Sommaire

1.) STRUCTURE D'UN PROGRAMME

```
#include <STDIO.H>                          ←Inclusion des fichiers de bibliothèque.

#define PI 3.14                             ←Déclaration des constantes.

float   r,p;                                ←Déclaration des variables globales

void perimetre(float rayon,float *peri);    ←Déclaration des Prototypes. (Une
                                            déclaration de prototype se termine toujours
                                            par un point virgule ;).
void perimetre(float rayon,float *peri)     ←Déclaration des procédures, fonctions ou
                                            de Sous Programmes.
{                                           ←Début de la procédure
        *peri=2*PI*rayon;
}                                           ←Fin de la Procédure

void main()                                 ←Programme Principal ou ordonnancement.
{                                           ←Début du programme Principal
    r=3;
    perimetre(r,&p);                        ←Instructions
    printf("Le perimetre du cercle
    de rayon %f est égal à %f",r,p);
}                                           ←Fin du Programme Principal
```

**** Règles de bases:***

- En Langage **C**, il est souvent nécessaire d'inclure des fichiers dit d'en-tête (***HEADER:*.H***) contenant la déclaration de variables, constantes ou de procédures élémentaires. Le fichier à inclure se trouve en général dans le répertoire des fichiers d'en-têtes (**`Include`**) alors on écrira:
  `#Include` **<***Nom_du_fichier.H***>.**
  Si le fichier se trouve dans le répertoire courant, on écrira:
  `#Include` **"***Nom_du_fichier.H***"**

- Toutes instructions ou actions se terminent par un point virgule **;**

- Une ligne de commentaires doit commencer par **/*** et se terminer par ***/** et peut éventuellement être écrit sur plusieurs lignes.

- Un bloc d'instructions commence par **{** et se termine par **}**.

- Le langage C est sensible à la « casse » : les mots du langage doivent être écrits en minuscules ; les noms de variables, constantes, fonctions doivent toujours être écrits de la même façon : Ma_Fonction ≠ ma_fonction ≠ MA_FONCTION.

- Les seuls caractères autorisés pour les noms de variables, constantes… sont :
  - lettres NON accentuées ;
  - chiffres (sauf au début du nom) ;
  - caractères souligné « _ ».

2.) VARIABLES ET CONSTANTES

2.1.) Constantes

```
Syntaxe:  #define <identificateur> <valeur>
```

2.2.) Variables

```
Syntaxe:
<type> <identificateur1>,..,<identificateurn> ;
```

Type	*Taille (En bits)*	*Signé*	*Non Signé (Unsigned)*
Caractère → **Char**	8	-128 à +127	0 à +255
Entier → **Int**	16	-32768 à +32767	0 à +65535
Entier Court → **Short**	16	-32768 à +32767	0 à +65535
Entier long → **Long**	32	-2147483648 à +2147483647	0 à +4294967295
Réel → **Float**	32	+/- $3,4*10^{-38}$ à +/- $3,4*10^{+38}$	Aucune Signification
Réel double précision → **Double**	64	+/- $1,7*10^{-308}$ à +/-$1,7*10^{+308}$	Aucune Signification

Remarque: Lors de l'affectation des variables si on met **0** avant la valeur, elle sera en **Octal** et si on met **0x** devant une valeur elle sera **hexadécimale**.

2.1.) Les tableaux

```
Syntaxe:
<classe>  <type>  <identificateur>[nb.éléments  de  la  première
dimension]... [nb.éléments de nième dimension];
```

<classe> est facultatif

2.2.) Les chaines de caractères

```
Syntaxe:
<classe> <type> <identificateur>[nb de caractères+1];
```

<classe> est facultatif

3.) AFFICHAGE ET SAISIE

3.1.) L'Affichage

3.1.1.) Codes d'affichage

```
Syntaxe:
printf(<"Format">,identificateur1, ....,identificateurn);
```

Type	*Format*
Entier décimal	%d
Entier Octal	%o
Entier Hexadécimal (minuscules)	%x
Entier Hexadécimal (majuscules)	%X
Entier Non Signé	%u
Caractère	%c
Chaîne de caractères	%s
Flottant (réel)	%f
Scientifique	%e
Long Entier	%ld
Long entier non signé	%lu
Long flottant	%lf

On peut aussi définir le type d'affichage des variables pour les nombres signés ou flottants en rajoutant des caractères de remplissage.

