux caractères!).

Les opérateurs binaires fournissent un résultat du même type que leurs opérandes. 2+2 est entier, 5.2 + 1.0 est flottant. Nous verrons plus loin le principe des conversions de types.

Notons ici que si on écrit : 9 / 5 * celcius, ça ne marchera pas car 9 / 5 vaut 1. Il faudrait écrire : 9 / 5.0 * celcius pour obtenir une division réelle.

*, /, % ont la même priorité et sont prioritaires sur + et - qui ont la même priorité.

opérateurs arithmétiques unaires

+, - en tant que signes ont la même priorité et sont prioritaires sur les opérateurs binaires.

(Ils sont évalués de droite à gauche).

parenthèses

L’usage des parenthèses dans les expressions arithmétiques est celui des mathématiques. Les parenthèses sont prioritaires sur les opérateurs arithmétiques.

Exp. 14 : entrer une valeur : scanf

scanf("%d", &celcius) ;

Cette instruction permet d'affecter dans celcius une valeur saisie au clavier. On parle de "lire une valeur au clavier".

La fonction scanf définie dans stdio.h. scanf veut dire scanner de manière formatée (d’où le f).

Comme le printf, le scanf possède en premier argument un **format** exprimé sous forme d'une chaîne de caractères, ici "%d". Le %d veut dire qu’on va lire un entier. Pour lire un flottant, on écrira %f pour les float, %lf pour les double, %c pour les caractères.

Les arguments suivants, il n'y en a qu'un seul dans notre exemple, précisent dans quelles variables on souhaite placer les valeurs lues. Notons qu'on écrit "&celcius" et pas "celcius". "&" est un opérateur : il fournit l'adresse d'une variable. Il faut toujours fournir l'adresse de la variable à laquelle on veut affecter la valeur lue par le scanf.

Avant le scanf, on fait un printf qui permet de dire à l’utilisateur ce qu’on attend de lui à ce moment du programme. Il faut toujours faire cela. Notez qu’on peut éviter le \n : ainsi la valeur tapée au clavier s’affiche à la suite de ce qui a été affiché.

1 K&R91 p. 196, §A5.

printf ("%d F = %d C \n", fahr, celcius);

Ici on affiche du texte et le contenu des variables. printf ça veut dire afficher (print) de façon formatée (f).

Les variables à afficher sont listées après le texte à afficher.

Les variables vont venir prendre la place des **formats** dans le texte à afficher. Ces formats ont la forme : %d pour un entier, %c pour un caractère, %f pour un flottant, %lf pour un double.

1.3 Test

1 Sujet

Ecrire un programme qui affiche la racine carrée d'un réel.

2 Analyse

Pour résoudre ce problème, il y a 3 étapes :

1) récupérer le réel
2) si le réel est négatif, il n'a pas de racine, sinon il a une racine qu'on calculera grâce à une fonction.
3) Afficher les résultats

3 Solution

```
# include <stdio.h>
# include <math.h>
void main(void)
{
    float rac, x;
    printf ("entrez un réel positif : ");
    scanf ("%f", &x);
    if (x < 0) {
        printf ("pas de racine \n");
    }
    else {
        rac = sqrt (x);
        printf ("la racine de %f = %f \n", x, rac);
    }
}
```

main.c

4 Exécution

```
$ racine ↵
entrez un réel positif : 4 ↵
la racine de 4.000000 = 2.000000
$ racine ↵
entrez un réel positif : -1 ↵
pas de racine
$ racine ↵
entrez un réel positif : 2 ↵
la racine de 2.000000 = 1.414214
$
```

5 Explications

Exp. 16 : math.h et sqrt

Il y a un include supplémentaire : **math.h.** Le fichier math.h contient la déclaration de la fonction **sqrt** utilisée dans le programme. La fonction sqrt est une fonction qui renvoie la racine carrée d'un nombre à condition qu'il soit positif.

Dans math.h, on trouve de nombreuses fonctions mathématiques : cos, acos, etc., log, log10, exp, pow(x, y) = x puissance y.

Notons que les directives include, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et qu'elles doivent obligatoirement commencer en début de ligne.

Exp. 17 : le test : déroulement du programme

Dans l'analyse, on a fait un test : si … sinon. Le test est une notion fondamentale de la programmation. Il permet de choisir si une instruction sera exécutée ou non.

C'est traduit en c par if et else

```
if (expression) {
    instructions
}
else {
    instructions
}
```

Les accolades marquent le début et la fin du bloc.

S'il n'y a pas d'alternative (de sinon) on peut se passer du else :

```
if (expression) {
    instructions
}
```

Exp. 18 : déroulement du programme, première notion de débranchement

Avec le test, les instructions ne sont plus exécutées les unes après les autres. Le test engendre des sauts, des débranchements : certaines instructions ne seront pas exécutées.

Exp. 19 : blocs et tabulations

On constate dans les exemples précédents qu'après chaque ouverture de bloc (après une accolade ouvrante), l'instruction suivante est écrite avec un décalage : une **tabulation**.

De même chaque fermeture de bloc est écrite avec une tabulation de moins par rapport à l'instruction précédente.

Bien que facultatif, c'est très important pour la clarté du programme et il faut prendre l'habitude de faire cela bien tout de suite.

C'est l'occasion du troisième précepte :

Précepte n° 3

Soignez immédiatement la syntaxe et les tabulations

Exp. 20 : opérateurs de comparaison et opérateurs logiques

if (n < 0) {

Dans le test, on trouve un opérateur de comparaison.

Les opérateurs de comparaison

On trouve en C les 6 opérateurs binaires de comparaison classiques :

>, >=, <, <=, = = , !=

A noter que égale s'écrit "= =", que différent s'écrit "!=".

Les opérateurs logiques

binaires

On trouve en C les deux opéateurs logiques classiques "et" et "ou" (* et +)

"et" s'écrit : &&

"ou" s'écrit : | |

if (a>=0 && a <100) {

L'évaluation se fait de gauche à droite et s'arrête dés que la véracité ou la fausseté est établie.

unaire

l'opérateur de négation : !, est de même priorité que les opérateurs de signe.

Priorité des opérateurs

Les opérateurs sont présentés par ordre de priorité décroissante.

Dans une même ligne du tableau, les opérateurs ont la même priorité.

OPEÉRATEURS	ASSOCIATIVITÉ	ARITÉ
()	gauche à droite	-
! + - &	droite à gauche	1

* / %	gauche à droite	2
+ -	gauche à droite	2
< <= > >=	gauche à droite	2
== !=	gauche à droite	2
&&	gauche à droite	2
||	gauche à droite	2
=	droite à gauche	2

Exp. 21 : printf % f

On affiche deux variables qui remplacent les formats dans l'ordre. Notons que, bien qu'on saisisse 4, on affiche 4,000000 car on affiche en format float : 6 chiffres après la virgule. Ce n'est pas très pratique, pas très lisible.

- On peut utiliser la variante suivante :

```
printf ("la racine de %.2f = %.4f \n", x, rac);
```

- Elle donne le résultat suivant :

```
$ racine ↵
entrez un réel positif : 2 ↵
la racine de 2.00 = 1.4142
$
```

- Ou encore la variante suivante :

```
printf ("la racine de %g = %g \n", x, rac);
```

- Elle donne le résultat suivant :

```
$ racine ↵
entrez un réel positif : 2 ↵
la racine de 2 = 1.41421
$
```

- Le format %g choisit l'affichage le plus simple.

1.4 Boucle[2]

1 Sujet

Ecrire un programme qui affiche la table des températures en degrés Fahrenheit de 0 à 300 par pas de 20 et leurs équivalents en degrés Celcius.

La formule c'est : °C = 5 / 9 *(°F – 32)

2 Analyse

Pour résoudre ce problème, il y a 4 étapes :

1) On part de 0 °F
2) On calcule le °C pour le °F en cours
3) On affiche les couple résultat.
4) On ajoute 20 au °F

Les 3 dernières étapes sont répétées tant que °F est < 300

3 Solution

```
# include <stdio.h>
/*
programme d'affichage de la table de conversion °F °C
*/
void main(void)
{
    int fahr, celcius ;
    int min, max, intervalle ;
    min = 0 ; /* borne inferieure de la table */
    max = 300 ; /* borne superieure de la table */
    intervalle = 20 ; /* intervalle entre les F */
    fahr = min ;
    while (fahr <= max) {
        celcius = 5 * (fahr - 32 ) / 9 ;
        printf ("%3d F = %3d C\n", fahr, celcius);
        fahr = fahr + intervalle ;
    }
}
```

main.c

4 Exécution

2 **Début du deuxième cours.**

```
$ table ↵
  0 F =  -17 C
 20 F =   -6 C
 40 F =    4 C
 60 F =   15 C

300 F =  148 C
$
```

5 Explications

Exp. 22 : commentaires

Le texte entre /* */ est du commentaire.

Il faut prendre l'habitude de commenter les programmes.

Bien commenter est un art difficile : il ne faut pas paraphraser le programme mais donner le sens d'une instruction ou d'un bloc d'instructions.

C'est le 4ème précepte :

Précepte n° 4
Commentez immédiatement vos programmes

Exp. 23 : la boucle tant que

while (fahr <= 300) {

Dans l'analyse, il y a une répétition d'un bloc d'instruction.

C'est ce qu'on appelle communément une boucle. La notion de boucle est une notion fondamentale de la programmation qui permet de répéter plusieurs fois des instructions sans avoir à les réécrire.

C'est traduit ici par une boucle tant que : while (tant que).

```
while (expression) {
    instructions
}
```

L'expression est évaluée. Si elle est vraie, alors les instructions sont exécutées et on recommence. Sinon on sort de la boucle et on passe à la suite.

Il existe une variante de la boucle tant que : la boucle faire tant que :

```
do {
    instructions
} while (expression)
```

Les instructions s'exécute toujours une première fois. L'expression est évaluée. Si elle est vraie, on recommence, sinon on passe à la suite.

Exp. 24 : printf %3d

printf ("%3d F = %3d C\n", fahr, celcius);

Le %3d permet de spécifier qu'on affiche le résultat sur 3 caractères. Ainsi, la liste des valeurs est cadrée à droite ce qui la rend lisible. C'est très important, non pas pour l'esthétique du programme, mais pour faciliter la lecture des résultats et donc la mise au point du programme. En effet, il y a bien sur une analyse théorique, mais il y a aussi un aspect expérimental dans la programmation. On avance aussi par essais-erreurs. Il faut donc que les essais soient aisément interprétables.

