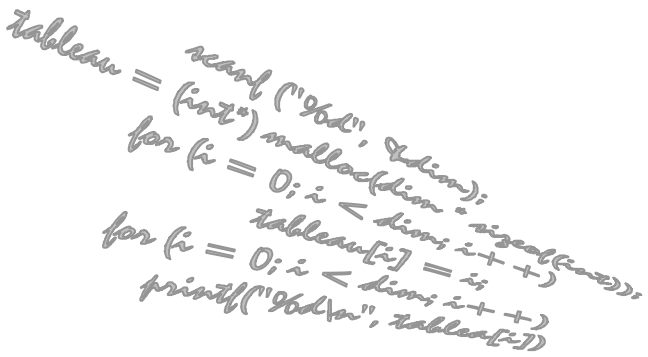
Faculté Polytechnique de Mons

Service d'Informatique

Introduction au Langage de Programmation C

Mohammed Benjelloun

1ère Candidature



Année académique 2003-2004

Avant-propos

Ces notes permettent de se familiariser avec le langage de programmation C. Elles constituent le support des travaux pratiques et des exercices dans le cadre de l'enseignement du cours d'Introduction à l'Informatique de 1e Candidature à la Faculté Polytechnique de Mons. Ce support a pour objectif d'aider l'étudiant dans l'assimilation des principales notions et de certains concepts d'algorithmique vu au cours théorique. Afin d'illustrer par la programmation ces concepts, ce syllabus est agrémenté de nombreux exemples (simples et courts) permettant au débutant de mettre immédiatement en application ce qu'il vient de lire.

Ces notes ne sont donc nullement un manuel de référence du langage C, mais simplement une aide à l'apprentissage du langage. Certaines constructions syntaxiques sont volontairement omises, par souci de clarté. D'autres concepts sont volontairement absents ou ne sont vus que superficiellement, afin de ne pas accabler le lecteur néophyte en programmation. Il ne faut donc pas considérer ce syllabus comme auto-suffisant, mais bien, pour ce qu'il doit être, c'est-à-dire une introduction à la programmation en C et un support d'exercices et de travaux.

Néanmoins, nous espérons qu'il vous permettra de mieux comprendre ce langage et qu'il vous en rendra l'apprentissage plus agréable. Bien entendu, nous ne pouvons que chaudement renvoyer le lecteur désireux d'en savoir plus à l'un des très nombreux ouvrages sur le C disponibles dans toutes les bonnes librairies, et notamment à:

Brian W. Kernighan, Dennis M. Ritchie, ***Le langage C, norme ANSI***, Masson, 1997 (Traduction de l'anglais, 3e édition).

Byron S. Gottfried, ***Programmation en C***, McGrawHill, 1997 (Traduction de l'anglais).

Leendert AMMERAAL, ***Algorithmes et structures de données en langage C, C ANSI et C++***, traduit de l'anglais par Chantal SAINT-CAST , Paris : InterEditions, 1996.

Gerhard Willms, ***Le grand livre de la programmation en langage C***, traduit de l'anglais par Georges-Louis Kocher, Paris : Micro Application, 1995.

Claude DELANNOY, ***Le livre du C:premier langage pour les débutants en programmation***, Paris : Eyrolles, 1994

Je voudrais remercier ici les collègues du Service d'Informatique, et notamment le Professeur Gaëtan Libert, qui ont pris soin de relire ces notes et de suggérer corrections et améliorations. Un merci tout particulier à Pierre Manneback pour, en plus de la lecture de ce support, avoir préparé le terrain du cours de programmation en C, étant donné qu'il l'avait enseigné avant moi. Je suis bien conscient qu'il reste des erreurs et/ou des imprécisions. Merci au lecteur assidu de les signaler!
Si vous notez la moindre erreur ou si vous souhaitez me proposer vos suggestions, n'hésitez pas à le faire. Vous trouverez mon adresse e-mail en bas de cette page .

Mohammed.Benjelloun@fpms.ac.be

Chapitre 1 : Introduction

1.1. Historique

Le langage de programmation C a déjà une longue histoire derrière lui dans le monde informatique. C'est un des plus puissants langages évolués.
Il trouve ses sources en 1972 dans les 'Bell Laboratories'. Il fut inventé par les deux ingénieurs Dennis Ritchie et Brian Kernighan pour concevoir un système d'exploitation portable, UNIX dont plus de 90% du noyau est écrit en C. Ce langage de programmation structuré, d'assez bas niveau, est très proche de la structure matérielle des ordinateurs (mots, adresses, registres, octets, bits, ...). En fait c'est un compromis entre un langage de haut niveau (Pascal, Ada ...) et un langage de bas niveau (assembleur). Son succès est dû au succès de UNIX et à la disponibilité de compilateurs efficaces et bons marchés (parfois même gratuits).
Le succès des années ultérieures et le développement de plusieurs compilateurs C ont rendu nécessaire la définition complète et stable d'un standard actualisé et plus précis. En 1983, le *'American National Standards Institute'* (*ANSI*) chargeait une commission de mettre au point une définition explicite et indépendante de la machine pour le langage C, qui devrait bien sûr conserver l'esprit du langage. Le résultat était le *standard ANSI-C*. C'est ainsi que la seconde édition du livre *'The C Programming Language'*, parue en 1988, respectant le standard *ANSI-C* est devenue la "référence" pour tout utilisateur de ce langage.

1.2. Intérêts du langage

Nous allons tout au long de ce support étudier le langage *ANSI-C* qui a su devenir le langage *universel* en programmation. En effet, afin de s'assurer de la portabilité d'un programme en C sur d'autres compilateurs, nous conseillons fortement de respecter les spécifications *ANSI-C* et de tenir compte des avertissements (warnings) à la compilation. Nous allons voir que sous ce terme se cachent: un langage qui possède assez peu d'instructions, mais qui fait appel à un ensemble de librairies standards, fournies avec le compilateur, permettant de contrôler tous les aspects envisageables tels les accès aux périphériques, la programmation système, ...

Le langage permet de développer des applications à la fois complexes, efficaces et rapides grâce aux structures de contrôles de haut niveau. Il permet aussi d'écrire un code extrêmement concis, alors que dans un autre langage la taille du programme peut être bien plus grande. Cependant cet atout peut se révéler être un inconvénient car parfois on perd en lisibilité.

Comme déjà évoqué, la popularité du langage C est due principalement à ses caractéristiques dont nous donnons un résumé. C est un langage:

Structuré : conçu pour traiter les tâches d'un programme en les mettant dans des blocs.
Simple : formé d'un nombre limité mais efficace d'opérateurs et de fonctions.
Efficace : possédant les mêmes possibilités de contrôle de la machine que l'assembleur, donc générant un code compact et rapide.
Portable : en respectant la norme *ANSI-C*.

Modulaire : permettant de séparer une application en plusieurs sous-modules s'occupant chacun d'un sous-domaine du problème initial. Ces modules pourront être compilés séparément. Un ensemble de programmes déjà opérationnels pourra être réuni dans une librairie.

Extensible : ne se composant pas seulement des fonctions standard; le langage est animé par des librairies de fonctions privées ou livrées par d'autres développeurs.

Ce syllabus est découpé en neuf chapitres, chacun d'entre eux nous permettant de nous focaliser sur chacun des points du langage (les types, les expressions, les instructions, les fonctions, manipulation de pointeurs ...). Aucune connaissance d'un langage de programmation n'est supposée, mais bien des notions d'algorithmique vue au cours.

Les notes, du chapitre 2 au chapitre 8, sont accompagnées de nombreux programmes (ou parties de programme) d'illustration. Nous appelons un programme un ensemble d'instructions contenant tout ce qu'il faut afin d'être compilé et exécuté. Ces programmes sont représentés comme suit :

Programme Chap.Num

Nous recommandons à nos lecteurs de suivre les styles de programmation suggérés.

Ces mêmes chapitres se terminent par une rubrique '' *Ce qu'il faut au moins retenir* '' permettant de donner un résumé du chapitre et mettre parfois l'accent sur des erreurs ou des pièges dans lesquels tombent souvent les étudiants. Cette rubrique se présente comme suit :

Quelques exercices suivent chaque chapitre. Nous invitons le lecteur à les résoudre afin de maîtriser le langage. Les solutions ne se trouvent pas dans ce syllabus : certaines seront proposées en séances de travaux et/ou d'exercices ; les étudiants de la faculté désireux d'en savoir plus sur les autres peuvent me contacter.

Le dernier chapitre décrit quelques erreurs observées lors des séances de travaux ou à l'examen. La plupart sont des erreurs d'inattention ou proviennent de la syntaxe du langage combinée à l'inattention du programmeur. Le but de ce chapitre est d'attirer l'attention des étudiants sur ce genre d'erreurs afin de les éviter.

Enfin, on trouvera en annexe A la table de précédence des opérateurs qui fixe l'ordre de priorité ainsi que la table des codes ASCII des caractères. En annexe B les fonctions de la librairie standard et enfin en annexe C les transparents des séances de travaux.

Chapitre 2 : Eléments du langage C

2.1. Structure d'un programme C

Un programme écrit en C doit obéir à certaines règles bien définies. Il est composé des *directives du préprocesseur* (une ligne de programme commençant par le caractère dièse (#)) et d'une ou plusieurs *fonctions* stockées dans un ou plusieurs fichiers. Une fonction particulière, dont la présence dans le programme est obligatoire, est la fonction programme principal ou *main*. Le programme commencera toujours son exécution au début de cette fonction.

Une fonction est un bloc de code d'une ou plusieurs instructions qui peut renvoyer une valeur à la fonction appelante. La forme générale *ANSI-C* d'une fonction est :

Classe [Type] Ident([liste_de_parametres_formels])
Corps_de_la_fonction

Les éléments entre [] dans cette syntaxe sont facultatifs, car une valeur par défaut existe. Nous verrons plus tard au chapitre 5 de nombreux exemples de fonctions.

La syntaxe précédente appelé déclaration de la fonction contient :

Classe : la classe de mémorisation de la fonction qui peut prendre par exemple la valeur static ou extern. Classe peut ne pas exister.

Type : le type de la valeur renvoyée par la fonction. Type étant un des types reconnus par le langage C comme :void, int, float… (voir 2.6).

Ident : l'identificateur représentant le nom par lequel on désigne la fonction (voir 2.3).

liste_de_parametres_formels :la liste de paramètres nécessaires pour la bonne exécution de la fonction. Chacun des paramètres ayant un nom et un type.

Corps_de_la_fonction : est un corps commençant par une accolade ouvrante ''**{**'' et terminée par une accolade fermante ''**}**'',comportant des définitions et des déclarations de variables, des instructions devant être exécutées par la fonction, ainsi que d'éventuelles déclarations d'autres fonctions devant être appelées par la fonction elle-même.

