Structure générale d'un programme C

1. Introduction:

Le langage C a été développé en 1972 par Dennis Ritchie dans les laboratoires Bell de la société AT&T. Bien que le langage C s'apparente à un langage assembleur indépendant de la machine plutôt qu'à un langage de haut niveau, il peut cependant être considéré comme un langage de programmation standard, en raison de sa grande popularité, de son association avec le système d'exploitation UNIX et de sa normalisation par l'ANSI (American National Standards Institute). Excepté quelques fonctions natives, le langage C est doté de fonctions indépendantes de la machine, contenues dans des bibliothèques auxquelles peut accéder tout programme écrit en C.

2. Structure générale:

La structure générale d'un programme C se présente dans l'ordre suivant:

- Inclusion des bibliothèques
- Déclaration des constantes
- Spécification des fonctions
- Déclaration des variables globales
- Programme principal
- Implémentation des fonctions

Par exemple:

```
#include <stdio.h>
#include <conio.h>  /*Inclusion des bibliothèques*/

#define TRUE   1
#define FAlSE  0    /*Déclaration des constantes*/

int Add (int,int);  /*Spécification des fonctions*/

int Sum;
char * Str;         /*Déclaration des variables globales*/

main()              /*Programme principale*/
 {

 }

int Add (int x, int y)   //Implémentation des fonctions
 {

 }
```

Exemple de programme en C:

```
/*Programme Bonjour*/
#include <stdio.h>

main()
 {
   printf("Bonjour");
 }
```

Remarque:

- *Dans le langage de programmation C, les lettres minuscules et les lettres majuscules sont différentes. Par exemple, l'identifiant 'Add' est différent de 'add'.*

- *Il n'y a que des fonctions dans le langage C.*

Types de données élémentaires, constantes et variables

1. Les types de données élémentaires:

En C, il existe un ensemble de type de données élémentaires ou simples qu'on peut classer en deux groupes:

- Les types scalaires.
- Les types réels.

a) Les types scalaires:

Ce sont les types de données qu'on peut énumérer, tel que les entiers, les caractères. Il existe principalement quatre types scalaires en C:

- char : caractère.
- int : entier.
- short : entier court.
- long: entier long.

b) Les types réels:

Ce sont les types de données qu'on ne peut pas énumérer. Il existe trois types en C:

- float : réel simple précision.
- double : réel double précision.
- long double : réel double précision long.

Le tableau suivant récapitule les types de données élémentaires utilisés en C:

Type	Occupation mémoire	Plage de valeur
char	1 octet (8 bits)	-128 → 127
unsigned char	1 octet	0 → 255
int	2 octets	–32768 → 32767
unsigned int	2 octets	0 → 65535
short	2 octets	–32768 → 32767
long	4 octets	–2,147,483,648 → 2,147,483,647
float	4 octets	3.4E +/- 38
double	8 octets	1.7E +/- 308
long double	10 octets	1.2E +/- 4932

Remarque:

Le type boolean (Vrais/Faux) n'existe pas en C, mais on peut le simuler par le type entier (int) et ce sera comme suit: Faux→ 0, Vrais→ les autres valeurs.

2. Les constantes:

Une constante est une donnée dont la valeur ne change pas au cours du programme. Par exemple:

- Décimal: 12
- Octal: 014
- Hexadécimal: 0xC, 0x1A
- Les constantes entiers long: 12L
- Les constantes à virgule flottante: 150.012 , 150 E-15
- Caractère: 'A', 'a'
- Chaîne de caractère: "Hello"

Remarque:

Il y a des symboles qui sont réservés tel que : [,] , ' , " , \ . Pour les utilisés dans les chaînes de caractère, il faut les procéder par \.

Exemple:

"What\'s up" pour obtenir What's up.

Pour déclarer une constante dans le langage C, on doit utiliser le mot réservé *define* dans la partie déclaration des constantes.

Par exemple:

```
#define NUMBER 12
#define TEXT "OK"
```

Remarque :

Il est conseillé de noter les constantes déclarées en majuscule pour les distinguées facilement dans le programme (conventionnel).

3. Les variables:

Une variable est une donnée dont la valeur change au cours du programme. Elle possède un nom, un type et qui est rangée à partir d'une certaine adresse en mémoire.

Avant qu'une variable ne soit utilisée dans le programme, il faut la définir (déclaration). Une telle définition détermine le nom, le type de la variable et lui réserve de l'espace mémoire conformément à son type.

La syntaxe de définition d'une variable en C est la suivante:

Type Nom_Var1 [, Nom_Var2 , …….]

Exemple:

```
int  x;
char c;
double y,z;
```

Remarque:

Le nom d'une variable doit commencer par une lettre et sa taille ne dépasse pas les 32c.