- Un caractère de remplissage `'0'` au lieu de `' '` pour les numériques.
- Un caractère de remplissage `'-'` qui permet de justifier à gauche l'affichage sur la taille minimale (défaut à droite).
- Un caractère de remplissage `'+'` qui permet de forcer l'affichage du signe.
- Un nombre qui précise **le nombre de caractères qui doit être affiché** suivi d'un **point** et **d'un nombre précisant combien de chiffres après la virgule doivent être affichés**.

```
Syntaxe:
<Nb caractères affichés>.<Nombre de chiffres significatifs>
```

- `putchar`: Elle permet d'afficher un caractère à l'écran.

```
Syntaxe:
putchar(identificateur1);
```

3.1.2.) Codes de contrôle

Code de contrôle	*Signification*
`\n`	Nouvelle ligne
`\a`	Bip code ascii 7
`\r`	Retour chariot
`\b`	Espace arrière
`\t`	Tabulation
`\f`	Saut de Page
`\\`	Antislash
`\"`	Guillemet
`\'`	Apostrophe
`\'0'`	Caractère nul
`\0ddd`	Valeur octale (ascii) ddd
`\xdd`	Valeur hexadécimale dd

3.2.) La saisie

Codes de saisie

Type	*Format*
Entier décimal	%d
Entier Octal	%o
Entier Hexadécimal	%x
Entier Non Signé	%u
Caractère	%c
Chaîne de caractères	%s
Flottant	%f
Long Entier	%ld
Long flottant	%lf
Long entier non signé	%lu

- **`getchar`**: Elle permet de saisir un caractère au clavier.

Syntaxe:
```
identificateur1 = getchar( void );
```

- **gets**: Elle permet de saisir une chaîne de caractères au clavier.

Syntaxe:
```
gets(identificateur de chaîne de caractères);
```

4.) LES OPERATEURS

4.1.) L'opérateur d'affectation

Syntaxe:
`<identificateur> = <expression>;`

4.2.) Les opérateurs arithmétiques

+	Addition
-	Soustraction ou changement de signe
*	Multiplication
/	Division
%	Modulo (Reste)

++	Incrémente de 1
--	Décrémente de 1

- Si l'opérateur d'incrémentation ou de décrémentation est placé **avant** l'identificateur alors la variable sera incrémetée ou décrémentée **avant** d'être utilisée.

Les opérateurs binaires

&	ET
|	OU
^	OU Exclusif
~	Non (Complément à 1)
>>	Décalage à droite
<<	Décalage à gauche

4.3.) Les opérateurs combinés

+=	Addition et affectation
-=	Soustraction et affectation
*=	Multiplication et affectation
/=	Division et affectation
%=	Modulo et affectation
&=	ET et affectation
|=	OU et affectation
^=	OU exclusif et affectation
<<=	Décalage à gauche et affectation
>>=	Décalage à droite et affectation

4.4.) Les opérateurs relationnels

>	Supérieur à
>=	Supérieur ou égal à
<	Inférieur
<=	Inférieur ou égal à
==	Egal à
!=	Différent

4.5.) Les opérateurs Logiques

!	Négation
&&	ET Logique
||	OU Logique

4.6.) L'opérateur de conversion de type

- **La conversion implicite.** Elle est effectuée pour **évaluer le même type de données** lors d'évaluation d'expressions. Les conversions systématiques de `char` en `int`, en `float`, en `double`, la conversion se fait toujours **du type le plus petit** vers **le plus long** (exemple: de `char` vers `double`).