La figure 2.1 nous présente une structure simple d'un programme C.

Nous verrons tout au long de ce syllabus plus en détail et d'une manière plus précise le rôle de chaque bloc. Remarquez cependant les parenthèses obligatoires après les noms des fonctions et les accolades d'ouverture et de fermeture délimitant le début et la fin de chaque fonction. La fonction *fonc1*() retourne une valeur à la fonction appelante *main()*. Les fonctions *fonc2()* et *main()* ne retournent rien. En effet le mot-clé ***void*** (vide en anglais) devant ces fonctions, indique qu'elles ne retournent aucune valeur. Le (*void*) après *main*() indique que cette fonction ne prend aucun argument.

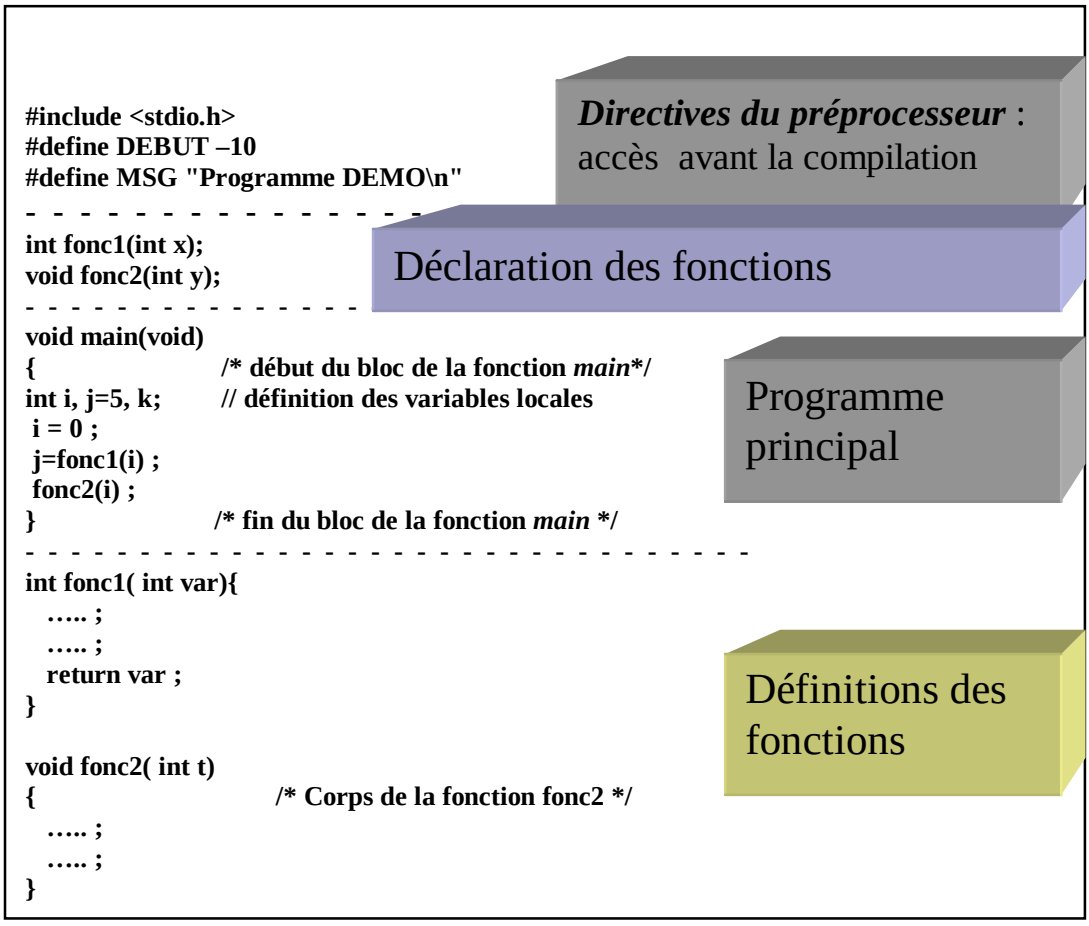


Figure 2.1 : Une structure simple d'un programme C

2.2. Commentaires en C

Dans un programme C, il est possible (et d'ailleurs préférable) d'ajouter des commentaires facilitant ainsi la lecture et la compréhension du programme. En effet, comme nous avons la possibilité d'utiliser des expressions compactes, ceci peut parfois rendre le programme moins lisible et sans commentaires ou explications, ces programmes peuvent devenir incompréhensibles.

Toute suite de caractères encadrée par les symboles '**/***' et '***/**' correspond à un commentaire, et ne joue évidemment aucun rôle lors de l'exécution du programme. Ces commentaires sont donc ignorés par le compilateur et ne sont indispensables que sur le plan de la documentation du programme. Notons que les commentaires imbriqués sont interdits. Il faut noter aussi que certain compilateurs acceptent le symbole '//' pour un commentaire sur une seule ligne.

Exemple 1

syntaxiquement corrects :

```
/* ce programme est un commentaire
   écrit en deux lignes */
// un commentaire sur une seule ligne
/*  un autre commentaire sur une seule ligne  */
```

syntaxiquement incorrects :

```
/* imbriquer /* ce n'est pas autorisé */ donc à éviter */
```

2.3. Identificateurs

Un *identificateur* est un nom donné à un objet du programme (variable, constante, fonction), composé de lettres et de chiffres commençant par une lettre. Le caractère _ (souligné) est considéré comme une lettre.

- En C, les lettres minuscules sont différenciées des majuscules ainsi TAILLE, Taille et taille sont trois identificateurs différents et peuvent être utilisés dans le même programme.
- Les mots réservés par le compilateur utilisés, dans la syntaxe du C, ne peuvent être utilisés comme identificateurs à cause des confusions que cela pourrait entraîner.
- La norme ANSI-C a fixé à 31 le nombre de caractères significatifs d'un identificateur, bien que la longueur de ce dernier puisse être plus importante. Mais attention, dans ce cas, il peut y avoir confusion entre identificateurs lors de l'exécution d'un programme!

Exemple 2

Identificateurs valides:

Xx, y1, fonc1, _position, DEBUT, fin_de_fichier, MSG, VeCteuR

Identificateurs invalides :

3eme	commence par un chiffre
x#y	caractère non autorisé (#)
no-commande	caractère non autorisé
taux de change	caractère non autorisé (espace)
main	mot réservé du C

2.4. Directives du préprocesseur

Les directives du préprocesseur permettent d'effectuer un prétraitement du programme source avant qu'il soit compilé. On y retrouve ce dont le compilateur a besoin pour compiler correctement un programme. Ces directives sont identifiées par le caractère dièse (#) comme premier caractère de la ligne. On les utilise pour l'inclusion de fichiers, macros, compilation conditionnelle, …

2.4.1. Inclusion de fichiers

La directive **#include** insère, au moment de la compilation, le contenu d'un fichier, généralement réservé à l'inclusion de fichiers appelés *fichiers en-tête* contenant des déclarations de fonctions précompilées, de définition de types... Ces fichiers sont traditionnellement suffixés par **.h (**header file).

Syntaxe

```
#include <nom-de-fichier>      /* répertoire standard */
#include "nom-de-fichier"      /* répertoire courant */
```

Ainsi l'instruction suivante :

```
#include <stdio.h>          /* Entrees-sorties standard */
```

invoque la librairie des fonctions d'entrée-sortie (saisie au clavier *scanf*() et affichage à l'écran *printf*()).

Le langage C fait appel à un ensemble de librairies standards, fournies avec le compilateur, contenant des fonctions prédéfinies permettant de contrôler tous les aspects envisageables tels que le traitement de chaînes de caractères <**string.h**>, le traitement et la classification de caractères <**ctype.h**>, les utilitaires généraux(gestion de mémoire, conversions, interruption programme, …) <**stdlib.h**>, la gestion de la date et de l'heure <**time.h**>, la manipulation des fonctions mathématiques <**math.h**>, etc.

2.4.2. Pseudo-constantes et Pseudo-fonctions

La directive **#define** permet la définition de *pseudo-constantes* et de *pseudo-fonctions*.

Syntaxe

#define identificateur [chaîne-de-substitution]

La directive *#define* permet de déclarer des constantes ou des macros sans que le compilateur ne réserve de place en mémoire. Le préprocesseur remplace tous les mots du fichier source identiques à l'identificateur par la *chaîne-de-substitution*, sans aucune vérification syntaxique plus poussée.

```
#define     DEBUT   –10
#define     MSG     "Programme DEMO\n"          /* voir Figure 2.1 */
#define     PI      3.14159
#define     TAILLE      100
#define     MAXI    (3 * TAILLE + 5)
#define     Begin   {
#define     End     }
#define     Nom_macro(identif_p1 , ... ) texte
#define     SOMME(X,Y)  X+Y
#define     MULTIP(A,B)  (A)*(B)
```

Il faut rappeler que les constantes sont des données dont la valeur ne peut pas être modifiée durant l'exécution du programme.

Il existe une autre manière de définir des constantes en C par l'instruction ***const***. Dans ce cas les constantes PI et TAILLE seront définies comme suit :

```
const  DEBUT =  –10 ;          // ou const  int DEBUT =  –10 ;
const TAILLE = 100;
const float PI = 3.141596;
```

Nous attirons l'attention sur la manière de définir et d'affecter les constantes avec *#define* et avec *const*. En effet, si avec *const* on utilise le symbole d'affectation (=) et le point virgule, l'utilisation de *#define* n'en a pas besoin.

Pour définir les dimensions des tableaux, nous utiliserons l'instruction *#define*.

2.5. Points-virgules

En C, le point virgule est un *terminateur* d'instruction, on en met donc après chaque instruction. On met également un point virgule après chaque déclaration.

Les habitués du langage Pascal seront surpris d'en trouver à des endroits où ils n'ont pas l'habitude d'en voir, comme avant un *else* par exemple. En effet, en Pascal, on met un point virgule seulement entre deux instructions, comme séparateur et non terminateur.