Les entrées sorties

1. L'affectation:

C'est une opération qui permet de donner une valeur à une variable. La syntaxe générale d'une affectation dans le langage C est la suivante:

Nom_Var = Valeur / Fonction / Expression / Variable

Exemple:

```
Num = 12;
Test = 1.2;
Nr = 'c';
```

2. L'initialisation:

L'initialisation c'est le fait de donner une valeur initiale à une variable lors de sa déclaration. La syntaxe de l'initialisation est la suivante:

Type Nom_Var = Valeur

Exemple:

```
int  i=1;
char c='A';
float k=12.3;
```

Remarque:

Dans l'initialisation Il faut que le type de la valeur affectée corresponde au type de la variable.

3. Les saisies formatées:

Pour saisir des données à partir du clavier, on doit utiliser la fonction prédéfinie *scanf* qui appartienne à la bibliothèque *<stdio.h>.* Sa syntaxe générale est la suivante:

scanf ("format1…",[argument1,…])

Format: indique le format qui correspond à la variable à saisir.
Argument: c'est l'adresse de la variable qu'on veut saisir.
Pour déterminer l'adresse d'une variable, il faut la procéder par &.

Exemple:

```
scanf("%d",&Num);//saisir la variable décimale Num
scanf("%c",&Ch); //saisir la variable caractère Ch
scanf("%f",&Rx); //saisir la variable réelle Rx
scanf("%d%d",&a,&b);//saisir les variables a et b
scanf("%s",Chc); //saisir la chaîne de caractère Chc
```

Remarque:

On ne mais pas de & devant une variable chaîne de caractère car elle représente déjà l'adresse de la chaîne.

On ne doit pas mettre de message dans scanf.

Donc pour chaque type de variable, on doit préciser le format correspondant. Le tableau suivant détermine ses formats:

Type	Format
int	%d
long	%ld
char	%c
float	%f
Chaine de caractère	%s

4. Les sorties formatées:

Pour l'affichage formaté des données, on utilise la fonction *printf* de la bibliothèque <*stdio.h*>. Sa syntaxe générale est la suivante:

printf ("Chaîne de caractère et format",[argument1…])

Argument: peut être une valeur une variable une expression ou une fonction.

Exemple:

```
printf("bonjour\n");// \n fait un saut de ligne
printf("La valeur de i est: %d",i);
printf("la case %d à comme valeur %f",i,r);
printf("%4d",32456);// le résultat sera 2456
printf("%5.3f",3.1415);//le résultat sera 3.142
//5 chiffres dont 3 avant la virgule
```

Remarque:

Si on veut préciser la largeur du format d'affichage on doit placer après le % un nombre qui détermine le nombre des caractères à afficher.

Exercice:

Ecrire un programme C qui permet de saisir une variable entière a, une variable réelle b et une variable caractère c. Puis afficher les trois variables sur la même ligne.

```
#include <stdio.h>
main()
{
     int a;    float b;  char c;
     printf("Donner l'entier a:");
     scanf("%d",&a);
     printf("Donner le réel b:");
     scanf("%f",&b);
     printf("Donner le caractère c:");
     scanf("%c",&c);
     printf("a= %d, b=%f, c=%c",a,b,c);
}
```

Opérateurs et expressions

1. Expressions:

- Une expression est composée d'opérandes et d'opérateurs.
- Un opérande isolé est déjà une expression par lui-même.
- Toutes expressions en C possèdent une valeur.
- Le type d'une expression dépend du type de ses opérandes.

Exemple:

```
2 + 3        x + y     float z=3.14;        long x=2,y;
```

2. Opérateurs:

Il existe quatre genres d'opérateur:

- Arithmétiques
- Logiques
- Relationnels (ou de comparaison)
- De bits

On distingue trois types d'opérateurs:

- Unaire (-, +)
- Binaire (+, -, *, /,…)
- Ternaire (le conditionnel (?))

a) Opérateurs arithmétiques:

Opérateur		Signification	Exemple
Binaire	+	Addition	X + Y
	-	Soustraction	X – Y
	*	Multiplication	X * Y
	/	Division	X / Y
	%	Reste de la division	X % Y
Unaire	-	Négation	- X

b) Conversion de type:

En C, il existe de conversion de type: implicite et explicite.

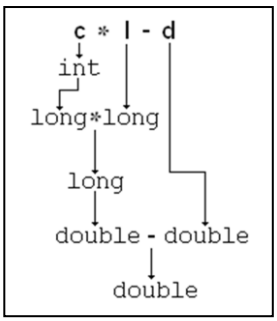
- Conversion implicite:

Ce type de conversion se fait automatiquement par le compilateur lors de l'affectation ou lors de l'évaluation des expressions. Généralement, la conversion se fait vers le type de plus grande capacité.

Exemple:

Conversion vers le type le plus haut.

```
char c;
long l;
double d;
```



Pour l'affectation le type de l'expression ou de la variable du membre droit est harmonisé avec celui de la variable du membre gauche.