Nouveau précepte :

Précepte n°5

Ne vous occupez pas d'une belle présentation de l'exécution du programme tant que le programme n'est pas complètement terminé.

Par contre, soignez la lisibilité de l'affichage des résultats pour faciliter la mise au point du programme.

1.5 Tableau

1 Sujet

On a une série de 7 notes : 12, 15, 18, 10, 2, 6, 7. Rangez ces notes dans un tableau et calculez la moyenne de ces notes.

2 Analyse

Pour résoudre ce problème, il y a 4 étapes :

1) On initialise le tableau avec les 7 notes
2) On parcourt tout le tableau et on ajoute chaque élément à une variable somme initialisée préalablement à 0
3) On divise la somme par le nombre d'élément du tableau.
4) On affiche le résultat.

3 Solution

```
# include <stdio.h>
# define N 7

void main(void )
{
    float moy ;
    int i, somme ;
    int tab[N] = {12, 15, 18, 10, 2, 6, 7};
    somme=0 ;
    for (i=0; i < N; i++) {
        somme = somme + tab[i];
    }
    moy = (float)somme / N;
    printf ("la moyenne du tableau vaut %.1f \n",  moy);
}
```

main.c

4 Exécution

```
$ moyenne ↵
la moyenne du tableau vaut 10.0
$
```

5 Explications

Exp. 25 : #define

 # define N 7

C'est une directive pour le préprocesseur. Elle demande de remplacer dans tout le programme le symbole N par 7. Attention, ce remplacement est fait comme dans un traitement de texte : toutes les occurrences de N en tant que mot sont remplacées ! !

Exp. 26 : int tab[N]

int tab[N] = {12, 15, 18, 10, 2, 6, 7};

Un tableau est un nouveau type de variable qui permet de regrouper dans une seule variable plusieurs valeurs de même type.

tab est un tableau de N, c'est-à-dire 7, entiers.

Pour accéder à un élément du tableau, on écrit par exemple : tab[2]. Le premier élément du tableau c'est tab[0] ; tab[0] c'est une variable comme une autre, ici de type int.

Dans notre exemple, le tableau est initialisé avec la liste des valeurs entre accolades : tab[0] vaut 12, tab[5] vaut 6.

Exp. 27 : la boucle for

for (i=0; i < N; i++) {

C'est un troisième type de boucle. On l'utilise le plus souvent pour parcourir tous les éléments d'un tableau. Cette boucle permet de gérer un compteur qui est incrémenté à chaque pas de la boucle.

```
for (i=0; i < N; i++) {
    instructions
}
```

Le compteur c'est i : il est initialisé à 0 au début de la boucle. Ensuite on teste : est-ce que i est < à N ? si oui on exécute les instructions, sinon on quitte la boucle. Si on a exécuté les instructions, quand on arrive à l'accolade fermante, on exécute la troisième instruction de la boucle : i++ : c'est une incrémentation : cela équivaut à i=i+1.

Exp. 28 : opérateurs d'incrémentation[3]

i++ ; c'est équivalent à i=i+1 ;

L'opérateur ++ permet d'incrémenter de 1 une variable. Il peut être mis avant ou après la variable. On peut écrire i++; ou ++i ;

Idem pour l'opérateur -- qui permet de décrémenter de 1 une variable.

3 Bra98 p. 128.

Exemple

```
n = 5; /* n vaut 5 */
x = n++; /* post incrémentation : x vaut 5, n vaut 6 */
x = ++n; /* pré incrémentation :: x vaut 7, n vaut 7 *:
```

Post incrémentation

x = 3 + n++; équivaut à : x = 3 + n ; n=n+1 ;

"op n++" ou "n++ op" : on utilise n avec op avant de d'incrémenter n.

L'incrémentation a lieu **« postérieurement » à l'évaluation de l'expression** dans laquelle il se trouve.

Le ++ est placé après l'opérande sur lequel il porte.

Pré incrémentation

x =3 + ++ n ; équivaut à : n=n+1 ; x=3 + n ;

"op++n" ou "++n op" : on utilise n avec op après avoir incrémenter n.

l'incrémentation a lieu : **« antérieurement » à l'évaluation de l'expression** dans laquelle il se trouve.

Le ++ est placé avant l'opérande sur lequel il porte.

Restrictions

L'incrémentation ne s'applique qu'à des variables (des lvalues). (i+j)++ est illégale.

Exp. 29 : le cast : (float)

moy = (float)somme / N;

Pourquoi n'a-t-on pas écrit : moy = somme / N;

Parce que cette division serait une division entière, or on veut une division réelle. Pour cela on va changer le type d'une des variables. Ce changement n'est valide que dans l'expression. Pour changer le type on écrit, devant la variable : (nouveau type).

On pourrait aussi régler le problème en déclarant somme en float.

Attention : (float)somme/N ce n'est pas la même chose que float(somme/N).

1.6 Fonction

1 Sujet

Ecrire une fonction qui renvoie la soustraction de deux entiers et le programme qui permet de l'utiliser.

2 Analyse de la fonction

Entrée : deux entiers, a et b

Sortie : le résultats de a - b

Etapes : on soustrait a et b et on renvoie le résultat.

3 Analyse du *main*

3 étapes :

1) lire les 2 nombres
2) appeler la fonction avec les 2 nombres
3) afficher le résultat

4 Solution

```
# include <stdio.h>

int soustraction (int a, int b);

void main(void)
{
    int entier1, entier2, res;
    printf ("entrez deux entiers : ");
    scanf ("%d %d", & entier1, & entier2);
    res = soustraction (entier1, entier2);
    printf ("%d - %d = %d\n", entier1, entier2, res);
}

/* ================================================
    fonction soustraction: renvoie la soustraction
    des 2 valeurs fournies en entrée
 ================================================ */
int soustraction (int a, int b)
{
    int res ;
    res = a-b ;
    return res ;
```

```
}
```
main.c

5 Exécution

```
$ main ↵
entrez deux entiers : 6 4 ↵
6 - 4 = 2
$
```

6 Explications

Exp. 30 : écrire une fonction

La fonction soustraction

Dans ce programme, il y a une deuxième fonction : soustraction. La notion de fonction est une notion fondamentale de la programmation structurée qui permet de décomposer un programme en sous-programmes.

La fonction soustraction est décrite comme le *main* : elle a une entête et un corps.

Entête : int soustraction (int a, int b)

L'entête a 3 parties :

- le type de la valeur retournée par la fonction : ici int
- le nom de la fonction : ici soustraction
- entre parenthèses, la liste des **paramètres** dits **formels** de la fonction : ici (int a, int b)

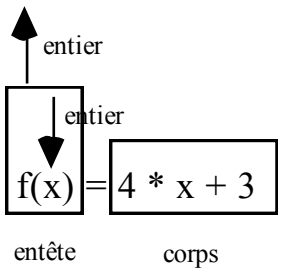
Pour chaque paramètre formel, on précise son type et son nom. S'il y en a plusieurs, on sépare les couples type-nom par des virgules.

Corps de la fonction

Le corps de la fonction est comme celui du *main* : il débute par une accolade ouvrante qui marque le début de la fonction en tant que bloc d'instructions et se termine par une accolade fermante qui marque la fin de la fonction. Dans le bloc, on trouve les instructions.

Entrées et sorties d'une fonction

Les paramètres fournis à la fonction sont des entrées. Le résultat de la fonction est une sortie.



Dans cet exemple, on dit qu'on a 1 paramètre en entrée et 1 en sortie. On peut avoir autant de paramètres en entrée qu'on veut. Par exemple : f(x, y). On verra qu'on peut aussi avoir plusieurs paramètres en sortie, mais cela fera intervenir la notion de pointeur.

La valeur retournée par la fonction sera appelée la **sortie standard**.

On a déjà vu que le *main* est une fonction comme une autre, qui n'a ni paramètre en entrée, ni paramètre en sortie : void main(void). Toutes les fonctions peuvent n'avoir aucun paramètre en sortie (par exemple une fonction qui ne fait que de l'affichage).

Fonction et procédure

Une fonction est utilisable directement dans une expression comme n'importe quel opérande. Par exemple, la fonction sinus a un argument en entrée (un angle) et ressort une valeur. On peut écrire : 4 * sin (x).

C'est ce qui caractérise une fonction et ce qui la différencie d'une procédure.

En C, tous les modules s'appellent des fonctions. Dans d'autres langages on parle de procédure quand on ne peut pas utiliser le module directement dans une expression, de fonction quand on le peut.

Intérêts d'une fonction

Les fonctions ont deux intérêts : d'une part elles permettent de répéter plusieurs fois une même séries d'instructions ; d'autre part elles permettent de structurer le programme et l'analyse en parties autonomes : ainsi les choses sont plus claires, plus facile à comprendre, plus facile à reprendre (mettre à jour, *main*tenir).

C'est l'occasion d'un nouveau précepte :

Précepte n°6

Construisez votre programme avec des fonctions

Exp 31 : return : débranchement

return res ;

Les deux usages du return :

- return permet de renvoyer la valeur d'une expression (une constante, une variable ou un calcul) sur la sortie standard de la fonction.
- return marque aussi la fin de la fonction : return fait quitter la fonction.

Le return est ce qu'on appelle un débranchement.

Exp 32 : appeler une fonction

res = soustraction (entier1, entier2);

La fonction soustraction est appelée dans le *main*.

Les variables ou valeurs passées en paramètres à une fonction au moment de son appel sont appelées les **paramètres d'appel**.

Notez que le nom des paramètres formels (a et b) est sans rapport avec le nom des paramètres d'appel (entier1 et entier2). L'important c'est l'ordre : entier1 vient prendre la place de a et entier2 celle de b.

La sortie standard de la fonction, comme dans le cas de sqrt, est une valeur qui peut être affectée dans une variable.

Exp 33 : déroulement du programme

Le déroulement général du programme ne change pas : il consiste à exécuter toutes les instructions du *main* les unes après les autres.

Mais, dans le *main*, il y a un appel de la fonction soustraction. On part alors dans la fonction soustraction en remplaçant les paramètres formels par les valeurs fournies à l'appel dans le *main*.

La fonction s'arrête après le return et on repart dans le *main*. La valeur renvoyée est utilisée ou pas dans le *main*.

Exp 34 : prototype

int soustraction (int a, int b);

Le prototype c'est l'entête de la fonction. Il faut le déclarer au début du programme, avant le *main*, afin qu'elle soit reconnue par le compilateur dans le *main*.