2.6. Type de données

Lors de la programmation, on a besoin de manipuler et de stocker les données. Le langage C est déclaratif et typé, les variables et les fonctions utilisées doivent être déclarées et leur type est vérifié à la compilation. Il permet la représentation des données sous différents formats. Les différents types de données de base à partir desquels on peut construire tous les autres types, dits types dérivés (tableaux, structures, unions ...) sont les suivants:

Type	Représentation	Signification	Intervalle	Taille en octes	Exemple
caractères	signed char ou char unsigned char	Un caractère unique	-128 à 127 0 à 255	1	'a' 'A' 'Z' '\n'
entiers	signed int ou int unsigned int short ou long	entier signé (par défaut) entier positif spécifie la taille de l'emplacement mémoire utilisé	-2^{31} à $2^{31}-1$ 0 à 2^{32} -2^{15} à $2^{15}-1$	2 ou 4	0 1 -1 4589 32000
réels	float double long double	Réel simple précision Réel double précision Réel précision étendue	+-10 E-37 à +-10 E+38 +-10 E-307 à +-10 E+308	4 8	0.0 1.0E–10 1.0 - 1.34567896

Le codage en mémoire va déterminer les valeurs limites (minimales, maximales, précision numérique) pour les nombres entiers et réels. On peut définir des types non signés (unsigned) qui correspondent à des valeurs toujours positives. Les caractères sont eux représentés en machine par un entier variant de –128 à 127, correspondant pour les valeurs positives au code ASCII (code standard associant un ensemble de caractères standards à des entiers, voir Annexe A). A noter que les caractères accentués ou spéciaux du français (é, à, ç, ô, …) ne font pas partie du code ASCII. Nous les éviterons donc.

Il existe d'autres caractères spéciaux représentant une séquence d'échappement qui seront toujours précédés de l'anti-oblique (*backslash*) \ et sont interprétés comme suit :

Caractères	Signification	CODE ASCII (hexadécimal)
\n	Génère une nouvelle ligne (newline)	0x0A
\t	Tabulation horizontale	0x09
\v	Tabulation verticale	0x0B
\b	Retour d'un caractère en arrière (backspace)	0x08
\r	Retour chariot (return)	0x0D
\f	Saut de page (form feed)	0x0C
\a	Déclenche un signal sonore (alarm)	0x07
\\	Affiche une barre oblique inverse (backslash)	0x5C
\'	Affiche une apostrophe (single quote)	0x2C
\"	Affiche un guillemet (double quote)	0x22

2.7. Les variables

Les variables, qui doivent être déclarées avant leur utilisation, sont des données dont la valeur peut être modifiée durant l'exécution du programme. Chaque variable possède un type de données, une adresse mémoire et un nom qui est son identificateur. Pendant l'exécution du programme, elles occupent de la place mémoire.

Définir une variable, c'est donc associer un type et un identificateur**.** La syntaxe de cette opération est :

Type Ident [= valeur] {, Ident [= valeur]};

Dans cette notation, les éléments entre crochets [ ] sont optionnels, les éléments entre accolades {} peuvent apparaître 0, 1 ou plusieurs fois.
Type est un des types reconnus par le langage C.
Ident est l'identificateur représentant le nom par lequel on désigne la variable.
valeur est la valeur initiale qu'on souhaite attribuer à la variable.
Si, dans la déclaration, la variable n'est pas initialisée (on ne lui attribue pas de valeur initiale), cette dernière peut alors contenir n'importe quoi.

Exemple 3

```
int i, j=5, k;          // i, j, k sont des variables de type entier et j est initialisée à 5 (voir Figure 2.1)
float moyenne;          // moyenne est une variable de type virgule flottante,
char une_lettre;        // une_lettre est une variable de type caractère
int   Compteur = 0;
char   TAB = '\t';
float   X = 2.03e-3;
```

Le compilateur réserve la mémoire nécessaire pour stocker ces variables. Leur initialisation n'est pas nécessaire lors de la déclaration, mais il est possible de les initialiser par affectation.

En utilisant l'attribut **const**, nous pouvons indiquer que la valeur d'une variable ne change pas au cours d'un programme:

```
const int  MAX = 1000;
```

```
const double  Precision = 0.00000000001;
const char  NEWLINE = '\n';
```

2.8. Mots réservés

L'attribution des noms à des identificateurs est soumise à une limitation due au groupe des mots dits réservés ou mots-clés. Les mots réservés du langage C sont des noms prédéfinis qui ont une signification particulière, et ne peuvent pas servir de noms de variables personnelles.
La signification des mots réservés les plus utilisés sera donnée au fur et à mesure du cours.
Nous ne verrons cependant pas ici la signification de tous ces mots réservés.

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

2.9. Opérateurs et expressions

2.9.1. Opérateurs arithmétiques

Les opérateurs arithmétiques binaires sont

`+`	`addition`
`-`	`soustraction`
`*`	`multiplication`
`/`	`division (entière et rationnelle!)`
`%`	`modulo (reste d'une div. entière)`

7%2 vaut 1, alors que 7/2 vaut 3. Si l'on désire obtenir une division réelle, il faut que le numérateur et/ou le dénominateur soi(en)t réel(s). Ainsi 7.0/2, 7/2.0 et 7.0/2.0 donnent la même valeur, soit 3.5.

Le compilateur considère le type des opérandes pour savoir comment effectuer les opérations. Si, par exemple, on a :

```
int       i = 5, j = 4, k;
double    f = 5.0, g = 4.0, h;

k = i / j;        // k = 5/4 = 1
h = f / g;        // h = 5.0/4.0 = 1.25
h = i / j;        // h = 5/4 = 1.0000
```

Les opérateurs + et – sont également utilisés comme opérateurs de signes unaires (opérateurs placés devant un entier ou un réel pour indiquer son signe).

La priorité dans les évaluations d'expression est donnée aux opérateurs unaires + et -, puis aux opérateurs *, / et %, et enfin aux opérateurs binaires + et – (voir Annexe A). Par exemple,

5 + 3 * -7 = 5 + (3*(-7)) = -16
5 – 2 / -1 + 8 = 5 – (2/(-1)) + 8 = 15

On peut toujours utiliser des parenthèses en cas de doute de priorité, ce qui peut d'ailleurs faciliter la lisibilité.

(5 + 3) * (-7) = -56
(5 + 10) / (-1 + 8) = 15/7 = 2 (en arithmétique entière)

Les expressions sont formées de combinaisons d'opérandes (constantes caractères ou numériques, identificateurs de variables ou de fonctions) et d'opérateurs, avec éventuellement des parenthèses.

Ainsi,

(3*x*x + 5*x*y + 7)/(x + 1)
2**cos*(PI*x)
car + 3

où x, y sont des variables réelles, PI une constante réelle, *car* une variable caractère, *cos*() une fonction cosinus, sont des expressions correctes. La troisième expression fournit le 3^{e} caractère, dans l'ordre ASCII (voir Annexe A), après le caractère contenu dans la variable *car* (par exemple 'e' si *car* vaut 'b').

2.9.2. Conversions de type forcées (casting)

Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation à l'aide de la syntaxe:

(Type) <Expression>

Exemple 4

Nous divisons deux variables de type entier. Pour avoir plus de précision, nous voulons avoir un résultat de type rationnel. Pour ce faire, nous convertissons l'une des deux opérandes en float. Automatiquement le compilateur C convertira l'autre opérande en float et effectuera une division rationnelle:

```
int I=3, J=4;
float K;
K = (float)I/J;
```

La valeur de I est explicitement convertie en float. La valeur de J est automatiquement convertie en float. Le résultat de la division (type rationnel, valeur 0.75) est affecté à K.

Attention !

Les contenus de I et de J restent inchangés; seulement les valeurs utilisées dans les calculs sont converties !

Exemple 5

```
int i=0x1234, j;
char d,e;
float r=89.67,s;
j = (int)r;                 // r= 89
s = (float)i;               // s= 4660.00 = 4x16^0 + 3x16^1 + 2x16^2 + 1x16^3
d = (char)i;                // d= 34
e = (char)r;                // e= 89
```

2.9.3. Opérateurs logiques et opérateurs de comparaison

Les opérateurs logiques permettent de calculer des expressions logiques, au résultat vrai (différent de 0), ou faux (égal à 0) .

Opérateurs logiques

&&	et logique (and)
||	ou logique (or)
!	négation logique (not)

Opérateurs de comparaison

==	égal à
!=	différent de
<, <=, >, >=	plus petit que, ...

Les résultats de ces opérateurs sont du type entier. Le type booléen n'existe pas. Le résultat d'une expression logique vaut 1 si elle est vraie et 0 sinon. Réciproquement, toute valeur non nulle est considérée comme vraie et la valeur nulle comme fausse.

- la valeur 1 correspond à la valeur booléenne vrai
- la valeur 0 correspond à la valeur booléenne faux

Les opérateurs logiques considèrent toutes valeurs différentes de zéro comme valeur vrai et égales à zéro comme valeur faux.

Exemple 6

```
x == y                          // vrai si x égal à y (!0), faux sinon (0)
rayon > 0                       // vrai si rayon>0 (!0), faux sinon (0)
(rayon > 0) && (rayon < 10)     // vrai si rayon strictement entre 0 et 10
(c <= 'b') || ( c > 'x')        // vrai si le caractère c vaut 'a', 'b', 'y' ou 'z'
int   i;
float f;
char  c;
  i = 7;     f = 5.5;          c = 'w';
```

```
f > 5                    ====>  vrai (1)      //retourne la valeur 1(vrai)
(i + f) <= 1             ====>  faux (0)
c == 'w‘                 ====>  vrai (1)
c != 'w'                 ====>  faux (0)
c >= 10*(i + f)          ====>  faux (0)
(i >= 6) && (c == 'w')   ====>  vrai (1)
(i >= 6) || (c == 119)   ====>  vrai (1)      // code ASCII de 'w' est 119
```

Pour les expressions logiques formées de **&&** ou de ||, le programme évalue tout d'abord l'expression de gauche, et ensuite celle de droite, mais seulement si le résultat n'est pas encore déductible. C'est ce qu'on appelle une *évaluation paresseuse*.

!expr1 est vrai si expr1 est faux et faux si expr1 est vrai ;
expr1&&expr2 est vrai si les deux expressions expr1 et expr2 sont vraies et faux sinon. L'expression expr2 n'est évaluée que dans le cas où l'expression expr1 est vraie ;
expr1 || expr2 = (1) si expr1=(1) ou expr2=(1) et faux sinon. L'expression expr2 n'est évaluée que dans le cas où l'expression expr1 est fausse.

2.9.4. Opérateurs d'affectation

Le symbole d'affectation = est un opérateur à part entière qui peut être utilisé au sein d'expressions. Son rôle consiste à évaluer l'expression du membre de droite et à transférer ce résultat comme valeur de l'objet au membre de gauche.