Exemple:

```
int i;
i= 3.0; // harmoniser le float en int → i contient 3
long l;
l=i; // conversion vers le type le plus haut
```

- Conversion explicite:

Elle est réalisée par un opérateur de conversion de type. La syntaxe est la suivante:

(type) < expression ou constante ou variable >

Exemple:

```
int i;
long l;
i=(int) 3.0;
i=(int) l;
```

c) Opérateurs relationnels:

Opérateur	Signification	Exemple
= =	Egalité	X = = Y
<	Inférieur	X < Y
>	Supérieur	X > Y
<=	Inférieur ou égal	X <= Y
>=	Supérieur ou égal	X >= Y
!=	Différent	X != Y

Le résultat de l'évaluation d'une expression contenant un opérateur relationnel est Vrai ou Faux. Malheureusement en C, le type *boolean* n'existe pas. Pour cela, le langage C interprète le Vrai/Faux de la manière suivante:

- Une expression est logiquement fausse si sa valeur est 0.
- Une expression est logiquement vraie si sa valeur est ≠ 0.

d) Opérateurs logiques:

Opérateur	Signification	Exemple
&&	Et logique	X && Y
||	Ou logique	X || Y
!	Négation logique	! X

Remarque:

- *Toute expression logique doit être mise entre parenthèse.*
- *L'évaluation des expressions logiques se fait de gauche à droite. Donc pour un "ET" logique, si le premier opérande est évalué "Faux" alors le compilateur ne vérifie pas la valeur du deuxième opérande. Même chose pour un "OU" logique, si le premier opérande est "Vrai".*

e) Opérateurs de bits:

Opérateur	Signification	Exemple
&	Et	X & Y
|	Ou	X | Y
^	Ou exclusif	X ^ Y
~	Négation	~ X
>>	Déplacement des bits de X vers la droite Y position	X >> Y
<<	Déplacement des bits de X vers la gauche Y position	X << Y

f) Opérateurs d'affectation:

Les opérateurs d'affectation mettent dans leur opérande de gauche la valeur de leur opérande de droite.

Opérateurs	Signification	Exemple
++	Incrémentation de 1	i++, ++i
--	Décrémentation de 1	i--, --i
=	Affectation simple	X=Y
+=	X = X + Y	X+=Y
-=	X = X – Y	X-=Y
*=	X = X * Y	X*=Y
/=	X = X / Y	X/=Y
%=	X = X % Y	X%=Y
>>=	X = X >> Y	X>>=Y
<<=	X = X << Y	X<<=Y
&=	X = X & Y	X&=Y
|=	X = X | Y	X|=Y
^=	X = X ^ Y	X^=Y

Exemple:

```
int x,y,z;
x=1;
x=y=1;
y=x++; // affectation puis incrémentation
z=++x; // incrémentation puis affectation
```

g) Autres opérateurs:

Le langage C possède d'autre opérateur:

- **? :** opérateur conditionnel, sa syntaxe est Exp1 ? Exp2 : Exp3

Exemple:

```
int x=10,y=5;
(x!=y) ? x++ : y++; // x contient 11
```

- **, :** opérateur séquentiel qui permet de rassembler syntaxiquement deux expressions en une seule.

Exemple:

```
X++,y++;
```

- **sizeof :** calcule l'occupation mémoire en octet requise par une variable ou un type de donnée.

Exemple:

```
int i;
printf("%d",sizeof(i));
printf("%d",sizeof(int));
```

- **& :** fournit l'adresse de son opérande.

Exemple:

```
int x;  //&x fournit l'adresse de la variable x
```

- **. :** permet d'accéder à un champ d'un enregistrement.

Exemple:

```
printf("%s",Personne.nom);
```

- **-> :** permet d'accéder à un champ d'une variable qui pointe sur un enregistrement.

Exemple:

```
printf("%s",Pers->nom);
```

Structures conditionnelles

Les structures conditionnelles permettent d'exécuter une instruction ou un groupe d'instruction si une condition est vérifiée. En C, il existe trois types de structure conditionnelle à savoir: if, if-else, switch.

1. La structure "if":

La syntaxe générale de la structure "if" est la suivante:

if (condition)

< Suite d'instruction >

Si la condition est vraie alors la suite d'instruction est exécutée.

Exemple:

```
if(x==1)
    x++;
if (ok)
 {
    x=y+10;
    printf("%d",x);
 }
```

2. La structure "if - else":

C'est la forme générale de la structure conditionnelle "if", sa syntaxe est la suivante:

if (condition)

< Suite d'instruction1 >

else

< Suite d'instruction2 >

Si la condition est vraie alors la suite d'instruction1 est exécutée, si la condition est fausse alors la suite d'instruction2 est exécutée.