L'inclusion des fichiers ".h" avec le #include consiste à ajouter les prototypes des fonctions utilisées (ici le printf et le scanf). Cf. explication 4.

1.7 Conclusion : rappel des notions fondamentales

1 Format général d'un programme C

```
#include

#define

prototypes

main
{
        instructions
}

fonction 1
{
        instructions
}

fonction 2
{
        instructions
}
```

2 La programmation : des données et des traitements

La programmation, quel que soit le langage, distingue toujours deux notions : les données et les traitements. C'est une chose à retenir. C'est une dichotomie aussi importante que celle entre hardware et software.

Pour le moment on a vu :

Données :

- les variables
- les tableaux
- les types

Traitements :

- écrire
- lire
- affectation
- test
- boucle
- fonction

Ces premières notions, présentées succinctement, sont fondamentales et aussi suffisantes pour commencer à programmer. Il faut les avoir bien comprises. Les chapitres suivants vont approfondir certaines de ces notions et en présenter de nouvelles.

Mais, redisons le une fois encore, l'essentiel à comprendre se trouve, dans un premier temps, dans ces premières notions qui conditionnent la compréhension de l'informatique en générale.

CHAPITRE 2 : LES 4 TYPES DE BASE

Ce chapitre présente les types simples du langage C et leurs utilisations avec des variables, des constantes et des lectures et des écritures.

Le langage C ne comporte que 4 types de base : "char", "int", "float" et "double".

On peut préciser ces types par 4 attributs : "short", "long", "signed" et "unsigned".

- "short", "long" modifient le nombre d'octets sur lesquelles sont codés les types de bases.
- "signed" et "unsigned" modifient le do*main*e de valeurs du type.
-

On pourra caractériser une variable par l'attribut : "const" qui indiquera que les valeurs de la variable ne seront jamais modifiées.

RETENIR : char, int, long, float, double, const.

2.1 Les caractères : char

1 Il existe 3 types de caractères

"char", "signed char", unsigned char".

Les trois types sont codés sur 1 octet et peuvent donc prendre 256 valeurs différentes.

Les "signed char" vont de -128 à 127, les "unsigned char" de 0 à 255.

Les "char" sont signés ou pas selon les machines. Mais dans tous les cas, les caractères imprimables sont toujours positifs.

RETENIR : on n'utilise que le type "char" !

2 Codage, code ASCII et type

- Un octet = 8 bit, un bit c'est deux valeurs possibles, par convention 0 et 1. Un octet peut donc coder 2 puissance 8 = 256 valeurs.
- Code ASCII : codage standard de 128 caractères (de 0 à 127)

  0-9 : 48-57

  A-Z : 65-90

  a-z : 97-122
- exemple

  en binaire : A = 0 1 0 0 - 0 0 0 1

  en hexadécimal : A = 4 1 (c'est une autre forme de représentation)

- type

  Une variable de 1 octet, codée 0 1 0 0 - 0 0 0 1, vaut 'A' si son type est caractère, vaut 65 si son type est entier sur un octet.

  Ce qui compte, c'est l'interprétation que l'on fait du codage, c'est-à-dire le type.

3 Déclaration des variables

```
char c;
signed char sc;
```

- Quand on déclare une variable on crée un contenant avec un nom et un type. On dit aussi qu'on alloue de la mémoire auquel on donne un nom et un type.
- Une variable a toujours une valeur. En effet, quand on crée la variable c, on crée un contenant de 1 octet. Les bits de cet octet sont forcément dans un certain état (0 ou 1). Donc la variable a une valeur.

4 Déclaration des constantes

une constante est une variable dont on ne peut pas modifier la valeur.

```
const char retour = '\n';
```

5 Affichage des caractères

```
char cara;
cara ='A'
printf("%c\n", cara);
```

6 Lecture des caractères

```
char cara;
scanf("%c", &cara);
```

A noter : l'usage de l'opérateur & qui permet de passer l'adresse de cara.
La fonction scanf pose des problèmes avec la lecture des caractères.

Pour les éviter, deux solutions :

- fflush : il faut faire précéder tous les scanf de %c d'un fflush(stdin) .

```
char cara;
fflush(stdin);
scanf("%c", &cara);
```

- la fonction getchar permet aussi de lire des caractères :

```
char cara;
cara = getchar();
```

7 Notation des constantes caractères

Cas général

Entre apostrophes (quottes simples) : 'a' 'Z' '*' '$'.

Code ASCII

On peut écrire un caractère à partir de son code ASCII, exprimée en octale ou en hexadécimale.
Pour ça, on fera précéder :

- • le nombre hexadécimal d'un : \x suivit de deux chiffres hexa (de 0 à F)
- • le nombre octal d'un : \ suivit de trois chiffres octal (de 0 à 7)

Exemples :
Les tois notations suivantes sont équivalentes :
caractère : 'A' hexa : '\x41' octal '\101'

8 Affichage

```
char c1, c2;
c1 ='A';
c2 ='\x43';
printf("<%c> <%c> <%c> <%c> <%c>\n",c1,66,c2,'\104','E');
```

```
<A> <B> <C> <D> <E>
```

Notez que les entiers sont considérer comme des valeurs ASCII.

9 Cas de l'antislash

L'antislash est un caractère ignoré sauf quand il est suivi d'un x (pour la notation hexa), d'un chiffre de 0 à 7 (pour l'octal), ou d'un caractère du tableau ci-dessous[4], auquel cas il aura un rôle particulier dans l'affichage.

Par exemple, on peut « afficher » :

un bip :	\a
un saut de ligne :	\n
une tabulation horizontale :	\t
un antislash :	\\

4 Del97 p. 23.

Toutes les autres possibilités sont dans ce tableau, à consulter à l'occasion.

Notation C	Code ASCII	Abréviation usuelle	Rôle
\a	7	BEL	Bip
\b	8	BS	retour arrière (backspace)
\f	12	FF	saut de page (form feed)
\n	10	LF	saut de ligne (line feed)
\r	13	CR	retour chariot (carriage return)
\t	9	HT	tabulation horizontale (horizontal tab)
\v	11	VT	tabulation verticale (vertical tab)
\\	92	\	\
\'	39	'	
\"	34	"	
\?	63	?	

2.2 Les entiers : int et long

1 Il existe 6 types d'entiers (c'est-à-dire 6 do*main*es de valeurs)

"int", "short", long", "unsigned int", "unsigned short", unsigned long".

Un même type peut être nommé de différente manière. Par exemple, les types "short", "short int", "signed short int", "signed short" veulent dire la même chose. Le mieux est de se limiter aux int et aux long.

RETENIR : int, long

Chaque type impose sa taille et ses valeurs limites

Tous les int n'ont pas la même taille sur toutes les machines (ou, plus précisément, sur tous les processeurs) et sur tous les compilateurs.

Traditionnellement, sur PC, le "short int" est identique au "int"[5] : les deux sont codés sur 2 octets. Cependant, aujourd'hui avec visual-C++, les int sont codés sur 4 octets, comme les long.

Si on veut faire rentrer deux octets dans 4 octets, on n'aura jamais de problème (compatiblité ascendante). Par contre, si on veut faire rentrer 4 octets dans 2 octets, on aura des problèmes pour tous les entiers qui vont au-delà du domaine de valeurs sur 2 octets.

Chaque compilateur peut choisir la taille, mais :

- la taille minimum des short et des int doit être de 16 bits.
- la taille minimum des long doit être de 32 bits.
- les long doivent être plus longs que les int.

Nom du Type	**Format**	**Do*main*e de valeurs**
Int	**32 bits, signé (4 octets)**	**-2 147 483 648 à +2 147 483 647**
Short	16 bits, signé (2 octet)	-32 768 à +32 767
Long	32 bits, signé (4 octets)	-2 147 483 648 à +2 147 483 647
Unsigned int	16 bits, signé (2 octets)	0 à +4 294 967 295
Unsigned short	16 bits, signé (2 octet)	0 à +65 535
Unsigned long	32 bits, signé (4 octets)	0 à +4 294 967 295

5 Del97 p. 19, K&R91 p. 36, Bra98 p. 51.

Opérateur sizeof

Sizeof est un opérateur du langage C qui permet de connaître la taille en octet d'un type.
Sur visual-C++ :

- sizeof (int) vaut 4
- sizeof (short) vaut 2
- sizeof (long) vaut 4

Définition des valeurs limites

Toutes les valeurs limites sont définies dans le fichier **<limits.h>** :

Nom	**Signification**	**Limite**
SHRT_MIN	valeur short minimale	-32 768
SHRT_MAX	valeur short maximale	32 767
USHRT_MAX	valeur unsigned short maximale	65 535
INT_MIN	valeur int minimale	-2 147 483 648
INT_MAX	valeur int maximale	2 147 483 647
UINT_MAX	valeur unsigned int maximale	4 294 967 295
LONG_MIN	valeur long minimale	-2 147 483 648
LONG_MAX	valeur long maximale	2 147 483 647
ULONG_MAX	valeur unsigned long maximale	4 294 967 295

Remarques méthodologiques

- Dans un souci de portabilité, on choisira toujours le type int (signed ou unsigned).
- On évitera le type short quand il n'est pas nécessaire d'optimiser la taille.
- On évitera le type long si la limite n'est pas dépassée.
- On évitera l'usage des valeurs limites quand il n'y a pas de risque de débordement.

2 Déclaration des variables

```
int a, b, entier;
short s;
long l;
short int x;
signed long int;
```

3 Déclaration des constantes

```
const int a = 8;
```

4 Initialisation des variables

```
int a=0;
short b;
b=5;;
```

5 Notation des constantes entières

Les constantes sont typées.

int

+233 32 -145

long : l ou L derrière (ou rien par défaut)

123 456 789 123 456 789L 123 456 789l 125L

unsigned : u ou U derrière

123u 125U

hexadécimal : Ox ou OX devant (zéro devant)

0x1F en hexadécimal vaut 31 en décimal : 16 + 15(F) = 31.
0x1FUL en hexadécimal vaut 31 en décimal (unsigned long)

octal : O devant (zéro devant)

037 en octal vaut 31 en décimal : 3*8+7=31.