- **La conversion explicite.** On peut changer le type d'une variable vers un autre type en utilisant l'opérateur `cast` (type) en le mettant devant l'identificateur de variable à convertir.

Priorités

Le plus prioritaire ➔ *Le moins prioritaire*

Le plus prioritaire	`() [] -> .`
	`! - ++ -- - * & (case) sizeof`
	`* / %`
	`+ -`
	`<< >>`
	`== !=`
	`&`
⬇	`-`
	`|`
	`&&`
	`||`
	`? :`
	`= += -= *= /= %= >>= <<= &= |= ^=`
Le moins prioritaire	`,`

5.) STRUCTURES

5.1.) structures conditionnelles

< Si…….alors >

```
Syntaxe:
if (condition) instruction;

ou

if (condition)
{
      instruction1;
      ...
      instructionn;
}
```

< Si…….alors….Sinon >

```
Syntaxe:
if (condition) instruction;

ou

if (condition)
{
     instruction1;
     ...
     instructionn;
}
```

Structures de Choix

```
Syntaxe:
switch (identificateur)
{
case valeur1 :
         instruction_1 ;         ou         {instructions_1...} ;
     break;
case valeuurm :
         instruction_2 ;         ou         {instructions_2...} ;
     break;
case valeurk:
         instruction_j ;         ou         {instructions_j...} ;
     break;
default :
         instruction_i ;         ou         {instructions_i...} ;
     break;
}
```

5.2.) Les structures itératives

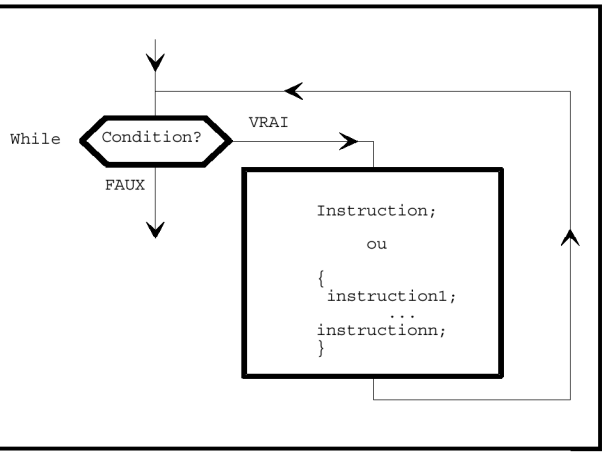
< Tant que…..Faire >

```
Syntaxe:

while (condition) instruction;

ou

while (condition)
{
      instruction1;
            ...
      instructionn;
}
```

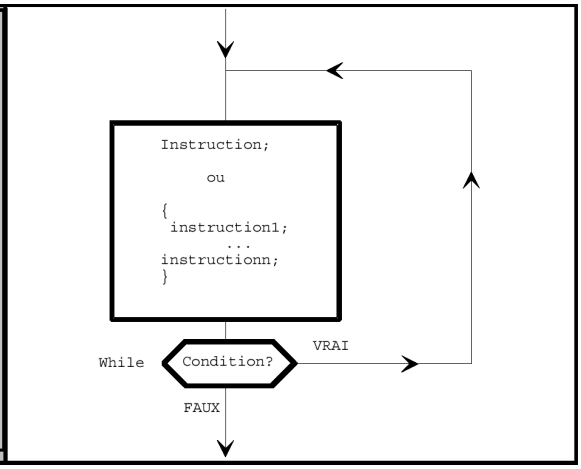


< Faire….Tant que >

```
Syntaxe:
do instruction;
while (condition) ;

ou

do
{
      instruction1;
            ...
      instructionn;
}
while (condition);
```