L'instruction d'affectation la plus simple est donc : variable = expression ;
Une autre instruction tout aussi correcte est

variable1 = variable2 = expression ;

En pratique, nous retrouvons souvent des affectations comme:

```
i = i + 5 ;   /* On prend la valeur contenue dans la variable i, on lui ajoute 5 et on met le résultat dans
                 la variable i.  */
```

En C, il est possible d'utiliser une formulation plus compacte:

```
i += 5 ;      /* réalise la même opération que l'instruction précédente */
```

L'opérateur **+=** est un *opérateur d'affectation.*

Pour la plupart des expressions de la forme:

expr1 = (expr1) op (expr2)

il existe une formulation équivalente qui utilise un opérateur d'affectation:

expr1 op= expr2

Opérateurs d'affectation

+=	ajouter à
-=	diminuer de
***=**	multiplier par
/=	diviser par
%=	modulo

Ainsi ces expressions sont équivalentes :

```
X= X + 32 ;          X += 32 ;
F= F/1.5 ;           F /=  1.5 ;
I= I *(J+10);        I *= J+10;
```

Tab[n*i+j]= Tab[n*i+j]*3; Tab[n*i+j] *= 3;

Le C fournit aussi d'autres opérateurs contractés pour des opérations usuelles. Ainsi l'opérateur d'incrémentation ++ ajoute 1 à son opérande entière. De même l'opérateur de décrémentation -- retranche 1 à son opérande. On peut les utiliser soit devant une variable (*préfixé* : mise à jour de la variable *avant* son utilisation dans l'expression courante), soit derrière la variable (*postfixé* : mise à jour *après* son utilisation dans l'expression courante).

Exemple 7

```
int i=4, j=3, k;
    i++;        // équivalent à i= i+1, donc i vaut maintenant 5
    --j;        // équivalent à j= j-1, donc j vaut maintenant 2
    k = ++i;    // i = i+1 puis  k = i,  donc i=k= 6
    k = j--;    // k = j puis j=j-1, donc k vaut 2 et j vaut 1
```

Ce dernier exemple nous incite à faire attention lors de l'utilisation des opérateurs contractés. Il faut savoir quand utiliser une variable préfixée ou postfixée. La confusion entre les deux peut induire des erreurs dans le programme. Il est à noter que les formulations compactes ne sont pas obligatoires et n'ont pour rôle que de réduire la quantité de code du programme.
Voici une explication sur l'utilisation de la forme compacte pour incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable :

postfixe	`X = I++`	passe d'abord la valeur de I à X et ***incrémente I après***
	`X = I--`	passe d'abord la valeur de I à X et ***décrémente I après***
préfixe	`X = ++I`	***incrémente d'abord I*** et passe la valeur incrémentée à X
	`X = --I`	***décrémente d'abord I*** et passe la valeur décrémentée à X

2.9.5. Classes de priorités

Parmi les opérateurs que nous connaissons jusqu'ici, nous pouvons distinguer les classes de priorités suivantes:

Priorité 1 (la plus forte):	`( )`
Priorité 2:	`! ++ --`
Priorité 3:	`* / %`
Priorité 4:	`+ -`
Priorité 5:	`< <= > >=`
Priorité 6:	`== !=`
Priorité 7:	`&&`
Priorité 8:	`||`
Priorité 9 (la plus faible):	`= += -= *= /= %=`

Evaluation d'opérateurs de la même classe :

→ Dans chaque classe de priorité, les opérateurs ont la même priorité. Si nous avons une suite d'opérateurs binaires de la même classe, l'évaluation se fait en passant *de la gauche vers la droite* dans l'expression.

← Pour les opérateurs unaires (`!,++,--`) et pour les opérateurs d'affectation (`=,+=,-=,*=,/=,%=`), l'évaluation se fait *de droite à gauche* dans l'expression.

2.10. Nos premiers programmes

2.10.1. Le programme C le plus court

Notre premier programme est le programme C le plus court à écrire. Et pour cause, c'est un programme qui ne fait rien. Il est constitué uniquement de la fonction ***main,*** obligatoire pour que le programme soit exécutable.

```
void main(void)
{
}
```

Cette fonction ne retourne aucune valeur (premier void) et ne prend aucun argument (deuxième void). Les parenthèses après *main* sont obligatoires, elles spécifient que l'on est en train de décrire une fonction.
Les accolades **{}** encadrent un bloc d'instructions (ensemble de déclarations et instructions exécutables), constituant le **Corps_de_la_fonction**. Dans notre exemple, ce corps est vide, donc le programme ne fera rien.

2.10.2. Le premier programme

A titre d'exemple, notre premier programme devient le programme 2.1 qui calcule la circonférence et la surface d'un disque.

```
void main(void)
        /* Ce programme calcule la circonférence et
                        la surface d'un disque de rayon 2 */
{
 const float PI = 3.141596;                  // initialisation de la constante PI
 int rayon = 2;                              // initialisation de la variable rayon
 float circonference, surface, temp;         // déclaration des variables réelles

 temp = PI*rayon;                            // affectation d'une variable temporaire
 circonference = 2*temp;                     // calcul de la circonférence
 surface = temp*rayon ;                      // calcul de la surface
}
```

Programme 2.1

L'exécution de ce programme n'affichera aucun résultat à l'utilisateur. En effet, aucune fonction d'affichage n'a été invoquée !

2.11. Fonctions de saisie et d'affichage

En C, les fonctions de saisie/affichage ou d'entrée/sortie (I/O) sont gérées par des librairies, c'est-à-dire des ensembles de fonctions standards pré-programmées, livrées avec les compilateurs.

2.11.1. La fonction d'affichage printf()

Une fonction qui permet l'affichage formaté des messages à l'écran est la fonction ***printf()***. Formaté signifie que nous pouvons contrôler la forme et le format des données affichées. ***printf()*** prend un nombre variable de paramètres ou arguments où le premier argument est une chaîne de caractères décrivant le format d'affichage.

Syntaxe : int ***printf***(const char *format [, arg [, arg]...]);
Description : Permet l'écriture formatée (à l'écran par défaut).

Pour utiliser cette fonction dans un programme, il faut inclure la librairie standard I/O <**stdio.h**>. Ainsi, Le programme 2.2 ne fait rien d'autre qu'afficher les mots suivants à l'écran: " Salut a tous "

```
#include <stdio.h>                    /* pour standard I/O */
void main(void)
{
        printf(" Salut a tous ");
}
```

Programme 2.2

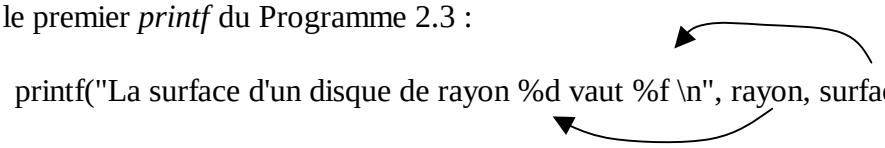
Remarquez que nous évitons d'utiliser des caractères spéciaux (à,é,à,è,ù,ô,…) dans les affichages. Leur traitement dépend en effet des compilateurs et sort du cadre de la norme ANSI.

Et si l'on désire afficher les résultats du programme 2.1

```
#include <stdio.h>

void main(void)      /* Ce programme calcule la circonférence et
                                la surface d'un disque de rayon 2 */
{
  const float PI = 3.141596;                    // initialisation de la constante PI
  int rayon = 2;                                // initialisation de la variable rayon
  float circonference, surface, temp;           // déclaration des variables réelles

  temp = PI*rayon;                              // affectation d'une variable temporaire
  circonference = 2*temp;                       // calcul de la circonférence
  surface = temp*rayon ;                        // calcul de la surface

  printf("La surface d'un disque de rayon %d vaut %f \n", rayon, surface);
  printf("La circonférence vaut %f\n", circonference);
}
```

Programme 2.3

Pour indiquer sous quel format (nombre de décimales, endroit,…) afficher ces variables, on utilise un *descripteur* d'affichage de type **%d** pour les entiers, **%f** pour les réels, **%c** pour les caractères et **%s** pour les chaînes de caractères.
Ainsi, le premier *printf* du Programme 2.3 :



```
printf("La surface d'un disque de rayon %d vaut %f \n", rayon, surface);
```

remplace %d dans la chaîne de caractères par la valeur entière du rayon et le %f par la valeur réelle de la surface. L'affichage à l'écran donne le résultat suivant :

```
La surface d'un disque de rayon 2 vaut 12.566384
```

Quelques descripteurs des paramètres passés en lecture ou en écriture.

"**%c**" : un caractère.
"**%d**" ou "**%i**" : entier signé.
"**%e**" : réel avec un exposant.
"**%f**" : réel sans exposant.
"**%g**" : réel avec ou sans exposant suivant les besoins.
"**%G**" : identique à g sauf un E à la place de e.
"**%o**" : le nombre est écrit en base 8.
"**%s**" : chaîne de caractère.
"**%u**" : entier non signé.
"**%p**" : pointeur.
"**%x**" ou "**%X**" : entier base 16 avec respect majuscule/minuscule.

Pour pouvoir traiter correctement les arguments du type **long**, il faut utiliser les spécificateurs **`%ld, %li, %lu, %lo, %lx`**.

On peut rajouter à ces formats des spécifications de mise en forme: longueur de champ, nombre de décimales. Par exemple:

%4d	affiche un entier sur 4 caractères	1234
%10.2f	affiche un réel sur 10 caractères, avec 2 décimales	-123456.78
%.4e	affiche un réel, en notation scientifique, avec 4 décimales	0.1234E+01
%10s	affiche une chaîne sur 10 caractères, en laissant des blancs à gauche	hello

2.11.2. La fonction de saisie scanf()

Le dual de ***printf***(), c'est la fonction ***scanf***(), définie aussi dans la librairie <stdio.h>.
Tout comme *printf*(), *scanf*() prend comme premier argument un format, et comme arguments suivants les variables à traiter.

Syntaxe : int **scanf**(const char *format [argument, ...]);
Description : lit à partir de *stdout* (clavier en principe), les différents arguments en appliquant le format spécifié.
Retour : retourne comme résultat le nombre d'arguments correctement reçus et affectés.

Attention, *scanf*() ne s'occupe que de lecture, et pas d'affichage. Dans le format, on ne mettra que des descripteurs des variables à lire. De plus, les variables à lire (donc modifiées, car elles auront une valeur différente à la sortie de *scanf*()) devront être passées par adresse. L'adresse mémoire d'une variable est indiquée par le nom de la variable précédé du signe **&**.

Sachez cependant que si on veut lire, par exemple, la valeur d'une variable entière rayon, on pourra le faire avec la commande

scanf("%d", &rayon); // **&**rayon est l'adresse en mémoire de la variable rayon.

Le symbole "**&**" est indispensable pour la lecture (*scanf*()), ce ne l'était pas pour l'affichage (*printf*()).