Exemple:

```
if (x>0)
     y=x+10;
else
  {
     y=0;
     x++;
  }
```

Remarque:
Si la suite d'instruction dépasse une ligne, il faut ajouter les crochés ({}).

3. La structure "switch":

C'est une structure conditionnelle à choix multiple, sa syntaxe est la suivante:

```
switch (variable)
  {
    case Valeur1:   <suite d'instruction1>
                    break;
    case Valeur2:   <suite d'instruction2>
                    break;
    ....
    default :       <suite d'instruction>
  }
```

Selon la valeur de la variable, la suite d'instruction respective sera exécutée.

Exemple:

```
switch (x)
  {
     case 1: y=2;
          break;
     case 2: y=5;
          break;
     case 3: y=10;
          break;
     default: printf("Erreur");
  }
```

Exercice:

Ecrire un programme C qui selon une valeur saisie entre 1 et 7 affiche le jour correspondant.

```
#include <stdio.h>
main()
{
     int x;
     printf("Donner le jour: ");
     scanf("%d",&x);
     switch (x)
       {
          case 1: printf("Lundi");      break;
          case 2: printf("Mardi");      break;
          case 3: printf("Mercredi");   break;
          case 4: printf("Jeudi");      break;
          case 5: printf("Vendredi");   break;
          case 6: printf("Samedi");     break;
          case 7: printf("Dimanche");   break;
          default : printf("Erreur");

       }
}
```

Structures itératives

Les structures itératives sont des structures qui permettent de répéter un ou plusieurs instructions un certain nombre de fois. Il existe principalement deux types de structure, l'une ayant un nombre de répétition connu et l'autre ayant un nombre de répétition inconnu.
En C, il existe trois forme itérative à savoir: "for", "while", "do while".

1. La structure itérative "for" (pour):

C'est une structure à nombre de répétition connu, sa syntaxe générale est la suivante:

for (Variable=Valeur_Debut; Variable<Valeur_Fin; Variable+=Pas)
    <Suite d'instruction>

Exemple:
On désire afficher les entiers de 1 à 5.

```
int i;
for (i=1;i<=5;i++)
     printf("%d \t",i);
// \t pour faire une tabulation entre les affichages
```

Remarque:

Si le nombre de ligne de la suite d'instruction dépasse une ligne, alors on doit ajouter les croches. (Même chose pour les autres structures)

2. La structure itérative "while" (tant que):

C'est une structure à nombre de répétition inconnu avec une condition d'arrêt. Sa syntaxe est la suivante:

while (condition)
    <Suite d'instruction>

Tant que la condition est vraie, l'itération continue.

Exemple:

```
int i=1;
while(i<=5)
   {
     printf("%d \t",i);
     i++;
   }
```

3. La structure itérative "do while"(répéter tant que):

Elle possède les propriétés de la structure "while" mais avec un changement de l'emplacement de la condition. Elle ressemble au schéma de la structure répéter jusqu'à. La syntaxe de la structure est la suivante:

do
    <Suite d'instruction>
while (condition);

On répète la suite d'instruction, tant que la condition est vraie.
Exemple:

```
int i=1;
do
 {
     printf("%d \t",i);
     i++;
 } while (i<=5);
```

Remarque: Il faut toujours vérifier la condition d'arrêt pour les structures "while" et "do while".

4. La structure itérative générale (itérer):

C'est une structure qu'on peut dégager de la structure "while" ou "do while" de la manière suivante:

```
while(1)
{
        <Suite d'instruction1>
        if (condition)
                break; // nous permet de sortir de la boucle si la condition
                       // est vraie
        <Suite d'instruction2>
}

do
{
        <Suite d'instruction1>
        if (condition)
                break;
        <Suite d'instruction2>
} while(1);
```

Exercice n°1:

Ecrire un programme C qui permet de calculer la somme des entiers de 1 à 10.

```
#include <stdio.h>
main()
 {
     int i, sum=0;
     for (i=1;i<=10;i++)
          sum+=i;
     printf("la somme est : %d",sum);
 }
```

Exercice n°2:

Ecrire un programme C qui permet de calculer le produit des entiers pairs de 2 à 20.

```
#include <stdio.h>
main()
 {   int i, prod=1;
     for (i=2;i<=20;i+=2)
          prod*=i;
     printf("le produit est : %d",prod);
 }
```

Types de données complexes

Le langage C dispose de deux types de données complexes à savoir les tableaux et les structures.
La différence entre les deux provient de la nature des éléments qu'on peut y ronger. Un tableau ne contient que des données de même type, tandis qu'une structure peut être composée à partir d'éléments dissemblables.