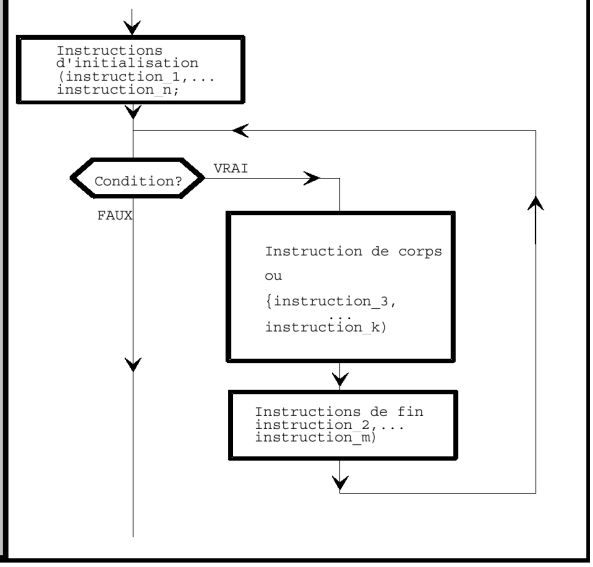


< Pour….Faire….Jusqu'à >

```
Syntaxe:
for
(instr_1,...,instr_n d'init;
   condition ;
instr_2,...,instr_m de fin)
 instruction de corps;

ou

for
(instr_1,...,instr_n d'init;
   condition ;
instr_2,...,instr_m de fin)
{
         instruction_3;
            ...
         instruction_k;
}
```



6.) LES POINTEURS

6.1.) L'opérateur d'adresse

Il permet de connaître l'adresse de n'importe quelle variable ou constante.

6.2.) Déclaration et manipulation de pointeur

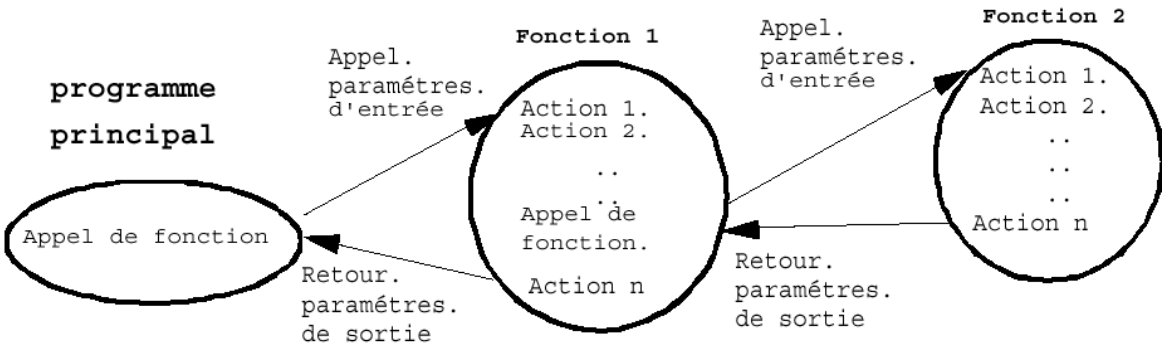
```
Syntaxe:
<classe> <type> *ptr_identificateur_de_pointeur;
```

<classe> est facultatif

6.3.) Arithmétique des pointeurs

- On peut incrémenter ou décrémenter, ce qui permet de sélectionner tous les éléments d'un tableau. Un pointeur sera toujours incrémenté ou décrémenté du nombre d'octets du type de variable pointée.
  Exemple: un **entier** occupe 2 octets donc un pointeur de type **int** sera incrémenté ou décrémenté de **deux octets**.
- Ajouter une constante **n**, qui permet au pointeur de se déplacer de **n** éléments dans un tableau. Le pointeur sera augmenté ou diminué de **n** fois le nombre d'octet(s) du type de la variable.
- Additionner ou soustraire une constante **n**, qui permet au pointeur de se déplacer de **n** éléments dans un tableau. Le pointeur sera augmenté ou diminué de **n** fois le nombre d'octet(s) du type de la variable.