6 Affichage des entiers

```
int n, l;
n = 65;
l = 0101;
printf("<%d> <%d> <%d> <%d> <%d>\n", n, 'A', l, 0x41, 65);
printf("<%5d>\n", n);
printf("<%d> <%d> <%d> <%d> <%d>\n",n,66,'A'+2,'\104','E');
```

```
<65> <65> <65> <65> <65>
<   65>
<65> <66> <67> <68> <69>
```

7 Lecture des entiers

```
char n;
scanf("%d", &n);
```

A noter : l'usage de l'opérateur & qui permet de passer l'adresse de n.

2.3 Les flottants : float et double[6]

Le type flottant permet de représenter, de manière approchée, une partie des nombres réels. D'une manière seulement approchée car si le nombre est trop grand, on ne peut pas représenter tous les chiffres qui le composent, de même s'il a trop de chiffres après la virgule.

1 Il existe 3 types de réels (appelés flottants)

"float", "double", long double".

short, signed et unsigned ne s'appliquent pas aux float.

Comment le C représente-t-il les flottants ?

Par convention, on considère que le nombre

123,456789

est codé par le couple (1.234567 ; 2) et s'écrit :

1.234568 E+2.

On considère donc les flottants comme constitués de :

- une mantisse comprise entre 1 et 9 (1 dans notre exemple)
- 6 chiffres après la virgule (234568 dans notre exemple : il y a un arrondi sur le dernier chiffre)
- un exposant (+2 dans notre exemple)

Seuls ces 7 chiffres sont garantis sans approximation.

On va donc définir pour chaque type :

- • la puissance max et min (soit les bornes du do*main*e de valeurs, mais pas toutes les valeurs possibles);
- • le nombre maximum de chiffres après la virgule sans erreur d'arrondi ;
- • l'epsilon : la différence entre 1 et la plus petite valeur représentable supérieure à 1. Il va servir à faire des comparaisons d'égalité entre flottants.

Ici nous ne donnerons que les caractéristiques des float et des double. C'et bien suffisant pour la plupart des calculs. Si on fait des calculs scientifiques poussés, il vaut faire attention et utiliser des bibliothèques et des processeurs spécialisés.

6 **Début du troisème cours.**

Toutes les valeurs limites sont définies dans le fichier **<float.h>** :

Nom[7]	**Signification**	**Valeur**
float : sur 4 octets		
FLT_DIG	Précision, en chiffres décimaux	6
FLT_MIN	Plus petit nombre	1 E-38
FLT_MAX	Plus grand nombre	1 E+38
FLT_EPSILON	Différence entre 1 et la plus petite valeur représentable supérieure à 1	0.000000119209 (6 zéros après la virgule)
double : sur 8 octets		
DBL_DIG	Précision, en chiffres décimaux	15
DBL_MIN	Plus petit nombre	1 E-308
DBL_MAX	Plus grand nombre	1 E+308
DBL_EPSILON	Différence entre 1 et la plus petite valeur représentable supérieure à 1	0.00000000000000022204 (15 zéros après la virgule)

2 Déclaration des variables

```
float x, y, reel;
double dx;
long double ld;
```

3 Déclaration des constantes

```
const float eps = 1.0e-5;
const double = 3.1415926535;
```

4 Comparaison des flottants

Les calculs flottants sont toujours approximatifs. Il faut donc faire très attention quand on fait des comparaisons :

il ne faut pas écrire :

```
if (f1 = = f2) {
```

mais :

```
if (f1 - f2 < FLT_EPSILON) {
```

7 Bra98 p. 56.

5 Affichage des flottants

1er exemple

```
    float f;
    f=123.123456789;
    printf("<%f> <%e> <%g>\n", f, f, f);
    f=123456789;
    printf("<%f> <%e> <%g>\n", f, f, f);
```

```
<123.123454> <1.231234e+002> <123.123>
<123456792.000000> <1.234568e+008> <1.234567e+008>
```

A noter dans cet exemple :

- 8 chiffres corrects avec le %f, dont le 8ème parfois avec un arrondi.
- 6 chiffres après la virgule avec le %e, dont le 6ème avec un arrondi.
- le format le plus simple avec %g

=> 7 chiffres significatifs

%f et %e propose toujours 6 chiffres après la virgule, le dernier est arrondi si nécessaire.

2ème exemple

```
#include <stdio.h>
#include <float.h>
void main () {
    float f;
    double d;
    f=3.1415926535898123 45;
    d=3.1415926535898123 45;
    printf("<%.20e> <%d> \n", f,FLT_DIG);
    printf("<%.20e> <%d> \n", d,DBL_DIG);
}
```

```
<3.14159274101257324000e+00> <6>
<3.14159265358981221000e+00> <15>
```

A noter dans cet exemple :

- 6 chiffres corrects après la virgule pour le float. Au delà c'est faux.
- 15 chiffres corrects après la virgule pour le double. Au delà c'est faux.
- 6 et 15 correspondent à FLT_DIG et DBL_DIG

3ème exemple

```
f=1.0000111111;
printf("<%.20f><%.20f>\n", f, FLT_EPSILON);
d=1.000000000111111;
printf("<%.20f><%.20f>\n", d, DBL_EPSILON);
```

```
<1.00001108646392822000><0.00000011920928955078>
<1.00000000011111090000><0.00000000000000022204>
```

A noter dans cet exemple :

- 6 chiffres corrects après la virgule pour le float. Au delà c'est faux.
- 14 chiffres corrects (seulement !) après la virgule pour le double. Au-delà c'est faux.
- On constate que les epsilon commence au 7ème et 16ème chiffre après la virgule.

Conclusion :

Prudence dans les calculs scientifiques complexes !!!

6 Notation des constantes flottantes

Par défaut : double

On peut donc écrire :

12.43	-0.38	-.38	4.	.27	
4.25E4	4.25e4	42.5e3	425E2	425.e13	425.0e13

float : F ou f derrière

12.43F

long double : L ou l derrière

12.43l

7 Lecture des flottant

```
float x;
scanf("%f", &x);
```

A noter : l'usage de l'opérateur & qui permet de passer l'adresse de x.

2.4 Codes de conversion des printf et des scanf

Avec les lectures et les affichages, on a vu un certain nombre de codes de conversion pour les printf et les scanf. Généralisons.

1 Les codes de conversion du printf [8]

Caractères	Type de l'argument	Type d'impression
d, i	int ou char	nombre décimal
o	int	nombre octal non signé
x, X	int	nombre hexa non signé
u	unsigned int, u char, u short	nombre décimal non signé
c	char, short ou int	caractère isolé
s	char *	chaîne de caractères
f	double	notation décimale de la forme [-]m.dddddd, où le nombre de d est donné par la précision (6 par défaut)
e, E	double	[-]m.ddddddeou E+ou-xx, où le nombre de d est donné par la précision (6 par défaut). m est alors compris entre 0 et 9.
g, G	double	équivaut à %eouE si l'exposant est inférieur à -4 ou supérieur ou égal à la précision; sinon, équivaut à %f.
p	void *	pointeur : représentation dépendant de l'implémentation.
%		imprime un %

8 K&R91 p. 152.

2 Les codes de conversion du scanf [9]

Caractères	Type de l'argument	
c	char *	caractère, sans apostrophe
d	int *	entier décimal
f ou e ou E ou g	float *	nombre en virgule flottante comportant éventuellement signe, point, exposant
u	unsigned int *	entier non signé sous forme décimal
hd	short int *	entier court sous forme décimal
hu	unsigned short *	entier court et non signé sous forme décimal
ld	long int *	entier long sous forme décimal
lu	unsigned long *	entier long non signé sous forme décimal
lf ou le	double *	nombre en virgule flottante long
x	int *	héxadécimal, précédé ou non de 0x
o	int *	octal, précédé ou non de 0
i	int *	entier décimal, octal (précédé par un 0) ou hexadécimal (précédé par 0x ou 0X)
s	s *	chaîne de caractères (sans guillemets)

On peut paramétrer plusieurs éléments du format: le code de conversion (%d), le gabarit maximum et de qui concerne les règles de lecture. On n'aborde *main*tenant que le format

RETENIR : %c, %d, %f, %s, %e, %g

9 Del97 p. 59.
K&R91 p. 156.

2.5 Les constantes symboliques

1 Définition

La directive define permet de définir ce qu'on appelle une constante symbolique (ou nom symbolique ou symbole) :

```
#define symbole texte-de-remplacement
```

- La constante symbolique sera remplacée à chacune des ses occurrences en tant que mot dans le programme, après sa définition, par le préprocesseur.
- Les constantes symboliques ne correspondent pas à des variables, elles définissent une équivalence entre deux textes et sont remplacées avant la compilation : autrement dit la constante symbolique n'existe plus dans le programme une fois qu'il est compilé.
- On ne peut définir qu'un symbole par #define.
- Il n'y a pas de point virgule à la fin de la ligne.
- En général, les #define apparaissent au début du programme (juste après les #include qui servent, entre autre, à inclure des #define).

2 Attention !

le préprocesseur ignore la syntaxe du C et remplace toutes les occurrences de la constante symbolique par son texte de remplacement.

Exemple :

```
#define max 100
...
int max;
...
```

Il y aura remplacement de la variable max par 100 ! Donc, sera compilé : "int 100" qui ne veut rien dire.

Par contre, si on écrit #define m 100, le m ne sera pas remplacé dans max car le préprocesseur cherche les mots du programme.

3 Conseil

Les constantes symboliques sont écrites en majuscules avec des traits d'union bas (underscore : _).

Les identificateurs de type, de variable et de fonction s'écrivent en minuscule, avec une majuscule au début de chaque nouvelle partie significative du nom, à l'exception de la première.

C'est l'occasion d'un nouveau précepte :

Précepte n°7

Utilisez des règles pour nommer les variables, les fonctions, les types, etc. et tenez y vous.

CHAPITRE 3 : INSTRUCTION, EXPRESSION ET STRUCTURES DE CONTROLE

3.1 Présentation

Un programme est constitué d'un certain nombre d'**instructions impératives** : fait ci, fait ça.
Par exemple :

- met 2 dans i;
- si a est positif, met 5 dans b, sinon affiche bonjour.

On peut considérer qu'il y a 7 types d'instruction :

1. la déclaration de variable (et de type)
2. l'affectation
3. l'évaluation d'une expression
4. le test
5. la boucle
6. le débranchement
7. l'appel de fonction

En général, les instructions peuvent en imbriquer d'autres :

- Une déclaration de variable peut imbriquer une affectaion.
- Une affectation peut imbriquer un appel de fonction ou l'évaluation d'une expression ou une autre affectation.
- L'évaluation d'une expression peut imbriquer un appel de fonction ou une affectation.
- Le test et la boucle peuvent imbriquer tous les autres types d'action.
- L'appel de fonction peut imbriquer l'évaluation d'une expression.