Nous pouvons maintenant améliorer notre programme 2.3. Le rayon peut alors prendre n'importe quelle valeur réelle puisque sa valeur est demandée à l'utilisateur et l'affichage des résultats est limité à 5 décimales.

```
#include <stdio.h>
void main(void)
{
        const float PI = 3.141596;                  // initialisation de la constante PI
        float circonference, surface, temp, rayon;  // déclaration des variables réelles

        printf("Rayon ?");                          // affichage du texte "Rayon ?"
        scanf("%f", &rayon);                        // attente d'un réel
        temp = PI*rayon;                            // affectation d'une variable temporaire
        circonference = 2*temp;                     // calcul de la circonférence
        surface = temp*rayon ;                      // calcul de la surface

        printf("La surface d'un disque de rayon %.5f vaut %.5f \n", rayon, surface);
        printf("La circonférence vaut %.5f\n", circonference);
}
```

Programme 2.4

L'utilisation de *scanf*() avec les caractères peut poser certains problèmes si l'on ne fait pas attention comme le montre les deux exemples suivants :

Exemples	**Entrées**	**Résultats**

char c1, c2, c3;
scanf("%c%c%c",&c1,&c2,&c3);
scanf("%c%1s%1s",&c1,&c2,&c3);
scanf(" %c %c %c",&c1,&c2,&c3);
	a b c
	a b c
	a b c
		c1='a' c2=' ' c3='b'
		c1='a' c2='b' c3='c'
		c1='a' c2='b' c3='c'

char c; int i; float x; scanf("%2d %5f %c",&i,&x,&c);	12 123.567 r	i=12 x=123.5 c=6

Autres exemples :

```
int JOUR, MOIS, ANNEE;
scanf("%d %d %d", &JOUR, &MOIS, &ANNEE);
```

Les entrées suivantes sont correctes et équivalentes:
1 1 2002 ou 1 01 2002 ou 01 01 2002
ou 1
1
2002

S'il n' y a pas d'espace entre les %d mais / :
scanf("%d/%d/%d", &JOUR, &MOIS, &ANNEE);
Les entrées suivantes sont correctes et équivalentes:
1/1/2002 ou 01/01/2002
Les entrées suivantes sont incorrectes:
1 /1 /2002 ou 1 1 2002

La suite d'instructions :
int RECU ;
RECU = scanf(« %d %d %d », &JOUR, &MOIS, &ANNEE) ;
printf(« \n %d %d %d %d\n », RECU, JOUR, MOIS, ANNEE) ;

donne les résultats indiqués dans le tableau suivant :

Introduit :		RECU	JOUR	MOIS	ANNEE
1 1 2002	**==>**	3	1	1	2002
1/1/2002	**==>**	1	1	-858993460	-858993460
1.1 2002	**==>**	1	1	-858993460	-858993460
1 1 20.02	**==>**	3	1	1	20

La première ligne de ce tableau indique que la variable RECU vaut 3, c'est-à-dire, la valeur retournée par *scanf* correspondant au nombre d'arguments correctement reçus et affectés. Les arguments et les affectations sont : JOUR =1, MOIS = 1 et ANNEE = 2002.
Il existe d'autres fonctions de saisie et d'affichage comme le montre le tableau 2.1 qui suit. Nous verrons plus tard plus en détail et d'une manière plus précise le rôle de ces fonctions. Elles se trouvent toutes dans la librairie ***stdio.h***.

Fonction	Syntaxe	Description
printf()	printf(const char *format [, arg [, arg]...]);	Écriture formatée sortie standard
scanf()	scanf(const char *format [, arg [, arg]...]);	Lecture formatée entrée standard
putchar()	putchar(int c);	Écrire le caractère c
getchar()	getchar() ;	Lecture d'un caractère
puts() **gets()**	*puts(char *s); *gets(char *s);	Ecriture/Lecture d'une chaîne de caractères, terminée par \n
sprintf() **sscanf()**	sprintf(char *s, char *format, arg ...); sscanf(char *s, char *format, pointer ...);	Ecrit dans la chaîne d'adresse s. Lit la chaîne d'adresse s.

Tableau 2.1

La librairie <***conio.h***> contient notamment la fonction ***getch***() qui permet d'attendre une saisie clavier. Elle est utile par exemple après un *printf*(), ce qui permet d'arrêter l'exécution du programme tant que l'utilisateur n'a pas introduit une valeur au clavier et ainsi lire tranquillement ce qui est affiché à l'écran. Cette fonction n'est pas compatible avec *ANSI-C* et elle peut seulement être utilisée sous MS-DOS.

Ce qu'il faut au moins retenir :

Avant d'utiliser des fonctions standards pré-programmées comme *printf*() et *scanf*(), il faut d'abord inclure la librairie standard I/O <**stdio.h**>.

Avant de manipuler une variable I (I++ , I-- , etc.) ou l'afficher, il faut l'initialiser.

La fonction *scanf*() exige l'adresse des variables de type *int, char, float, …*, alors que la fonction *printf*() n'a besoin que de la valeur comme le montre l'exemple suivant :

```
int k ;
scanf("%d ", & k) ;
printf(" La valeur de k est = %d", k) ;
```

Exercices

2.1. Citer parmi les identificateurs suivants ceux qui sont valides et ceux qui ne le sont pas.

a) recordb)long c) %interet d) étudiant e) MaVariable3
f) ma_var_4 g) 2x h) Long i) double j) ma-var-5

2.2. Citer les chaînes de caractères valides parmi les suivantes. Corriger les autres pour qu'elles deviennent correctes.

a) " Bonjour à vous ! \n" b) " Son nom est "Victor Dupont" "
c) ' C'est genial ' d) " Ce n\'est pas\ntoujours facile"

2.3. Les déclarations suivantes sont-elles correctes ? Les corriger si possible.

a) int i = 3, j , k = 5; **b)** float x = y = z = 1.0; **c)** char c = "bonjour"; **d)** char d = '\n';

2.4. Les appels suivants d'affichage par printf() vont-ils correctement s'exécuter ? A défaut, comment les corriger ?

```
int i = 7, j = 8;
char c = 'a';
float x = 3.5;
printf ("%i  %d  %c\n", i, j, c);
printf ("c vaut\n%c\n",c);
printf ("Voici les resultats:",i, j, "et encore", c);
printf ("%f %6f %6.2f", x, x, x);
printf ("%d %d", i + j*4, i*x);
```

2.5. Corriger le programme suivant pour qu'il s'exécute correctement.

```
Main(){
  int i, j ;
    printf(" i=%d alors entrer la valeur de j ", i) ;
    scanf("%d", j);
}
```

2.6. Corriger les erreurs de ce programme afin que l'on puisse afficher la valeur de j :

```
int main()
 float i, j;
{
  j=i;
  printf(‘’ valeur de j = %d ’’, j);
}
```

Chapitre 3 : Les structures de contrôle

Les structures de contrôle définissent la suite dans laquelle les instructions sont effectuées et la façon de contrôler le déroulement du programme. Dans ce chapitre, nous allons voir ce que met à notre disposition le langage C comme structures dites de contrôle de flux.
Constatons déjà que la particularité la plus importante des instructions de contrôle en C est le fait que les '*conditions*' en C peuvent être des *expressions quelconques qui fournissent un résultat numérique*. La valeur zéro correspond à la valeur logique *faux* et toute valeur différente de zéro est considérée comme *vrai*.

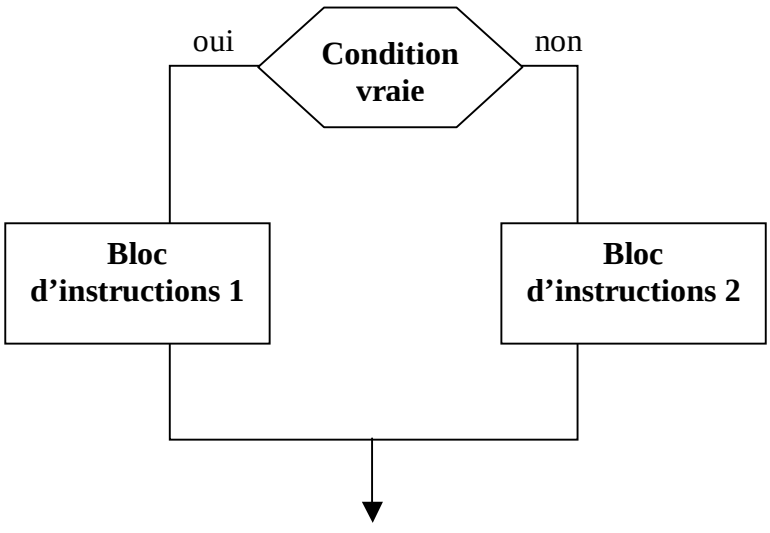
3.1. La structure alternative

La structure alternative (ou instruction conditionnelle) permet de ne pas exécuter systématiquement certaines instructions, mais seulement dans certains cas bien prévus par le programmeur.

En langage algorithmique une structure alternative peut s'écrire de la manière suivante :

```
si (<expression logique>)        // Si l'<expression logique> a la valeur logique vrai,
alors                            // alors
    <bloc d'instructions 1>      // ce bloc est exécuté
sinon                            // Si l'<expression logique> a la valeur logique faux,
    <bloc d'instructions 2>      // ce bloc est exécuté
fsi                              // fin de si
```

Son organigramme se présente comme suit :



Cette structure alternative se programme en C comme suit :

if (<expression logique >)
    <bloc d'instructions 1>
else
    <bloc d'instructions 2>

La partie **<expression logique >** peut désigner :

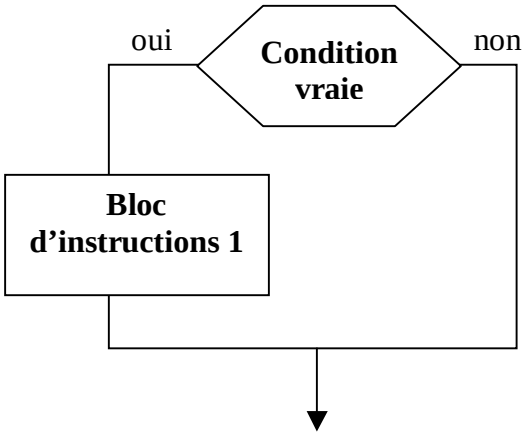
- une variable d'un type numérique,
- une expression retournant un résultat numérique.

La partie **<bloc d'instructions>** peut désigner :

- un bloc d'instructions compris entre accolades,
- une seule instruction terminée par un point-virgule.

Si <expression logique > est vrai (retourne une valeur différente de zéro) <bloc d'instructions 1> est exécuté. Dans le cas contraire c'est <bloc d'instructions 2> qui est exécuté.

On remarquera que l'équivalent de ''alors '', comme ''*then* '' en langage Pascal, n'existe pas. La partie **else** est facultative. On peut donc utiliser **if** sans la partie **else**.