7.) LES FONCTIONS



7.1.) La déclaration de prototype

```
Syntaxe:
<type_iden_sortie> iden_fonc(<type> iden_1,..., <type> iden_n);
```

7.2.) La déclaration de fonction

```
Syntaxe:
<type_iden_sortie> iden_fonc(<type> iden_1,..., <type> iden_n)
{
     /* Déclaration de variable(s) locale(s) */
     <type> iden_2,...,iden_m;
          .
          .
          .
     /* renvoie dans le paramètre de sortie */
     return(valeur);
}
```

7.3.) L'appel de fonction

```
Syntaxe:
iden_var_sortie = iden_fonc(iden_1_var,...,iden_n_var) ;

          ou

iden_fonc.(iden_1_var,...,iden_n_var) ;
```

7.4.) Fonctions sans paramètres d'entrée sortie

Elles servent en général à structurer un programme. Au lieu d'écrire des milliers de lignes dans le programme principal `main()` les unes derrière les autres, il vaut mieux les rassembler dans des fonctions.

Pour dire au compilateur qu'aucun paramètre n'est nécessaire, on utilise le mot clé `void` qui veut dire `rien`.

7.5.) Fonctions avec paramètres d'entrée sortie

Elles permettent de communiquer des valeurs de variables globales en général aux variables locales de la fonction appelée. En retour la fonction appelée peut renvoyer une valeur de retour.

7.6.) Fonctions avec paramètres d'entrée sortie (passage des paramètres par adresse)

C'est toute la puissance du **C** de pouvoir passer à une fonction les valeurs et ou **les adresses** de variables. Ainsi toute modification de valeur d'une variable globale dans une fonction sera répertoriée dans le programme principal, ce qui permet **de pouvoir passer des tableaux ou structures de données ou encore d'avoir plusieurs paramètres de sortie**.

7.7.) Fonctions standard du C

7.7.1.) Fonctions d'entrée sortie

Fonctions	Description
`getchar()`	Saisie d'un caractère.
`putchar()`	Affichage d'un caractère.
`printf()`	Affichage formaté.
`scanf()`	Saisie formatée.
`gets()`	Saisie d'une chaîne de caractères.
`puts()`	Affichage d'une chaîne de caractères.

7.7.2.) Les fonctions de manipulation de caractères

Fonctions	Description
`isascii()`	Test ASCII
`tiascii()`	Conversion numérique ASCII
`isalnum()`	Test lettre ou chiffre
`isalpha()`	Test Lettre majuscule ou minuscule
`iscntrl()`	Test caractère de contrôle
`isdigit()`	Test chiffre décimal
`isxdigit()`	Test chiffre hexadécimal
`isprint()`	Test caractère imprimable
`ispunct()`	Test ponctuation
`isspace()`	Test séparateur de mots
`isupper()`	Test lettre majuscule
`islower()`	Test lettre minuscule
`toupper()`	Conversion en majuscule
`tolower()`	Conversion en minuscule

Les fonctions `is....` renvoient `0` si **`FAUX`** ou différent de `0` si **`VRAI`**.

7.7.3.) Les fonctions mathématiques

Fonctions	Description
`abs()`	Valeur absolue.
`acos()`	Arc Cosinus.
`asin()`	Arc Sinus.
`atan()`	Arc Tangente.
`atan2()`	Arc Tangente X/Y.
`ceil()`	Arrondit par excès.
`cos()`	Cosinus.
`cosh()`	Cosinus Hyperbolique.
`exp()`	Exponentielle.
`fabs()`	Valeur absolue d'un nombre réel.
`floor()`	Arrondit par défaut.
`fmod()`	Modulo réel.
`frexp()`	Décompose un réel.
`ldexp()`	Multiplie un réel par 2^x.
`log()`	Logarithme népérien.
`log10()`	Logarithme décimal.
`modf()`	Décomposition d'un réel.
`pow()`	Puissance.
`sin()`	Sinus.
`sinh()`	Sinus Hyperbolique.
`sqrt()`	Racine Carrée.
`tan()`	Tangente.
`tanh()`	Tangente Hyperbolique.

8.) <u>AUTRES</u>

Voir l'aide de PIC C