Par exemple :

- a = 5; affectation simple
- a=b=3; affectation multiple;
- x=sqrt(5); affectation et appel de fonction;
- printf("bonjour"); appel de fonction simple;
- etc.

En général, on appelle **structures de contrôle** les instructions de type **test** ou **boucle**.

3.2 L'affectation

1 Affectation ordinaire

```
a = 5 + 3;
b = a;
```

bien sur, "a + 4 = c" n'a pas de sens. La partie gauche de l'affectation (on parle de "lvalue", pour left value) doit être une variable et pas une expression ni une contante.

2 Successions d'affectation

```
a = b = 5;
```

L'évaluation d'une telle expression est fonction de l'associativité de l'affectation qui est de droite à gauche. L'instruction précédente est donc équivalente à :

```
a = (b = 5);
```

C'est donc b = 5 qui est évalué en premier, ensuite, a = b. L'instruction est donc aussi équivalente à :

```
b = 5;
a = b;
```

3 Conversion automatique

Le résultat de l'expression à droite de l'affectation est convertie dans le type de la variable affectée (on parle de "lvalue"[10] pour "left value", valeur à gauche de l'affectation, autrement dit, modifiée, en sortie. Une lvalue, c'est une variable en tant que contenant. Cette notion de "lvalue" sera réintroduite avec les pointeurs et les tableaux).

Attention, les conversions automatique peuvent engendrer des pertes d'informations si on ne va pas dans le sens : du plus petit au plus grand

10 Del97 pp 38, 134, 138.

3.3 Test : le if

```
if (expression) {
    instructions
}
else {
    instructions
}
```

Les accolades marquent le début et la fin du bloc.
Quand il n'y a qu'une seule instruction, on peut se passer des accolades.

```
if (expression)
    instruction
else
    instruction
```

3.4 Expression

En C, comme dans la plupart des langages, il y a des règles de formations des expressions. L'application de ces règles permet d'obtenir des expressions bien formées.

Dans une version simplifiée, on peut considérer qu'il y a quatre règles de formation des expressions :

- Une **expression** est formée soit par un **opérande,** soit par un **opérateur** associé avec avec des **opérandes**.

  exemples: 4 x f(3) 4+3

- Les **opérandes** peuvent être des **constantes** (4), des **variables** (x), des **appels de fonctions** (f(x)), ou des **expressions** (4+x).
- Les **paramètres d'appel** des fonctions sont des **expressions**.

  exemples: f(3) f(x) f(g(3)) f(4+x)

- Les **opérateurs** sont ceux de l'arithmétique (+, -, *, etc.), ceux de la logique (||, etc.), et les opérateurs propre au C (&, etc.) dont l'affectation.
-

L'évaluation d'une expression consiste à détermier la valeur de cette expression.

3.5 Le type logique

Il n'y a pas de type logique en C. Mais, par définition, une expression qui vaut 0 est fausse, une expression qui ne vaut pas 0 est vrai.

logique	**entier**	**hexa**	**octal**	**caractère**	**pointeur**
vrai	≠ 0	≠ '\x00'	≠ '\000'	≠ '\0'	≠ NULL
faux	0[11]	'\x00'	'\000'	'\0'	NULL

Remarque : les chaînes de caractères ne sont pas des éléments d'une expression car ce sont des tableaux. Quand on teste une chaîne de caractères, c'est en réalité le pointeur sur le premier élément du tableau qu'on teste.

On peut écrire :

```
if (correct == 0)
```

ou

```
if (!correct)
```

ou

```
if (correct)
```

ou

```
#define FAUX 0
…
if (correct == FAUX)
```

if (« bonjour ») ou if('A') ou if ('0') est toujours vrai.

if ('\x00') est toujours faux.

11 Bra98 p. 118.

3.6 Test : le “else if”

Le else if n'est pas une instruction particulière. C'est toujours un if. Son principe est que l'alternative "else" ne comprend qu'une instruction débutant par un if. Du coup, on peut se passer des accolades pour le bloc et on met le else et le if sur la même ligne (certain langage introduise le mot clé "elsif").

```
if (a < 0) {
    instructions
}
else if (a == 0) {
    instructions
}
else if (a < 100) {
    instructions
}
else {
    instructions
}
```

C'est équivalent à :

```
if (a < 0) {
    instructions
}
else {
    if (a == 0) {
        instructions
    }
    else {
        if (a < 100) {
            instructions
        }
        else {
            instructions
        }
    }
}
```

On voit bien que la deuxième notation est beaucoup moins lisible que la première.

3.7 Test : le switch

```
switch (expression) {
    case expression-constante: instructions
    case expression-constante: instructions
    …
    default : instructions
}
```

A la différence du else if, le switch ne permet de tester qu'une seule variable avec l'opérateur d'égalité.

```
switch (n) {
    case 0:
        printf("passage en 0/n");
        break;
    case 1:
        printf("passage en 1/n");
        break;
    case 2:
        printf("passage en 2/n");
        break;
    default:
        printf("instructions par défaut/n");
}
```

Dans cette écriture, si n = 0,

```
passage en 0
```

si n = 1,

```
passage en 1
```

si n = 2,

```
passage en 2
```

Avec le break :

A noter l'usage du break : c'est un débranchement qui permet de quitter le switch .

Sans le break :

```
switch (n) {
    case 0:
        printf("passage en 0/n");
    case 1:
        printf("passage en 1/n");
    case 2:
        printf("passage en 2/n");
    default:
        printf("instructions par défaut/n");
}
```

Dans cette écriture, si n = 0,

```
passage en 0
passage en 1
passage en 2
```

Cette structure présente peu d'intérêt : il faut toujours mettre des break dans les switch ;

3.8 La boucle for : pour i de 1 à N

```
for (i=1; i <= N; i++) {
    instructions
}
```

Formulation générale :

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

On peut aussi écrire, s'il n'y a qu'une instruction :

```
for (exp1; exp2; exp3)
    instruction
```

La boucle for est constituée de :

- Une entête de boucle : for (exp1; exp2; exp3) {
- Des instructions
- Une fin de boucles : }

Les 5 étapes du déroulement d'une boucle for :

1) La première expression (exp1) est évaluée avant d'entrer dans le for.

- La deuxième expression (exp2) est évaluée

2) Si exp2 est vrai, les instructions sont exécutées,
   sinon, on passe après la fin de la boucle et l'exécution de la boucle est finie.
4) La troisième expression (exp3) est évaluée après l'exécution des instructions
5) On revient en 2

Il y a un précepte de programmation associé à la boucle for qui dit :

Précepte n° 8

Ne touchez pas aux paramètres d'une boucle for dans les instructions du corps de la boucle

3.9 La boucle while : boucle tant que

```
while (expression) {
    instructions
}
```

On peut aussi écrire, s'il n'y a qu'une instruction :

```
while (expression)
    instruction
```

L'expression est évalué. Si elle n'est pas nulle, alors les instructions sont exécutées et on recommence. Sinon on sort de la boucle.

Équivalence entre while et for

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

est équivalent à

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

sauf en ce qui concerne le continue (on va voir ça un peu plus bas).

3.10 La boucle do while: répéter tant que

```
do {
    instructions
} while (expression)
```

Les instructions s'exécutent. L'expression est évaluée. Si elle est vraie on recommence, sinon on passe à la suite (à ne pas confondre avec la boucle répéter jusqu'à ce que qu'on trouve dans d'autres langages).

3.11 Boucle : les boucles sans fin

1 boucle for

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

2 boucle while

```
while ( 1 ) {
    instructions
}
```

3.12 Les débranchements

Un débranchement est une instruction qui permet d'arrêter le déroulement séquentiel des instructions.

1 Le break

On a déjà vu le break dans les switch.

Il permet aussi de quitter une boucle prématurément.

Il doit forcément être dans un test, sinon toutes les instructions qui sont derrière lui deviennent inutiles.

Exemple :

```
boucle (…) {
    if (condition) break;
    instructions
}
```

C'est équivalent à :

```
boucle (…) {
    if (condition){
        break;
    }
    else {
        instructions
    }
}
```

Le principal intérêt du break en programmation structurée, c'est de traiter à part les cas particuliers et du coup de diminuer le niveau d'imbrication des instructions du cas général.

2 Le continue

Il permet de sauter à la fin de la boucle, et donc de repartir en début de boucle.

Exemples :

sans continue :

```
for (i = 0; i < n ; i ++) {
    if (tab [i] < 0 ) {
        instructions du cas particulier
    }
    else {
        instructions du cas général
    }
}
```

avec continue :

```
for (i = 0; i < n ; i ++) {
    if (tab [i] < 0 ) {
        instructions du cas particulier
        continue ;
    }
    instructions du cas général
}
```

Le principal intérêt du continue en programmation structurée, c'est de traiter à part les cas particuliers et du coup de diminuer le niveau d'imbrication des instructions du cas général.

3 Le return[12]

Le return a deux fonctions :

1) Il permet de renvoyer la valeur d'une fonction.
2) Il permet de quitter la fonction dans laquelle on se trouve.

```
return expression;
return 3;
```

On peut utiliser le return sans expression, juste pour quitter la fonction.

```
return;
```

Les instructions derrière un return ne sont jamais exécutées. Le return se trouve donc nécessairement soit à la fin d'une fonction, soit dans un test.

Le principal intérêt du return en programmation structurée, c'est de traiter à part les cas particuliers et du coup de diminuer le niveau d'imbrication des instructions du cas général.

On verra des exemples dans la suite du cours.

4 Le goto

Le goto permet de passer d'une instruction à n'importe quelle autre. On évite de s'en servir.

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).
>
> *Go To Statement Considered Harmful*
>
> *EdgarW. Dijkstra, 1968*

Précepte n° 9

N'utilisez jamais de goto

5 La fonction exit[13]

Cette fonction permet de quitter le programme à partir de n'importe quelle fonction.

exit est défini dans <stdlib.h>.

C'est un avatar du goto.

Il faut éviter de se servir de cette fonction, et toujours finir ses programmes proprement, c'est-à-dire dans le *main*.