1. Les tableaux:

Un tableau est une variable qui se compose d'un certain nombre de données élémentaires de même type rongées en mémoire les unes à la suite des autres.
La définition d'un tableau unidimensionnel admet la syntaxe suivante:

<Type> <Nom_Tableau> [<Nombre_d'élément>]

Exemple:

```
int tab[10]; // un tableau de dix entiers
float tab[5]; // un tableau de cinq réels
```

La syntaxe d'accès à un élément du tableau est:

<Nom_Tableau> [<indice>]

Remarque:

Un tableau en C commence par l'indice 0.

Exemple:

```
int tab[5]; // les éléments de 0 à 4
tab[0]=1;
```

Il existe deux moyens pour initialiser un tableau:

- En utilisant une structure itérative pour parcourir le tableau case par case.
- Soit une initialisation à la définition.

La syntaxe d'initialisation d'un tableau à la définition est la suivante:

<Type> <Nom_Tableau> [<Nombre_d'élément>] = {V_1,….,V_n}

Exemple:

```
int tab[3]={20,5,15};
```

La définition d'un tableau bidimensionnel est la suivante:

<Type> <Nom_Tableau> [<Nombre_d'élément >][<Nombre_d'élément >]

Exemple:

```
int mat[2][3];
int Tx[2][2]={5,9,1,6};
mat[0][0]=0;
mat[1][2]=10;
```

Remarque:

On ne peut pas affecter deux tableaux directement.

2. Les chaînes de caractères:

En C, les chaînes de caractère sont des tableaux de caractères. Toute chaîne de caractère se termine par un caractère de fin de chaîne "\0", donc au niveau de la déclaration d'une chaîne, il faut prévoir un caractère de plus.
Exemple:

```
char chaîne[21];\\20 caractères + \0
```

On peut faire des initialisations de la manière suivante:

```
char ch[10]={'a','b','c','d','\0'};
char nom[20]="Ahmed";
char nx[]="ali";
```

L'ajout du caractère de fin de chaîne se fait par le compilateur.

Remarque:

Pour saisir une chaîne de caractère par scanf, c'est inutile de mettre &.

```
scanf("%s",chaîne);
```

En C, il est interdit d'affecter deux chaînes de caractère.

Il existe un ensemble de fonction appartenant à la bibliothèque <string.h> qui permette de manipuler les chaînes de caractère:

- **Fonction de copy:**
  strcpy (<destination> , <source>)
  Exemple:

```
char ch[25];
strcpy(ch,"abcd");
```

- **Fonction de calcul de taille d'une chaîne:**
  strlen(<chaîne>)
  Exemple:

```
char ch[5]="abc";
int i;
i=strlen(ch);// i contient 3
```

- **Fonction de concaténation:**
  strcat(<chaîne1>,<chaîne2>)
  Le résultat sera dans la chaîne1:c'est la fusion de la chaîne1 et la chaîne2
  Exemple:

```
char fn[10]="source";
char ex[3]=".c";
strcat(fn,ex);// fn contient "source.c"
```

- **Fonction de comparaison de chaîne:**
  strcmp(<chaîne1>,<chaîne2>)
  si chaîne1=chaîne2 alors la fonction retourne 0
  Exemple:

```
char c1[5]="abc";
char c2[10]="abc";
if (!strcmp(c1,c2))
     printf("c1=c2");
```

3. Les structures:

Une structure peut contenir des variables de type différent. Chaque variable de la structure est appelée champ. La syntaxe de définition d'une structure est la suivante:

```
struct <Nom_Structure>
{
      <Type> <Nom_Champ1>
      <Type> <Nom_Champ2>
      ....
};
```

Exemple:

```
struct personne
      {
            char nom[21];
            char prenom[16];
            int age;
      };
```

Pour déclarer une variable de type structure, il faut appliquer la syntaxe suivante:

struct <Nom_Structure> <Nom_Variable>

Exemple:

```
struct  personne x;
x.age=20;
```

Pour déclarer un tableau de structure, la syntaxe est:

struct <Nom_Structure> <Nom_tableau>[<Nombre_Element>]

Exemple:

```
struct personne Tx[10];
Tx[0].age=10;
```

Exercice n°1:

Ecrire un programme C qui permet de saisir les éléments d'un tableau de 5 entiers.

```
#include <stdio.h>
main()
 {
     int Tx[5],i;
     for(i=0;i<5;i++)
          scanf("%d",&Tx[i]);
 }
```

Types de données personnalisées

Le langage C nous permet de définir des nouveaux types de données et ceci en utilisant les mots clés *typedef* et *enum*.

1. Typedef:

Elle nous permet de définir des nouveaux types à partir d'un ancien type. La syntaxe d'utilisation est la suivante:

typedef <type> <nouveau_type>

Exemple:

```
typedef  int entier;
entier i; // i de type entier

typedef struct personne
          {
               char nom[20];
               int age;
          } pers;
```

On peut faire par la suite:

```
pers x; // au lieu de struct personne x;
pers y[4]; // déclare un tableau de 5 structures.
```

Remarque:

En réalité typedef nous permet de renommer un type et non de créer un nouveau type.