Exemple 1

```
if (i < 10) i++;      //La variable i ne sera incrémentée que si elle a une valeur inférieure à 10.
```

Exemple 2

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

Le langage C possède un opérateur "***ternaire***" qui peut être utilisé comme alternative à **if -else** et qui a l'avantage de pouvoir être intégré dans une expression:

<expr1> ? <expr2> : <expr3>

L'exemple 2, en utilisant l'opérateur "ternaire" peut être écrit comme:

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

Exemple 3

```
if (A-B) printf("A est différent de B\n");
else printf("A est égal à B\n");
```

Exemple 4

```
if ((!recu) && (i < 10) && (n!=0) ){
        i++;
        moy = som/n;
        printf(" la valeur de i =%d  et moy=%f\n", i,moy) ;
 }
else {
   printf ("erreur \n");
   i = i +2 ;
}
```

Exemple 5

```
if ( (i%2) ==1 ) {          /* Attention de ne pas confondre == (opérateur logique d'égalité)
                                et = (opérateur d'affectation) */
        printf("Introduisez un entier  =  ");
        scanf(" %d", &j);
        printf ( "division par 2  =%5.2f\n", j/2.0);
}
else{
        printf("Introduisez l' entier %d ",i);
        scanf(" %d", &j);
        printf(" %d=%d ? \n", i,j);
}
```

Remarque importante :

Rappelez-vous que, x = 0 est une expression valide en C qui affecte à x la valeur zéro et qui retourne la valeur affectée, c'est-à-dire zéro.
Il est donc parfaitement légal d'écrire :

```
if (x = 0) {
        /* traitement */
        ...
}
```

Malheureusement le traitement particulier de x = 0 ne sera jamais appelé, et en plus dans le traitement des x différents de 0 la variable x vaudra 0 !

Il faut donc écrire :

```
if (x ==0) {
        /* traitement */
        ...
}
```

Parfois, on a besoin de traiter des décisions emboîtées: si ceci est vrai, alors faire cela; sinon, si ceci est vrai, faire cela,… On imbrique alors plusieurs instructions if-else l'une dans l'autre. En combinant plusieurs structures **if-else** en une expression nous obtenons une structure qui est très courante pour prendre des décisions entre plusieurs alternatives:

Les structures de décision imbriquées s'écrivent en C par la construction abrégée:

```
if ( <expr1> )
    <bloc1>
else if (<expr2>)
        <bloc2>
      else if (<expr3>)
              <bloc3>
            else if (<exprN>)
                    <blocN>
                 else <blocN+1>
```

Les expressions <expr1> ... <exprN> sont évaluées du haut vers le bas jusqu'à ce que l'une d'elles soit différente de zéro. Le bloc d'instructions lié à cette dernière est alors exécuté et le traitement de la commande est terminé.

Remarque :

Comme dans tout langage de programmation, il y a lieu d'être très soigneux dans l'indentation (structuration et alignement des instructions et des blocs). Dans le cas des structures **if-else** cela permet de bien cerner à quel instruction *if* chaque *else* se rattache.

En C, par convention, une partie *else* est toujours liée au dernier *if* qui ne possède pas de partie else.
Pour éviter des confusions et pour forcer une certaine interprétation d'une expression, il est recommandé d'utiliser des accolades { }.

Exemple 6

```
#include <stdio.h>
void main()
{
  char C;
        printf("Continuer (O)ui / (N)on ?");
        C=getchar();                    /* getchar : saisi un caractère de façon non formatée
                                        affectation du caractère à la variable C   */
        if (C=='O'){
            …..
           }
        else if (C=='N'){
                printf("Au revoir ...\n");
             }
              else{
                   printf("\a  Il faut choisir entre O et N !\n");
              }
}
```

3.2. La structure de sélection (*switch*)

L'instruction ***switch*** est une sorte d'aiguillage, elle est commode pour les "menus". Elle permet de remplacer plusieurs instructions **if-else** imbriquées. Cette instruction permet de gagner en lisibilité quand le nombre des **if-else** imbriqués augmente. La variable de contrôle est comparée à la valeur des constantes de chaque cas (***case***). Si la comparaison réussit, l'instruction du *case* est exécutée jusqu'à la première instruction ***break*** rencontrée.

```
switch (variable_controle)                  /* au cas où la variable vaut:  */
{
  case valeur1 : ensemble_instruction1;     /* cette valeur1(étiquette): exécuter ce bloc d'instructions.*/
        break;                              // sortie du case
  case valeur2 : ensemble_instruction2;
        break;
  case valeur3 : ensemble_instruction3;
        break;
        ..
        ..
  default : ensemble_instructionN;          /* cas par défaut si aucune des valeurs précédentes:
                                             exécuter ce bloc d'instructions. Facultatif mais recommandé */
}
```

variable_controle doit être de type *int, short, char ou long*.
break fait sortir du sélecteur. En l'absence de *break*, l'instruction suivante est exécutée; on peut ainsi tester plusieurs cas différents et leur attribuer la même instruction.
Le bloc "*default*" n'est pas obligatoire. *valeur1*, *valeur2*, …. doivent être des expressions constantes. L'instruction *switch* correspond à une cascade d'instructions if ...else

Exemple 7

```
char choix;
 printf("ensemble_instruction1 : TAPER 1\n");
 printf("ensemble_instruction2 TAPER 2\n");
 printf("POUR SORTIR TAPER 3\n");
 printf("\nVOTRE CHOIX: ");
        choix = getchar();          /* lecture d'un caractère isolé via la macro getchar() et
                                       son affectation à la variable choix   */
 switch(choix){
   case '1':  printf("\ensemble_instruction1 : case 1");
 break;
    case '2':  printf("\ensemble_instruction2 : case 2");
 break;
    case '3':   printf("\n  case 3 : FIN DU PROGRAMME ....");
  break;
   default :  printf("\nCE CHOIX N'EST PAS PREVU ");
 }
```

Notez la présence d'un ":" (deux points) après la valeur des étiquettes *case*.
l'étiquette " case **'1'**:" intercepte le cas où choix est égal à la valeur **'1'**, le bloc d'instruction simple :

```
printf("\ensemble_instruction1 : case 1");
```

est alors exécuté, puis l'instruction *break* nous fait sortir du bloc ***switch***. Cette instruction *break* est indispensable, sans elle l'exécution continuerait linéairement avec le bloc 2 :

```
printf("\ensemble_instruction2 : case 2");
…
```

l'étiquette " case **'2'**:" atteinte si choix est égal à **'2'**.
l'étiquette " case **'3'**:" atteinte si choix est égal à **'3'**.
l'étiquette " default : " atteinte si choix est différent de **'1', '2'** ou **'3'**.

Dans le cas où la variable ''choix '' est un entier, l'exemple 7 devient:

```
int choix;
 printf("ensemble_instruction1 : TAPER 1\n");
 printf("ensemble_instruction2 TAPER 2\n");
 printf("POUR SORTIR TAPER 3\n");
 printf("\nVOTRE CHOIX: ");
 scanf(" %d ", &choix);
 switch(choix){
  case 1:  printf("\ensemble_instruction1 : case 1");
break;
  case 2:  printf("\ensemble_instruction2 : case 2");
break;
  case 3:   printf("\n  case 3 : FIN DU PROGRAMME ....");
 break;
   default;  printf("\nCE CHOIX N'EST PAS PREVU ");
}
```

***Remarques* :**

- Une étiquette ne peut pas définir un intervalle de valeur. Par exemple "case 1..5:" ou "case 1&&5:" sont interdits; on écrira à la place plusieurs "case".
- Les valeurs des étiquettes n'exigent ni d'être ordonnées ni de former une série discrète (1, 2, 3…).
- La valeur d'une étiquette doit pouvoir être évaluée au moment de la compilation et être soit une valeur entière soit un caractère. Ainsi "case 5*j ou "case 6.0" ne sont pas correctes.
- L'étiquette "*default*" est optionnelle mais recommandée.
- Le *break* dans l'étiquette "*default*" n'est pas nécessaire car cette branche est la dernière des étiquettes *case* concernée par les instructions.

3.3. Les instructions répétitives

En C, nous disposons de trois structures de contrôle d'itérations, appelées aussi structures répétitives ou encore boucles, qui nous permettent d'exécuter plusieurs fois certaines phases de programmes. Les boucles conditionnelles sont:

1) la structure : **for** (pour… faire)
2) la structure : **while** (tant que),
3) la structure : **do - while** (faire… tant que)

Ces boucles ont toutes en commun le fait que l'exécution des instructions à répéter dépend, comme avec les instructions alternatives, d'une condition (<expression logique >).

3.3.1. Instruction for

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

```
for (initialisation ; <expression logique >  ; REinitialisation)
        {
                liste d'instructions
        }
```

avec liste d'instructions qui est une instruction simple ou composée. Ainsi si plusieurs instructions doivent être exécutées à chaque itération, on écrira :

```
for (initialisation ; <expression logique >  ; REinitialisation)
{
instruction 1;
instruction 2;
...
}
```

Exemple 8

```
x=0 ;
for (compt = 0 ; compt<=N; compt++)
{
        x += compt ;                    // x =x + compt
}
```

Cette boucle définit un compteur de nom *compt*, qui évolue de 0 à N en s'incrémentant à chaque passage. La variable x accumule la somme des valeurs attribuées à *compt* et réalise donc la somme des N premiers entiers.
Ce programme pourrait donc se traduire par "Initialiser un compteur *compt* à 0. Tant que le compteur est inférieur ou égal à N, incrémenter x de *compt*, incrémenter le compteur de 1".
Le compteur, une variable entière, prend successivement les valeurs 0, 1, .., N.

```
for (compt = 0 ; …..) {
```

Correspond à la première instruction "*initialisation*" qui sert à initialiser les variables. Il est recommandé que ces variables de contrôle de l'instruction **for** soient des variables locales.

```
for (compt = 0 ; compt<=N; …..) {
```

L'expression logique ou test "**compt<=N** ", est la condition de rebouclage ou condition d'arrêt. Elle est évaluée avant d'entrer dans la boucle. Si le test est vrai, le bloc est exécuté.

Si par exemple le test était le suivant :

```
for (compt = 0 ; compt<0; …..) {
```

le corps de la boucle ne serait jamais exécuté car *compt* (initialisé à zéro) ne vérifie pas la condition test (zéro n'est pas strictement inférieur à zéro).

for (compt = 0 ; compt<=N; **compt++**) {

La condition " *REinitialisation* " " **compt++**" permet de *re-initialiser* la variable de contrôle *compt*. Dans ce cas-ci, c'est une incrémentation. Cette dernière opération est effectuée à la fin de chaque tour de boucle.