Précepte n° 10

On évitera autant que possible l'usage du exit

12 K&R91 p. 25.
13 K&R91 p. 160.

CHAPITRE 4 : ENCORE DES OPERATEURS

4.1 Les opérateurs d'affectation élargie

à la place de

```
i = i + 2;
```

on peut écrire

```
i += 2;
```

Ce principe (op=) s'applique aux opérateurs arithmétiques : +, -, *, /, %
et aux opérateurs de traitement de bit.
Ces opérateurs ont une priorité très basse.
Si exp1 et exp2 sont des expressions, alors

```
exp1 op= exp2
```

équivaut à

```
exp2 = exp1 op (exp2)
```

Par exemple

```
x *= y+1;
```

équivaut à

```
x = x * (y+1);
```

On évalue d'abord l'expression à droite de l'affectation, puis on fait l'affectation.

4.2 Les expressions conditionnelles

1 Exemple : le test classique

```
if (a > b) {
    z = a;
}
else {
    z = b;
}
```

ou encore, écrit de façon anormalement concise :

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

est équivalent à

```
z = (a>b) ? a : b;
```

2 La formulation générale de l'opérateur conditionnel est la suivante :

```
exp1 ? exp2 : exp3
```

Il faut bien remarquer que l'expression conditionnelle est bien une expression, et qu'on peut l'utiliser partout où l'on peut faire figurer une expression.

3 Type du résultat, conversion

Le type du résultat est déterminé par les règles de conversions.

Par exemple :

```
(n > 0) ? f : n;
```

Si f est de type float et n de type int, l'expression ci-dessus sera de type float quelle que soit la valeur de n.

Les parenthèses ne sont pas obligatoires, mais pour une meilleure lecture on les met toujours.

4 Usage

Il ne faut pas utiliser l'opérateur conditionnel comme une instruction mais bien comme un opérateur qui renvoie un résultat. Autrement dit, il ne faut pas mettre d'affection dans l'opérateur conditionnel (ni de printf, qui est une forme d'affectation : on affecte dans la variable écran).

4.3 Les conversions automatique de type

Les seules conversions automatiques sont les conversions d'un opérateur "étroit" vers un opérateur plus "large", autrement dit les conversions qui garantissent qu'il n'y aura pas de perte d'informations. Par exemple la conversion d'un entier en un flottant dans une expression comme float + entier.

Les expressions qui n'ont pas de sens, comme se servir d'un flottant comme indice, sont interdites.

Les expressions qui peuvent conduire à une perte d'information, comme l'affectation d'un flottant en entier, peuvent générer un avertissement du compilateur (warning) mais ne sont pas illégales.

Ordre de conversion [14]:

short -> int -> long -> float -> double -> long double

(Dans le cas de signed et unsigned : je vous laisse chercher).

Le type char est convertit automatiquement en int sur un octet.

Exemple :

```
/* min : convertit c en minuscule */
int min (int c)
{
    if (c >= 'A' && c <= 'Z') {
        return c + 'a' - 'A';  /* 'a' c'est 97, 'A' 65 */
    }
    else {
        return c;
    }
}
```

14 Del97 p. 29.

4.4 Le cast

On peut forcer la conversion de type de n'importe quelle expression. C'est ce qu'on appelle "caster" une expression.

Par exemple, si n et p sont des entiers

```
int n, p;
float f;
n=10;
p=3;
f = (double) (n/p)
```

f aura comme valeur celle de n/p convertie en double, c'est-à-dire 3 car la division est entière.

Le type du cast est toujours mis entre parenthèse.

La priorité du cast est très élevée, c'est pourquoi on met toujours l'expression à "caster" entre parenthèses.

Si on écrivait :

```
(double) n / p
```

n serait convertit en double, puis l'expression serait évaluée. le premier opérande de la division étant un float, la division devient une division entre float. p est converti automatiquement en float. Le résultat serait donc 3.33333.

4.5 L'opérateur séquentiel

L'opérateur séquentiel, la virgule : " , " permet d'écrire plusieurs expression dans une même expression.

```
i++, j = i+k, j--;
```

est équivalent à

```
i++;
j = i +k;
j--;
```

Il ne faut pas abuser de cet opérateur. Il trouve parfois un intérêt dans les structures de contrôles (les boucles). Cependant, même dans ce cas, il tend à rendre l'écriture peu claire.

Précepte n° 10 bis

Evitez autant que possible l'opérateur séquentiel

4.6 Les parenthèses

On peut mettre n'importe quelle expression entre parenthèses. Les parentèses sont prioritaires sur tous les autres opérateurs.

4.7 Les 45 opérateurs

On peut *main*tenant présenter la table complète des opérateurs. Tous n'ont pas encore été abordé. Ils le seront au fur et à mesure du cours.

Les **opérateurs**[15] du C sont au nombre de 45. Ils ont une arité de 1, 2 ou 3. On dit qu'il sont unaires, binaires ou ternaires. Ils ont un niveau de priorité et une associativité (ordre de priorité en cas d'égalité).

Les opérateurs sont présentés par ordre de priorité décroissante.

Dans une même ligne du tableau, les opérateurs ont la même priorité.

OPEÉRATEURS	ASSOCIATIVITÉ	ARITÉ
() [] -> .	gauche à droite	- 2
! ~ ++ -- + - * & (*type*) sizeof	droite à gauche	1
* / %	gauche à droite	2
+ -	gauche à droite	2
<< >>	gauche à droite	2
< <= > >=	gauche à droite	2
== !=	gauche à droite	2
&	gauche à droite	2
^	gauche à droite	2
|	gauche à droite	2
&&	gauche à droite	2
||	gauche à droite	2
?:	droite à gauche	3
= += == *= /= %= &= ^= /= <<= >>=	droite à gauche	2
,	gauche à droite	2

15 K&R91 p. 53, Del97 p. 50, Bra98 p.108.

CHAPITRE 5 : LES TABLEAUX

5.1 Présentation et définition

Cf. exemple chapitre 1.5., explication n° 26.

5.2 Tableaux à une dimension

La déclaration d'un tableau s'effectue au moyen de l'opérateur [ ].

- Cet opérateur se place à la suite de l'identificateur du tableau à déclarer.
- Le nombre d'éléments du tableau est fixé par la constante entière située entre les crochets.
- 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.

Exemples:

```
main()
{
    int tab[5];
    tab[0] = 15;
    tab[1] = 3;
    tab[2] = tab [0];
…
}
```

tab :

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]
15	**3**	**15**		

Exemples :

5.3 Tableaux à plusieurs dimensions

On peut appliquer l'opérateur [ ] plusieurs fois, ce qui permet de créer des tableaux à deux dimensions ou plus.

Exemples :

```
int mat[4][3];
char Espace[3][5][2];
```

mat est un tableau de 4 * 3 = 12 éléments (par convention on dit 4 lignes est 3 colonnes). On parle de matrice pour les tableaux à deux dimensions.

espace est un tableau de 3 * 5 * 2 = 30 éléments.

mat[3][2]=5 ; on met 5 dans la 3 * **3** + 2 =11ème case du tableau, en partant de 0. (D'où l'intérêt de compter à partir de 0). La valeur **3** c'est le nombre de colonnes déclarées du tableau (int mat[4][**3**]) .

matrice :

	0	1	2
0	mat[0][0]	mat [0][1]	mat [0][2]
1	mat [1][0]	mat [1][1]	mat [1][2]
2	mat [2][0]	mat [2][1]	mat [2][2]
3	mat [3][0]	mat [3][1]	mat [3][2]

Graphiquement on représente les matrices sous la forme d'un tableau lignes, colonnes.

En réalité pour le langage C, toutes les données sont à la suite. D'où l'importance du nombre de colonnes déclarées, car c'est lui qui permet d'accéder aux éléments du tableau.

matrice :

0 ième case	1 ière case	2 ième case	3 ième case	4 ième case	5 ième case
[0][0] : 0*3+0	[0][1] : 0*3+1	[0][2] : 0*3+2	[1][0] : 1*3+0	[1][1] : 1*3+1	[1][2] : 1*3+2

6 ième case	7 ière case	8 ième case	9 ième case	10 ième case	11 ième case
[2][0] : 2*3+0	[2][1] : 2*3+1	[2][2] : 2*3+2	[3][0] : 3*3+0	[3][1] : 3*3+1	[3][2] : 3*3+2

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

```
#include <stdio.h>
#define NLMAX 5
void main (void)
{
    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]);
    }
}
```

Remarques :

- Si on ne met pas le &, on a n'importe quoi dans tab ! ! !
- On utilise une constante symbolique : NLMAX. L'intérêt est que si on décide de passer à un tableau de 100, il suffira de changer 5 en 100 une seule fois. Sinon, il aurait fallut le faire 3 fois, avec les risques d'erreurs que cela engendre.

5.5 Initialisation

1 Tableaux à une dimension :

On peut initialiser les tableaux à la déclaration.

Classique :

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

Que le début :

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

Le reste du tableau sera initialisé à n'importe quoi, c'est-à-dire par une valeur aléatoire (si locale, 0 si globale).

Autre possibilité : avec des expressions dont les opérandes sont des constantes

```
int tab [5] = {1+2, 2*3, N_MAX-1, 4}
```

Autre possibilité : taille par défaut

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

ça donne un tableau de 3 entiers.

2 Cas des tableaux à plusieurs dimensions

Soit une matrice de deux lignes et 3 colonnes, on peut l'initialiser ainsi :

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

ou encore :

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

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

tab est un tableau de 2 lignes et 3 colonnes. tab [1][2] n'a pas été initialisé.

```
int tab [] [] = { 1, 2, 3, 4, 5 }; /* erreur !!! */
```

Comme on ne connaît pas le nombre de colonnes, on ne peut pas remplir la matrice.

5.6 Typedef et tableaux

1 Présentation du typedef

Le typedef permet de donner un nouveau nom à un type.
L'ancien nom du type existe toujours.
La syntaxe est la suivante :

typedef ancien nom du type nouveau nom du type

2 Types simples

```
typedef int entier;
entier a, b;
```

3 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];
```

4 Tableaux à deux dimensions

```
typedef int typMat [10][5];
```

mat est le type d'une matrice de 10 lignes et 5 colonnes.