2. Enum:

Elle permet de rassembler dans une variable la liste de valeur qu'elle peut prendre. Cette liste doit être homogène: liste de noms, couleurs…Sa syntaxe est la suivante:

enum <nom_type> { nom 1,….nom n};

Exemple :

```
enum continents {Afrique, Amerique, Asie, Europe,
Australie};
```

Pour définir une variable du type déclaré:

```
enum continents conti;
conti=Asie;
```

Remarque :

Par défaut, le premier nom de la liste à la valeur 0, le deuxième à la valeur 1 et ainsi de suite. Toutefois, il est possible d'attribuer des valeurs spécifiques à chaque nom lors de la déclaration.

Exemple :

```
enum season {spring=0, summer=4, autumn=8,
winter=12};
```

Il est possible de déclarer une variable de type énumération lors de la déclaration.

```
enum boolean {true, false} check;
```

Exemple:

```
#include <stdio.h>
enum month(jan, feb, mar, apr, may, jun, jul, aug,
sep, oct, nov, dec};

main()
{
     int i;
     for(i=jan; i<=dec; i++)
          printf("%d",i);
}
```

Exemple:

```
#include <stdio.h>
enum  day{Sunday=1,  Monday,  Tuesday=5,  Wednesday,
Thursday=10, Friday, Saturday};

main()
{
     Printf("%d %d %d %d %d %d %d", Sunday, Monday,
Tuesday, Wednesday, Thursday, Friday, Saturday);
}
```

Classes de mémorisation

La classe de mémorisation d'une variable détermine ou elle sera rangée dans la mémoire centrale.

1. Variables locales et globales:

Selon l'emplacement de définition d'une variable dans le programme, on parle soit de variable locale ou de variable globale. Toutes variables déclarées à l'intérieur de n'importe quelle fonction est locale celle-ci même pour la fonction *main*. Les variables globales sont déclarées en dehors de la fonction *main*.

2. Classes de mémorisation:

Toute variable globale ou locale possède une seule classe de mémorisation.

a) Classes de mémorisation des variables locales:

- **Auto:**

  La classe de mémorisation automatique est attribuée à chaque variable même si le mot clé "auto" est oublié. Une variable de classe auto est créée à l'exécution du bloc et détruite automatiquement à la fin de son exécution.

  Exemple:

```
main()
{
      auto int x;
}
```

- **Static:**

  Contrairement aux variables de classe auto, les variables de la classe de mémorisation statique possèdent une durée de vie qui n'est pas limitée au bloc de la fonction mais au programme entier. Donc sa valeur ne changera pas sauf par la fonction qui l'utilise.

  Exemple:

```
main()
{
      static int j=0;
}
```

- **Register:**

  La classe de mémorisation registre est une démonstration de la force majeure du langage C de pouvoir manipuler les registres du microprocesseur. Généralement, ce sont les compteurs qu'on déclare de classe registre pour gagner en temps et en espace mémoire. La déclaration d'une variable de classe registre ne met pas systématiquement la variable dans l'un des registres du microprocesseur sauf disponibilité.

Exemple:

```
register int Rx;
```

b) Classes de mémorisation des variables globales:

Toutes les variables globales appartiennent par défaut à la classe de mémorisation externe. Les variables de la classe externe sont connues par toutes les fonctions du programme y compris les fonctions d'un autre programme appelé.

Le pré processeur C

Le pré processeur C, c'est la partie du programme dont on peut faire les opérations suivantes:

1. Inclusion des bibliothèques.

Pour utiliser les différentes fonctions qui appartiennent à une bibliothèque.
Exemple:

```
#include <stdio.h>
#include <string.h>
#include "myfile.c"
```

2. Déclaration des constantes.

Exemple:

```
#define TRUE 1
#define MaxValue 100
```

3. Définir des macros.

Les macros sont des fonctions rapides qui réalisent des opérations simples.
Exemple:

```
#define MINUS (x,y) (x-y)
#define MAX (x,y) (x>y?x : y)
#define COUCOU() printf("Coucou");
#define MAJEUR(age) if (age >= 18) \
                    printf("Vous etes majeur\n");
```

4. Effectuer des compilations conditionnelles.

Il est possible de réaliser des conditions en langage préprocesseur pour faire des compilations conditionnées.

a. #if

```
#if condition
    /* Code source à compiler si la condition est
vraie */
#elif condition2
    /* Sinon si la condition 2 est vraie compiler ce
code source */
#endif
```

Le mot-clé #if permet d'insérer une condition de préprocesseur.#elif signifie else if(sinon si). La condition s'arrête lorsque vous insérez un #endif. Vous noterez qu'il n'y a pas d'accolades en préprocesseur.