Les trois expressions "*initialisation*", "<*expression logique* > " et " *REinitialisation*" peuvent comporter plusieurs instructions pouvant être de types différents. Il suffit pour cela de séparer les différentes instructions par des virgules à la place du point-virgule habituel.
Dans le programme (Programme 3.1) suivant :

```
#include <stdio.h>
void main()
{
   int i,j, k=9;
   float F, r;
   for ( i=0, j=5 , r=0.0; (i<j) && (k!=0) ; i++ , j++, k--, r+= 0.2) {
            F=(float)i/k ;
            F+=r;
           printf( "\n  i=%d, j=%d, k=%d et F=%f ", i, j, k, F);
    }
}
```

Programme 3.1

l'expression initialisation contient trois initialisations : i=0, j=5 et r=0.0. Remarquons que ces variables sont de types différents.
L'expression <*expression logique* > combine deux vérifications "i inférieur à j" et "k différent de zéro" grâce à l'opérateur "et logique" noté "&&".
L'expression " REinitialisation" *modifie quatre variables : i, j, k et r.*

```
#include <stdio.h>
void main()
{
     int I, nombre=1, pairs, impairs, Val_Max;
           pairs = 0;
           impairs = 0;

            printf("Val_Max ? =  ");
           scanf("%d", &Val_Max );

           // Saisie de Val_Max  nombres maximum avec décompte des nombres pairs et impairs

           for ( I = 0; I < Val_Max  && nombre != 0;  I++ , ((nombre % 2) == 0) ? pairs++ : impairs++ )
           {
              printf("\nTapez votre %d%s nombre  (0 pour sortir): ", I+1, (I== 0) ? "ier" : "ieme");
              scanf("%d", &nombre);
              printf("Votre nombre est %d\n", nombre);
           }
      printf("\n\nVous avez tape %d nombre(s) pair(s)", pairs);
      printf(" et %d nombre(s) impair(s)\n", impairs);
}
```

Programme 3.2

Dans ce dernier programme, le contenu de la boucle est répété tant que le compteur I est inférieur à *Val_Max* et que l'utilisateur ne rentre pas la valeur zéro.

Cet exemple montre que l'on peut éventuellement utiliser des tests dans l'expression "*REinitialisation*". En effet, on incrémente compteur ("I++") et, selon le résultat du test "nombre % 2" qui retourne 0 si nombre est pair et 1 s'il est impair, on incrémente "pairs" ou "impairs".

Les trois expressions "*initialisation*", "<*expression logique* > " et " *REinitialisation*" sont facultatives, ainsi le bloc :

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

est un bloc d'instruction valide, vide et répété à l'infini comme le montre le programme suivant :

```
#include <stdio.h>

void main()
{
   char c;

   // Saisie et affichage d'un caractère tant qu'il est différent de S

      for ( ;  ; )
      {
            printf("\nTapez votre caractere  (S pour sortir): ") ;
            c=getchar();
            if (c =='S')
                  break;
            else{
                  printf("Votre caractere  est %c ", c) ;
            }
      }
      printf("\n\n  Fin du programme" );
}
```

Programme 3.3

L'exécution de ce programme peut présenter des problèmes liés à l'acquisition des caractères. Elle peut se passer de la manière suivante :

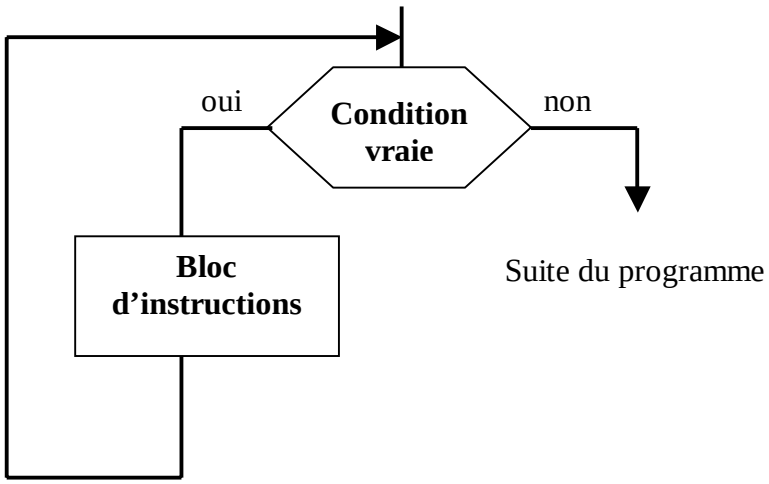
```
Tapez votre caractere  (S pour sortir): q
Votre caractere  est q
Tapez votre caractere  (S pour sortir): Votre caractere  est

Tapez votre caractere  (S pour sortir):
```

Nous laissons le soin à l'utilisateur de chercher et de découvrir la raison du problème. La réponse sera donnée plus loin et durant les séances de travaux et d'exercices.

3.3.2. Instruction while

Il s'agit de l'instruction: **tant que (**<expression logique >**)** // Condition vraie
faire{
BLOC D'INSTRUCTIONS
}

Organigramme:



L'instruction de boucle **while** permet de réaliser un traitement tant qu'une <expression logique > est vraie.

La syntaxe de l'instruction **while** est :

while (<expression logique > **)**
instruction **;**

Remarques :

L'expression <expression logique > test est toujours placée entre parenthèses et les {} ne sont pas nécessaires lorsque le bloc ne comporte qu'une seule instruction.
On peut rencontrer la construction suivante:

while (<expression logique >**);**

terminée par un '' ; '' et sans la présence du bloc d'instructions. Cette construction signifie: **"tant que l'expression est vraie attendre".**

Comme toujours, si le bloc instruction est composé de plusieurs instructions, on les placera entre les symboles de début et de fin de bloc: "{" et "}" soit :

while (<expression logique > **) {**
instruction 1**;**
instruction 2**;**
...
}

Le test se fait **d'abord,** le bloc d'instructions n'est exécuté que si <expression logique > est vraie(différente de 0). A chaque exécution du bloc, la condition est réévaluée, et ainsi de suite. Dès que l'expression est fausse, on sort de la boucle. Ainsi dans l'exemple suivant, le corps de la boucle sera exécuté 3 fois (valeurs i=3,4,5). A la sortie de la boucle, i vaudra 6.

```
int i=3;
while ( i < 6 ) {
i++;
}
```

Il est fréquent en C de voir une partie du travail ou la totalité reportée dans l'expression comme le montre le code suivant :

```
while ((c=getchar())!= EOL)
putchar(c) ;
```

Exemple 9

L'exemple 8 devient en utilisant la boucle **while** :

```
compt=0 ;
x=0 ;
while (compt<=N ) {
        x += compt ;
        compt++;
}
```

Cette boucle est équivalente à :

```
for ( ; compt<=N; ) {
        x += compt ;
        compt++;
}
```

En fait, la boucle *while* est une boucle *for* avec une initialisation avant la boucle et la re-initialisation à l'intérieur du bloc d'instructions.

Le programme 3.2 devient en utilisant la boucle **while** :

Programme 3.4

```
#include <stdio.h>
void main() {
    int I, nombre=1, pairs, impairs, Val_Max;
      pairs = impairs = I =0;

       printf("Val_Max ? = ");
      scanf("%d", &Val_Max );

      while (I < Val_Max  &&  nombre != 0) {
            printf("Tapez votre %d%s nombre  (0 pour sortir): ",I+1, (I == 0) ? "ier" : "ieme");
            scanf("%d", &nombre);
            printf("Votre nombre est %d\n", nombre);
            I++ ;
            ((nombre % 2) == 0) ? pairs++ : impairs++ ;
      }
      printf("\n\nVous avez tape %d nombre(s) pair(s)", pairs);
      printf(" et %d nombre(s) impair(s)\n", impairs);
}
```

Programme 3.5 : Saisir une suite de caractères, compter et afficher le nombre total de caractères, de lettres a et b.

```
#include <stdio.h>
#define EOL '\n'

void main()
{
  char c,
  int compt_a= 0,compt_b= 0, compt_tot= 0;
  printf("ENTREZ UNE PHRASE:   ");            // l'utilisateur saisit la totalité de sa phrase
    while((c=getchar())!=EOL) {               /*lors du 1er passage, getchar ne prend en compte
                                                  que le 1er caractère les autres sont rangés dans le tampon
*/
      if(c=='a') compt_a++;                   /* et récupérés par getchar lors des autres passages */
      if(c=='b') compt_b++;
      compt_tot++ ;
    }
   printf("NOMBRE DE a: %d, de b :%d  et le total = %d\n",compt_a, compt_b, compt_tot);
}
```

Programme 3.5

Voici un exemple d'exécution :

ENTREZ UNE PHRASE: **abracadabra magie**
NOMBRE DE a: 6, de b :2 et le total = 17

Programme 3.6 : un exemple qui interdit d'introduire des valeurs extérieures à un intervalle prédéfini.

```
#include <stdio.h>
#define NMAX 20

void main()
{
  int i, NE;

  printf("entrez une valeur entiere positive"
                                "inferieure ou egale a %d : \n", NMAX);
   scanf("%d",&NE);

  while((NE<=0)||(NE>NMAX))
   {
     if (NE<=0)
            printf("j'ai demande une valeur positive, redonnez la valeur : ");
     else
            printf("j'ai demande une valeur inferieure a %d, redonnez la valeur : ", NMAX);

     scanf("%d",&NE);
   }
}
```

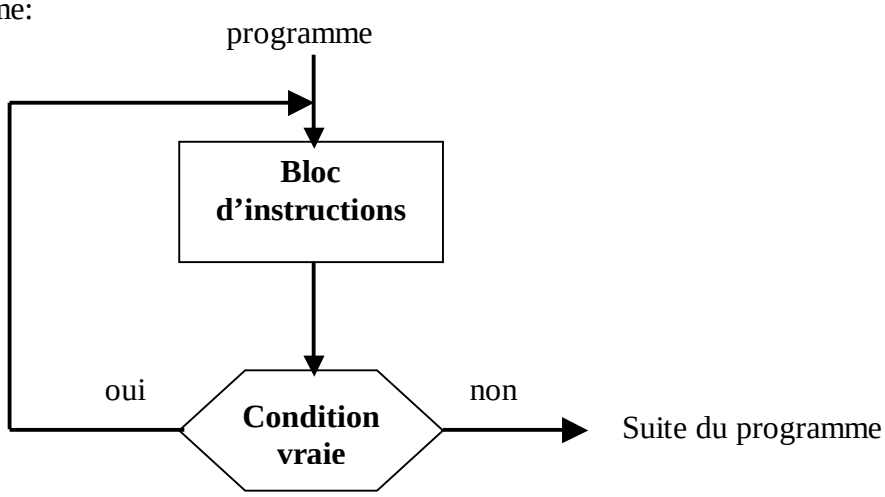
Programme 3.6

3.3.3. Instruction do ... while

Il s'agit de l'instruction:

```
faire{
        BLOC D'INSTRUCTIONS
}
tant que (<expression logique >)      // Condition vraie
```