```
typMat mat;
```

Cette déclaration est équivalente à :

```
int mat[10][5];
```

5 Structure générale d'un programme C

```
# include
# define
typedef
prototypes
main
fonctions
```

CHAPITRE 6 : LES FONCTIONS

6.1 Présentation et définition

Cf. exemple chapitre 1.6, explication n° 30.

6.2 Prototype

Pour pouvoir être reconnues, la fonction, comme n'importe quelle variable, doit avoir été déclarée avant son utilisation. Cette déclaration c'est ce qu'on appelle le prototype. Syntaxiquement c'est l'en-tête de la fontion.

1 Exemple :

```
#include <stdio.h>
int f (int x, int y);                        / * prototype */

void main (void)
{
    int a, b ;
    printf ("entrez a et b : ");
    scanf ("%d %d", &a, &b);
    printf ("f(%d, %d) = %d \n", a, b, f(a, b) ) ;
}

int f(int x, int y) {
    return 4 * x + 2 * y;
}
```

- Le prototype reprend l'entête de la fonction.
- Le nom des paramètres formels est facultatif. Il faut seulement lister leur type.
- La déclaration du prototype se termine par un ";".

Dans notre exemple, le prototype est déclaré en dehors du *main*. Il faut prendre l'habitude de structurer un programme C ainsi.

```
# include
# define
prototypes
main
fonctions
```

2 A éviter :

On pourrait aussi déclarer le prototype dans le *main*.

On pourrait aussi ne pas déclarer de prototype et écrire la fonction avant le *main*.

Cependant, il vaut mieux toujours déclarer les prototypes avant le *main* et le *main* avnant les fonctions. Ce sera utile pour la compilation séparée.

6.3 Tableaux et fonctions[16]

1 Tableaux à une dimension :

Comment passer un tableau en paramètre d'une fonction?

Exemple : max d'un tableau.

```
int maxTab (int tab [], int taille)
{
    int i, max;
    max = tab[0];
    for (i=1; i < taille; i++) {
        if (tab[i] > max) {
            max = tab[i];
        }
    }
    return max ;
}
```

L'appel de cette fonction se fera ainsi :

```
int t[10], max ;
…
max = maxTab (t, 10);
```

16 **Début du quatrième cours.**

Autres écritures :

A la place de

```
int max (int tab [], int taille)
```

on peut aussi écrire :

```
int max (int tab [10], int taille)
```

10 étant le nombre d'élément du tableau,

ou encore

```
#define NMAX 10
int max (int tab [NMAX], int taille)
```

En réalité, dans les deux écritures précédentes, 10 ou NMAX ne servent à rien. Ce sont des commentaires.

On peut aussi écrire :

```
int max (int * tab, int taille)
```

"tab" est un "int *", c'est-à-dire un pointeur d'entier. On reviendra sur cette écriture quand on verra les pointeurs.

On choisira de préférence la première écriture.

2 Cas des tableaux à plusieurs dimensions

On traitera cela plus tard car cela fait aussi intervenir la notion de pointeur.

6.4 Les anciennes règles

Les anciennes règles (anté C ANSI) fonctionnent encore et peuvent être vues dans d'anciens programmes. Elles sont encore syntaxiquement correctes mais elles ne doivent jamais être utilisées.

1 Entête

on peut toujours écrire :

```
Surface (long, larg)
int long, larg;
{
    instructions;
}
```

Les types des variables sont déclarés après l'entête. Le type de la sortie est int par défaut. On peut donc s'en passer.

2 Prototype

```
int surface ();
```

Pas de paramètres formels.

6.5 Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.

On peut en C écrire des fonctions récursives. Cela ne pose pas de problème particulier.

Rappelons que le problème de la récursivité c'est celui de la fin de la fonction : puisqu'elle s'appelle elle-même, cela risque d'être sans fin.

Dans une fonction récursive, il y aura donc, le plus souvent au début, un test de sortie de la fonction.

Souvent, les fonctions récursives correspondent à des suites mathématiques. Il faut donc trouver la formule de cette suite.

1 Exemple :

La fonction puissance : x puissance n.

Quelle est sa défintion sous forme de suite :

x puissance 0 = 1

Pour tout n >0, x puissance N = x * x puissance N-1 ;

La traduction informatique est très simple :

```
float puis(float x, int n) {
    if (n == 0) {
        return 1;
    }
    return x * puis (x, n-1);
}
```

2 Déroulement du programme :

Soit l'instruction suivante : y = puis(4, 3) ;

Que se passe-t-il ?

On évalue puis(4, 3)

On exécute l'instruction : return 4 * puis(4, 2) ;

- du coup on évalue puis(4, 2)
- on exécute return 4 * puis(4, 1) ;
  - du coup on évalue puis(4, 1)
  - on exécute return 4 * puis(4, 0)
    - du coup on exécute return 1 ;
  - du coup on peut évaluer 4 * puis(4, 0) : cela fait 4 * 1 , soit 4
- du coup on peut évaluer 4 * puis(4, 1) : cela fait 4 * 4, soit 16

du coup on peut évaluer 4 * puis(4, 2) : cela fait 4 * 16 soit 64

y vaut 64.

Notez la façon dont on représente le déroulement du programme : en représentant les imbrications d'appel de fonctions par des indentations, cela permet d'y voir (un peu) plus clair. Si on ne fait pas ça, très rapidement on n'y comprend plus rien.

6.6 Les macros

Il existe une autre manière d'écrire des fonctions, qui ne seront plus vraiment des fonctions, ce sont les macros.

Les macros s'écrivent à l'aide de la directive #define

1 Exemple

```
#define double(a) a+a
```

double (3) donne 3+3

2 Attention :

double(3) *2 donne 3+3*2 = 9!

En effet : le paramètre formel de la fonction, ici a, est remplacé par l'écriture à droite de la fonction, comme dans un #define classique.

Il faut donc mettre des parenthèses autour de l'expression.

```
#define double(a) (a+a)
```

double(3) *2 donne (3+3)*2 = 12

3 Attention encore ! ! !

```
#define carre(a) (a*a)
```

carre (a+1) donne (a+1 * a+1) = 2a+1!

Il faut donc mettte des parenthèses autour de chaque paramètre

```
#define carre(a) ((a)*(a))
```

double (a+1) donne ((a+1)*(a+1))

4 Conclusion

Il faut être très prudent. L'intérêt des macros, c'est d'éviter les fonctions : ainsi c'est plus rapide. Donc c'est dans la majorité des cas peu intéressant.

Ca peut aussi alléger l'écriture.

CHAPITRE 7 : LES CLASSES DE VARIABLES[17]

Il y en C comme dans tous les langages de troisième génération plusieurs caractéristiques pour définir les différentes classes de variables. (On préférera la notion de classe à celle type qui est polysémique et qui, en informatique, correspond surtout aux ensembles des valeurs possibles pour une variables).

Les deux principales caractéristiques sont :

- la profondeur (ou niveau de visibilité) de la déclaration : globale ou locale.
- la durée de vie de la variable : permanente ou temporaire.
- Les notions les plus importantes sont celles de globale et locale.

7.1 Profondeur d'une déclaration : globale et locale (externe et interne)

"La profondeur d'une déclaration est déterminée par son emplacement dans le programme. Une déclaration peut être : … de profondeur 0 lorsqu'elle est située à l'extérieur de tout bloc de fonction; une déclaration de bloc ou déclaration de profondeur n (n>= 1) lorsqu'elle est située au début d'un bloc"[18].

Rappelons qu'un bloc est un ensemble d'instructions comprises entre une accolade ouvrante et une accolade fermante.

1 Les variables globales

Les variables globales (ou externes) sont les variables définies en dehors de toutes fonctions (le *main* étant une fonction comme les autres) de sorte que toutes les fonctions peuvent y accéder.

"Puisque les variables externes sont accessibles partout, elles peuvent servir à communiquer des données entre fonctions à la place des arguments de fonctions et des valeurs de retour"[19].

C'est cependant ce qu'il faut éviter autant que possible!

2 Les variables locales

Les variables locales (ou internes) sont les variables définies à l'intérieur des fonctions (Il faut bien sur les distinguer des paramètres formels de la fonction).

Il est possible de placer des déclarations au début de n'importe quel bloc d'une fonction, d'où la possibilité d'avoir des variables de profondeur n > 1 : cela correspond au niveau d'imbrication du bloc dans la fonction.

Cependant :

On ne définira que des variables locales de niveau 1, c'est-à-dire déclarées au début du bloc de la fonction.

Du point de vue terminologique, il faut utiliser préférentiellement le couple globale-locale plutôt que externe-interne qui est moins courant.

17 Del97 p. 121.
18 Bra98 p. 72.
19 K&R91 p. 73.

3 Portée des variables globales (externes)

"La portée d'une variable externe ou d'une fonction commence à l'endroit où elle est déclarée et s'étend jusqu'à la fin du fichier en cours de compilation"[20].

En général, les variables externes (globales) sont déclarées au début du fichier, après les prototypes et les types, avant le *main* et les fonctions. Mais rien n'empêche de les déclarer, par exemple après le *main*, auquel cas elles ne seront pas visibles par le *main*. Cependant il faut éviter de faire cela!

```
#include
#define
prototypes
typedef
variables globales (externes)
main () {
    variables locales (internes)
    …
}
fonction_1 (…) {
    variables locales (internes)

    …
}
```

Les variables globales sont l'occasion d'un nouveau précepte :

Précepte n° 11

Evitez les globales ! ! !

On dit aussi : évitez les effets de bord ! ! !

4 Portée des variables locales (internes)

La portée d'une variable locale commence à l'endroit où elle est déclarée et est limitée au bloc où elle est déclarée, c'est-à-dire, d'après nos règles d'usage, à la fonction"[21].

5 Règle de résolution des conflits de portée

"Une déclaration de profondeur n masque toute déclaration de profondeur inférieure qui l'englobe"[22].

Autrement dit, dans le cas général, les variables locales masquent les variables globales de même nom.

Mais comme on n'évite les variables globales, on s'en moque ! ! !

20 K&R91 p. 79.
21 K&R91 p. 79, Bra98 p. 73.
22 Bra98 p. 73.

7.2 Durée de vie de la variable

Une variable peut être permanente ou temporaire.

1 Variable permanente

Une variable permanente est une variable dont la durée de vie est celle du programme.

Les variables globales (de profondeur 0) sont toujours permanentes.

2 Variable temporaire

Une variable temporaire a une durée de vie plus petite que celle des variables permanente du programme. Elle peut naître et mourir plusieurs fois pendant l'exécution du programme.

Par défaut, les variables locales des fonctions sont des variables temporaires. Elles n'ont d'existence et de valeur que pendant l'exécution de la fonction.

Le *main* étant une fonction, les variables locales du *main* sont, en principe, temporaires. Mais comme le *main* s'arrête en même temps que le programme, en réalité, les variables locales du *main* sont en quelque sorte permanentes, bien qu'elles ne soient pas visibles par les fonctions : leur durée de vie, c'est celle du programme.

3 Initialisation

Les variables globales sont initialisées à 0 par défaut[23].

Les variables temporaires ne sont pas initialisées par défaut[24].

23 Del97 p. 122.

24 Del97 p. 122.

7.3 Variables locales permanentes : les "static"

Jusque là, on a défini les deux espèces classiques de variables : les variables globales qui sont permanentes et les variables locales qui sont temporaires.

Il existe en C une troisième espèce de variable : la variable locale permanente, qu'on appelle aussi "locale rémanente", ou "locale globale" (notez la contradiction !) ou plus couramment "statique". (Attention, "static a plusieurs sens en C" : statique s'applique aussi aux globales. Mais quand on parle de statique, on fait généralement référence à une variable locale permanente.)

La variable locale permanente, comme son nom l'indique, est locale, donc définie dans une fonction et visible uniquement dans la fonction où elle est définie. Mais elle est permanente : donc elle conserve sa valeur après la sortie de la fonction, pour le prochain retour dans la fonction.

Les variables locales permanentes sont déclarées dans la fonction où elles sont visibles, précédées du mot clé : "static". Elles peuvent être initialisées explicitement au moment de la déclaration. Par défaut, elles sont initialisées à 0.

Notons que la contradiction apparente de la définition fait déjà penser au fait que les static c'est de la bidouille ! ! !

1 Exemple[25] :