L'intérêt, c'est qu'on peut ainsi faire des compilations conditionnelles.
En effet, si la condition est vraie, le code qui suit sera compilé. Sinon, il sera tout simplement supprimé du fichier le temps de la compilation. Il n'apparaîtra donc pas dans le programme final.

b. #ifdef

Nous allons voir maintenant l'intérêt de faire un #define d'une constante sans préciser de valeur, comme je vous l'ai montré précédemment:

En effet, il est possible d'utiliser #ifdef pour dire si la constante est définie. #ifndef, lui, sert à dire si la constante n'est pas définie.

On peut alors imaginer ceci:

```
#define WINDOWS

#ifdef WINDOWS
 */    Code source pour Windows/*
#endif

#ifdef LINUX
 */    Code source pour Linux/*
#endif

#ifdef MAC
 */    Code source pour Mac/*
#endif
```

C'est comme ça que font certains programmes multi-plates-formes pour s'adapter à l'OS par exemple.
Alors, bien entendu, il faut recompiler le programme pour chaque OS (ce n'est pas magique). Si vous êtes sous Windows, vous écrivez un #define WINDOWS en haut, puis vous compilez.
Si vous voulez compiler votre programme pour Linux (avec la partie du code source spécifique à Linux), vous devrez alors modifier le define et mettre à la place :#define LINUX. Recompilez, et cette fois c'est la portion de code source pour Linux qui sera compilée, les autres parties étant ignorées.

c. #ifndef

Pour éviter les inclusions infinies, il est possible d'utiliser #ifndef pour vérifier si cette partie a été déjà incluse ou non par le préprocesseur.

```
#ifndef DEF_NOMDUFICHIER // Si la constante n'a pas été définie le fichier n'a jamais été inclus
#define DEF_NOMDUFICHIER // On définit la constante pour que la prochaine fois le fichier ne soit plus inclus

/* Contenu de votre fichier (autres include, prototypes, define...) */

#endif
```

Les fonctions

A la différence des autres langages de programmation, le langage C ne manipule que des fonctions, même le programme principal *main* est une fonction.

1. Définition:

Une fonction est un sous-programme qui renvoi une valeur d'un seul type.

a) Spécification:

Une spécification d'une fonction est le fait de déclarer un prototype à cette fonction avant de l'implémenter. Sa syntaxe est la suivante:

Type Nom_Fonction ([type1, type2….])

Exemple:

```
float cube (float);
```

b) Implémentation:

Une implémentation c'est le corps principal de la fonction que nous avons déjà spécifiée. Sa syntaxe est la suivante:

```
Type Nom_Fonction ( [type1 var1, type2 var2….])
 {
      <Déclaration variable locale>
      <Suite d'instruction>
      [return (expression)]
 }
```

Exemple:

```
float cube (float x)
 {
     return(x*x*x);
 }
```

c) Appel:

Pour appeler une fonction dans le programme principale ou dans une autre fonction, la syntaxe est la suivante:

Nom_Fonction ([paramètre1, paramètre2…])

Exemple:

```
main()
 {
     float i=2, c;
     c=cube(i);
 }
```

Remarque:

On peut simuler une procédure en C par une fonction qui ne retourne rien (void).

2. Passage de paramètre:

En C, il existe deux modes de passage de paramètre:

- **Passage par copie de valeur:**

Ce passage permet de faire une copie de valeur des paramètres effectifs aux paramètres formels. Donc, tout changement des variables formels dans la fonction n'affectera pas les paramètres effectifs.

Exemple:

```
#include <stdio.h>
void permut (int,int);

main()
 {
     int a=5,b=3;
     printf("la valeur de a=%d et b=%d",a,b);
     permut(a,b);
     printf("la valeur de a=%d et b=%d",a,b);
 }

void permut (int x, int y)
 {
     int bf;
     bf=x;
     x=y;
     y=bf;
 }
```

On remarque que les valeurs de a et de b n'ont pas changé.

- **Passage par adresse:**

Ce passage fournit l'adresse des paramètres effectifs aux paramètres formels. Donc, tout changement des variables formels dans la fonction entraîne un changement dans les variables effectifs.

Exemple:

```
#include <stdio.h>
void permut (int *,int *);

main()
 {
     int a=5,b=3;
     printf("la valeur de a=%d et b=%d",a,b);
     permut(&a,&b);
     printf("la valeur de a=%d et b=%d",a,b);
 }

void permut (int * x, int * y)
```

```
{
    int bf;
    bf=*x;
    *x=*y;
    *y=bf;
}
```

On remarque que les valeurs de a et de b ont changé.

3. Pointeurs vers les fonctions:

Bien qu'en C les fonctions ne soient pas des entités variables, elles possèdent des adresses en mémoire. L'adresse d'une fonction est matérialisée par son nom. Pour ranger l'adresse d'une fonction dans un pointeur, il faut le définir de la manière suivante:

Type (* Nom_Pointeur) ([type1 var1,….])