Organigramme:



Il est parfois utile de faire le test non pas avant l'entrée de la boucle, mais à la sortie de celle-ci. Dans ce cas, il y aura toujours au moins une itération qui sera effectuée. Le C permet cette option par la construction do-while (faire…tant que…). La syntaxe de l'instruction est :

```
do {
        instruction1;
        instruction2;
        …..
} while (<expression logique >);
```

La condition est évaluée *après* le passage dans la boucle. Il faut noter l'existence du '' ;'' après *while*(<expression logique >)**.**
Le bloc d'instruction(s) est d'abord exécuté. L'expression de test <expression logique > (entre parenthèses après *while*) est ensuite évaluée. Si elle est fausse (valeur 0), on sort de la boucle. Si elle est vraie, on itère, en ré-exécutant le bloc d'instruction(s), et ainsi de suite, tant que le test de boucle est vrai.
De même, une boucle sans fin pourrait s'écrire comme:

```
do {
        instruction(s)
}while (1);
```

Il est évident que, bien souvent, on peut écrire avec un *while* (test avant) ou *for*, ce qu'on écrira avec un *do-while* (test après). Avoir les trois constructions facilite cependant la tâche du programmeur et l'écriture de programmes clairs et concis.

Exemple 10

Nous pouvons réécrire les instructions de l'Exemple 9 comme:

```
compt=0 ;
x=0 ;

do {
        x += compt ;
        compt++;
}
while (compt<=N );
```

Programme 3.7 : L'application de l'instruction *do ... while* au programme 3.5 nous donne :

```
#include <stdio.h>
#define EOL '\n'

void main()
{
  char c ;
  int compt_a= 0,compt_b= 0, compt_tot= 0;
  printf("ENTREZ UNE PHRASE:   ");                    // l'utilisateur saisit la totalité de sa phrase
   do {
        c=getchar() ;                                 /*lors du 1er passage, getchar ne prend en compte
                                                        que le 1er caractère les autres sont rangés dans le tampon
                                                        et récupérés par getchar lors des autres passages */
       if(c=='a') compt_a++;
       if(c=='b') compt_b++;
       compt_tot++ ;
    }   while(c!=EOL);

   printf("NOMBRE DE a: %d, de b :%d  et le total = %d\n",compt_a, compt_b, compt_tot);
}
```

Programme 3.7

L'exécution de ce programme donne cette fois-ci :

ENTREZ UNE PHRASE: **abracadabra magie**
NOMBRE DE a: 6, de b :2 et le total = 18

Il faut remarquer qu'ici **total = 18,** et pas à 17 comme dans l'exemple 7. En effet, après la phrase **abracadabra magie**, l'utilisateur termine par ''Enter'' ('\n') afin que la boucle s'arrête. Et comme le bloc d'instructions s'exécute avant d'arriver à la condition d'arrêt '' while(c!=EOL) '' , la variable *compt_tot* a été incrémentée par l'introduction du caractère ('\n').

3.4. Instructions de branchement

Les instructions de branchement transfèrent le contrôle du programme d'une instruction à une autre. Nous examinons ici les instructions :

break
continue
goto
return

La dernière instruction '' return'', sera traitée, pour des raisons pédagogiques, au chapitre des fonctions.

3.4.1. Instruction break

Elle ne peut s'utiliser qu'à l'intérieur d'une structure *for, while, do…while* ou *switch*. Elle provoque l'arrêt avant terme de ces instructions. Placée par exemple dans une boucle, l'instruction *break* provoque l'interruption immédiate de celle-ci et le contrôle est rendu à l'instruction suivante. Pour illustrer notre propos, considérons par exemple les instructions suivantes contenant deux boucles :

```
int i=0, j ;
 while (i<3){
    for (  j=0 ; j<10 ;  j++){
                if (j== 2)
                            break;
                else
                            printf("i=%d et j=%d\n ", i, j);
    }
  i++;
 }
```

L'exécution de ce code nous donne le résultat suivant :

```
i=0 et j=0
i=0 et j=1
i=1 et j=0
i=1 et j=1
i=2 et j=0
i=2 et j=1
```

En effet, chaque fois que j vaut 2, le *break* provoque l'arrêt de l'instruction *for* et rend le contrôle pour le i suivant du *while*.

3.4.2. Instruction continue

Elle ne peut s'utiliser qu'à l'intérieur d'une structure *for, while, do…while*. Alors que l'instruction *break* stoppe complètement une boucle simple, l'instruction ***continue*** permet de sauter un passage dans la boucle. L'exécution reprend alors au prochain passage dans la boucle. Les instructions suivantes permettent de comprendre l'effet de l'instruction *continue* :

```
int i=0, j ;
 while (i<3){
 for (  j=0; j<4; j++){
          if (j== 2)
                   continue;
          else
                   printf("i=%d et j=%d\n ", i, j);
 }
  i++;
 }
```

L'exécution de ce code nous donne le résultat suivant :

```
i=0 et j=0
i=0 et j=1
i=0 et j=3
i=1 et j=0
i=1 et j=1
i=1 et j=3
i=2 et j=0
i=2 et j=1
i=2 et j=3
```

En effet, chaque fois que j vaut 2, *continue* provoque un saut et passe à j=3 dans la boucle *for*.

3.4.3. Instruction goto

L'instruction ***goto*** provoque un saut à un endroit du programme repéré par une étiquette (label). Le programme continue alors à l'instruction qui se trouve à cet endroit-là. Cette instruction n'a pas les faveurs des programmeurs car son emploi abusif amène facilement à des programmes peu structurés et souvent illisibles. Voici un ensemble d'instructions qui illustre le fonctionnement de l'instruction *goto*:

```
int i=0, j ;
 while (i<3){
 for (  j=-2; j<4; j++){
          if (j== 0)
                   goto erreur;
          else
                   printf("i=%d j=%d i/j=%f\n ", i, j, (float)i/j);
 }
  i++;
 }

erreur : printf("Erreur car j =0\n ");
```

L'exécution de ce code nous donne le résultat suivant :

```
i=0  j=-2     i/j=0.000000
i=0  j=-1     i/j=0.000000
Erreur car j = 0
```

Ce qu'il faut au moins retenir :

1) Il est recommandé, afin de faciliter la lecture et le "débogage" de ne mettre qu'une seule instruction par ligne dans la source du programme et grâce à l'indentation des lignes, on fera ressortir la structure syntaxique du programme. Il est aussi recommandé de commenter les programmes et d'éviter les commentaires triviaux.

2)
```
if (i==j){          // pas i=j car c'est un test et pas une affectation
    …
}else
{
 ….
}
```

3)

```
int i;
scanf ("%d", &i);
switch (i) {
case valeur1 :
      ensemble_instruction1;
   break;
```

```
char i;
scanf ("%c", &i);
switch (i) {
case 'valeur1' :
     ensemble_instruction1;
   break;
```

4) Les trois structures de contrôle d'itérations :

```
for (initialis. ; ( ?test_vrai); REinitialisation)
{
   instruction 1;
   instruction 2;
}
```

```
while ( ?test_vrai)
{
   instruction 1;
   instruction 2;
}
```

```
do {
    instruction 1;
    instruction 2;
}while( ?test_vrai);
```

Exercices

3.0. Qu'affiche ce code à l'écran ?

```
int i=1;
while(i<5)
  printf("Dans while  %d\n",i);
  i++;
```

3.1. Le test suivant : if(choix == 'o' || 'O') peut-il remplacer if(choix == 'o' || choix== 'O'). Expliquer pourquoi ?

3.2. Que se passe-t-il si on retire tous les *break* de l'exemple 7 et que la valeur de la variable choix est égale à 1 ?

3.3. Ecrire un programme C qui calcule les racines d'un polynôme du second degré à coefficients réels. Examinez toutes les possibilités (racines réelles, complexes, doubles, infinité ou inexistence de solutions).

3.4. Ecrire un programme C qui calcule les 100 premiers nombres premiers.

3.5. Ecrire un programme qui demande le nombre d'éléments à acquérir N <= 10 et qui demande d'introduire soit un entier soit un caractère selon la valeur du compteur de la boucle *for*. Il faut donc suivre le schéma suivant :

for (i=0 ; i < N; ...
  Si i est impair : introduisez un entier et affichez sa moitié.
    printf (entier ?) scanf (entier) printf (entier /2)
  Si i est pair :
    Si i=2 ou i=6
      Calculez et affichez la somme des i de cette branche (2 puis 2+6)
    Sinon, introduisez un caractère et affichez-le.

3.6. Ecrire un programme C qui calcule le plus grand commun diviseur (PGCD) de deux nombres entiers positifs par l'algorithme d'Euclide.

3.7. Ecrire un programme qui calcule et affiche la factorielle d'un nombre entre 1 et 10. Tant que le nombre introduit est <=0 ou > 10, redemandez l'introduction de l'entier.

3.8. Ecrire un programme qui lit N nombres entiers au clavier et qui affiche leur somme, leur produit et leur moyenne. Choisissez un type approprié pour les valeurs à afficher. Le nombre N est à entrer au clavier. Résolvez ce problème en utilisant,

a) **while**,
b) **do - while**,
c) **for**.

Laquelle des trois variantes est la plus naturelle pour ce problème?

3.9. Modifiez l'exercice **3.8.** en ne demandant plus N, mais que la suite de chiffres non nuls entrés au clavier soit terminée par zéro.

3.10. Calculez le $N^{ième}$ terme U_N de la suite de FIBONACCI qui est donnée par la relation de récurrence:

$$U_1=1\ U_2=1\ U_N=U_{N-1} + U_{N-2}\ (\text{pour } N>2)$$

Déterminez le rang N et la valeur U_N du terme maximal que l'on peut calculer si on utilise pour U_N le type:

- **int**
- **long**
- **double**
- **long double**

3.11. Ecrire un programme qui calcule la série

$$S = 1 - \frac{X * 3!}{1 * 2} + \frac{X^2 * 5!}{2 * 4} - \frac{X^3 * 7!}{3 * 8} + \frac{X^4 * 9!}{4 * 16} \ldots..$$

L'utilisateur introduira la valeur de X et le nombre de termes à prendre en compte. Le calcul du $N^{ième}$ terme se fera à partir du N-1, …Il n'est pas demandé de chercher le terme général.