```
#include <stdio.h>
int suivant (void);
void main (void)
{
    int i ;
    do {
        i = suivant();
        printf ("suivant : %d \n", i);
    } while (i < 3);
}

int suivant (void)
{
    static int s = 1;
    return s++;
}
```

```
$ essai ↵
suivant : 1
suivant : 2
suivant : 3
$                                                    UNIX
```

25 Bra98 pp. 77-78.

Remarques :

- Un des problèmes posés par la variable locale permanente est celui de son initialisation : celle-ci n'est ne doit être faite qu'une seule fois, à l'occasion de la déclaration de la variable. Par défaut, ces variables sont, comme les globales, initialisée à 0. (La possibilité syntaxique d'initialiser une variable à l'occasion de sa déclaration trouve ici sa justification.)
- Noter que le ++ a lieu « après » le retour de la valeur : on renvoie 1, et s passe à 2.

Dans la programmation structurée en C, n'utilisez jamais les variables static ! ! !

7.4 Classe d'allocation

Avec les concepts de profondeur et de durée, on est resté à des caractéristiques de hauts niveaux qui sont communes aux langages de programmation impératifs structurés (langages de troisième génération, Pascal, C, ADA, etc.).

Il y a en C une autre notion, de plus bas niveau, mais qui offre de nombreuses possibilités de haut niveau (la compilation séparée ou la programmation objet). C'est la notion de classe d'allocation des variables.

La classe d'allocation des variables correspond à la façon dont les variables sont implémentées : on descend ici au niveau de la machine.

Une variable peut être implémentée de trois façons différentes :

- dans la zone mémoire permanente attribuée statiquement par le compilateur,
- dans la pile d'exécution (c'est une zone de mémoire qui est modifiée au fur et à mesure de l'exécution),
- dans un registre du microprocesseur.

L'implémentation d'un objet dépend de la profondeur de sa déclaration et de l'utilisation éventuelle des attributs static, register et auto.[26]

On parle donc de classe d'allocation "**statique**", "**automatique**" ou "**registre**".

classe d'allocation statique

Les variables globales et locales-permanentes sont des variables de classe d'allocation statique (il y aura des distinctions entre les variables globales).

Les variables statiques sont initialisées à 0 par défaut.

On peut ajouter l'attribut "**static**" devant une variable globale : dans ce cas, cette variable globale ne sera visible que dans le fichier où elle est déclarée et pas dans un autre fichier. Ceci concerne le développement multifichiers qu'on abordera plus tard. Mais de toutes les façons, la règle reste la même : évitez les globales ! ! !

register et auto

Les variables locales (non permanentes) sont des variables temporaires dont la classe d'allocation est :

- soit automatique (c'est le cas général),
- soit registre.

register

On ne peut appliquer le qualificatif "**register**" qu'aux variables locales de type scalaire. Celui-ci demande au compilateur d'utiliser, dans la mesure du possible, un registre pour y ranger la variable. Cela peut amener quelques gains de temps d'exécution[27].

Dans tous nos exercices, on n'utilisera jamais de variable registre. Evitez les registres tant que le problème de l'optimisation (l'accélération du programme) n'est pas clairement posé.

En général, évitez le mot clé "register" ! ! !

26 Bra98 p. 77.
27 Del97 p. 121.

auto

Par défaut, une variable locale est de classe d'allocation automatique. Cependant on peut appliquer le qualificatif "auto". Ce qualificatif n'est utilisable qu'avec les variables locales (déclarer une variable globale auto engendre une erreur). Du coup, ce mot clé, "**auto**" ne sert à rien et n'est généralement pas utilisé.

En général, évitez le mot clé "auto" ! ! !

7.5 Modifiabilité des variables : const et volatile

1 Const

On a déjà vu qu'on pouvait précéder les déclaration de variable du mot clé "**const**" qui fait que le contenu de la variable n'est alors plus modifiable.

On réabordera cela avec les pointeurs.

2 Volatile

Les variables dont la déclaration est précédée du mot clé "**volatile**" ont des propriétés particulières qui concernent l'optimisation[28] : même si elle n'est jamais modifiée dans le code, elle ne sera pas remplacée par une valeur constante. En effet, il est possible de modifier la valeur d'une variable de l'extérieur du programme, pratique qu'on évite dans la programmation structurée (on retrouvera les variables « volatile » avec les « thread » Java).

Dans la programmation structurée en C, n'utilisez jamais de variable volatile ! ! !

28 K&R91 p. 196, §A4.4, Bra98 p.82

CHAPITRE 8 : TYPES STRUCTURES

8.1 Définition d'une structure

Une structure est une variable qui rassemble plusieurs variables qui peuvent être de types différents et que l'on regroupe sous un seul nom pour les manipuler facilement[29].

Les structures s'appellent aussi des enregistrements ou des *record*.

8.2 Déclaration d'un type structure

1 Sujet :

je veux travailler sur les notes du DS de C des élèves de I' (entrer les données, les afficher, faire des moyennes, trouver le max, le min, etc.). Les élèves ont trois caractéristiques: un nom, un groupe et une note. Je vais regrouper ces trois caractéristiques dans une seule variable.

2 Solution :

```
struct eleve {
    char   nom[20] ;
    char   groupe; /* A, B ou C */
    int    note;
};
```

eleve est une structure qui contient 3 champs (notez l'utilisation du mot champ).

29 K&R91 p. 126.

8.3 Déclaration d'une variable de type structure

On utilise "struct eleve" comme un nouveau nom de type.

```
struct eleve el1, el2;
```

On peut aussi déclarer les variables en même temps qu'on déclare le type, mais c'est peu recommandé :

```
struct eleve {
    char   nom[20] ;
    char   groupe; /* A, B ou C */
    int    note;
} el1, el2;
```

8.4 Utilisation d'une structure, l'opérateur « point »

1 Initialisation

```
struct eleve e1 = {"Dupond", 'A', 12 };
```

2 Champ par champ

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 structure par l'opérateur "." suivi du nom du champ.

exemples :

```
scanf ("%s", &e1.nom)
scanf ("%c", &e1.groupe)
e1.note = 10;
```

L'opérateur "." a une priorité très élevée, supérieure à celle des autres opérateurs. C'est pourquoi on peut se passer de parenthèses.

3 Affectation globale

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.

Notons qu'on ne peut pas faire la même chose avec un tableau : ceci parce que la variable tableau est un pointeur. On comprendra cela mieux au chapitre suivant.

8.5 Simplification de l'écriture : les typedef

```
typedef ancien_type nouveau_type;
```

```
typedef struct eleve {
    char   nom[20] ;
    char   groupe; /* A, B ou C */
    int    note;
} typEleve;
```

Cette écriture renomme de type "struct eleve" en "typEleve".

Notez le principe d'écriture des variables : en minuscule, et si il y a plusieurs parties dans le nom de la variable, du type ou de la fonction, on met le début des parties intérieures en majuscule.

On peut aussi écrire de façon plus concise :

```
typedef struct { /* etc. */
```

Ainsi on pourra déclarer les variables :

```
typEleve el1, el2;
```

Retenons ici que le mieux est de toujours déclarer les structures avec un typedef. Cela allège le nom du type.

On peut aussi écrire ainsi :

```
struct eleve {
    char   nom[20] ;
    char   groupe; /* A, B ou C */
    int    note;
} ;

typedef struct eleve typEleve;
```

8.6 Imbrications de structures

1 Structure comportant des tableaux

On a défini un type élève avec une seule note. On souhaite *main*tenant que les élèves puissent avoir jusqu'à 10 notes.

On ajoute un tableau de 10 notes dans la structure.

2 Solution :

```
typedef struct eleve {
    char   nom[20]
    int groupe;
    int note[10];
} eleve;
```

A noter qu'on avait déjà un tableau de caractères pour gérer la chaîne de caractères.

on peut alors écrire :

```
e1.note[5]=4;
scanf ("%d", & e1.note[5])
```

enfin on pourra initialiser ainsi :

```
eleve e3 = {
    "Dupond", 'A',
    {12, 15, 10, 8, 12, 5, -1, -1, -1, -1}
};
```

3 Tableau de structure

Maintenant on veut travailler sur toute la classe, donc sur un ensemble de variable élève: un tableau d'élèves.

Déclaration et utilisation d'un tableau de 200 élèves :

4 Solution :

```
TypEleve tableleve[200];
tableleve[0].nom = "Dupond";
scanf ("%d", & tableleve[0].note[5]);
```

5 Structure de structure

Chaque note d'élève est caractérisée par la matière et la date en plus de la valeur.

6 Solution :

```
typedef struct infoNote {
    int note;
    matiere char[20];
    char date[9];
}typInfoNote;
typedef struct eleve {
    char          nom[20]
    int           groupe;
    typInfoNote   infoNote[10];
} typEleve;
```

Notez qu'il faut déclarer typInfoNote avant le typEleve car typEleve utilise typInfoNote.
On utilisera cette structure ainsi :

```
typEleve e1 ;
e1.infoNote[2].note=15;
```