Exemple:

```
#include <stdio.h>
int sqr (int);
main()
 {
    int (*pf)(int x);
    pf=sqr;
    printf("%d",pf(3));
 }
int sqr(int x)
 {
    return (x*x);
 }
```

Les fichiers

1. Déclaration:

Dans le langage C, un fichier est une suite d'octets manipulée par un pointeur. La déclaration d'un fichier passe par la déclaration d'un pointeur.
Exemple:

```
FILE * fp;
```

Il existe un ensemble de fonctions qui permettent la manipulation des fichiers. Ces fonctions appartiennent à la bibliothèque <*stdio.h*>.

2. Ouverture de fichier (fopen):

C'est la fonction *fopen* qui permet d'ouvrir un fichier, sa syntaxe est la suivante:

FILE * fopen (char * Nom_Fichier_Externe, char * mode)

Exemple:

```
FILE * fp;
fp= fopen("test.dat","r");
```

Remarque:

Si la fonction fopen n'arrive pas à ouvrir le fichier spécifié, alors elle retourne la valeur NULL.

Les modes d'accès en C sont:

Lettre	Désignation
r	Ouverture en lecture seule, le fichier doit exister.
w	Création d'un fichier pour écriture (si fichier existant: écrasement)
a	Ouverture en ajout à la fin de fichier. (si fichier inexistant: création)
r+	Ouverture d'un fichier existant pour mise à jour (lecture/écriture)
w+	Création d'un fichier pour mise à jour (lecture/écriture)
a+	Ouverture en ajout à la fin de fichier (si fichier inexistant: création)
t	Fichier de type texte
b	Fichier de type binaire

Exemple:

```
FILE * fp;
fp= fopen("stock.txt","rt");
```

3. Fermeture d'un fichier:

Si on veut fermer un fichier déjà ouvert, on doit utiliser la fonction *fclose*. Sa syntaxe est la suivante:

int fclose (FILE * Pointeur_Fichier)

Exemple:

```
fclose(fp);
```

4. L'accès séquentiel:

C'est le parcours du fichier enregistrement par enregistrement du début jusqu'à la fin.
Pour effacer un enregistrement, on doit le supprimer logiquement, car sa suppression physique est impossible.

- **Les ordres de lecture/écriture formatés:**

Pour lire à partir d'un fichier, on doit utiliser la fonction *fscanf*, sa syntaxe est la suivante:

int fscanf (FILE * Pointeur_Fichier, "Formats",var1,var2..)

Pour écrire dans d'un fichier, on doit utiliser la fonction *fprintf*, sa syntaxe est la suivante:

int fprintf (FILE * Pointeur_Fichier, "Formats",var1,var2..)

Exemple:

```
fscanf(fp,"%d",&a);
//Lire une variable entière du fichier.
fprintf(fp,"%d",a);
// Écrire une variable entière dans le fichier.
```

- **Les ordres de lecture/écriture par blocs:**

Pour lire à partir d'un fichier, on doit utiliser la fonction *fread,* sa syntaxe est la suivante:

size_t fread(void * pointeur_bloc, taille_bloc, nombre_enrg, FILE * Pointeur_Fichier)

Pour écrire dans un fichier, on doit utiliser la fonction *fwrite,* sa syntaxe est la suivante:

size_t fwrite(void * pointeur_bloc, taille_bloc, nombre_enrg, FILE * Pointeur_Fichier)

Exemple:

```
int x=10;
fwrite(&x,sizeof(x),1,fp);
```

5. L'accès direct:

C'est le branchement direct sur un enregistrement quelconque du fichier sans le parcourir. Le fichier devient comme un tableau.

- **Fseek:**

Pour positionner le pointeur du fichier sur l'enregistrement voulu, on doit utiliser la fonction *fseek*. Sa syntaxe est la suivante:

int fseek(FILE * Pointeur_Fichier, long offset, int base)

Le paramètre offset donne le nombre d'octet de décalage du pointeur relativement au troisième paramètre base.
Si le paramètre offset est positif le déplacement se fait vers la fin du fichier et inversement.

Le paramètre de base est récapitulé par le tableau suivant:

Valeur	Constante	Signification
0	SEEK_SET	Début de fichier
1	SEEK_CUR	Position courante du pointeur
2	SEEK_END	Fin de fichier

Exemple:

```
fseek(fp,0L,2);
//le pointeur du fichier est sur la fin de fichier.
```

- **Rewind:**

Cette fonction permet de positionner le pointeur au début du fichier, sa syntaxe est la suivante:

int rewind(FILE * Pointeur_Fichier)

Exemple:

```
rewind(fp); // équivalente à fseek(fp,0L,0)
```

- **Ftell:**

Elle retourne la position en octet du pointeur de fichier dans le fichier, sa syntaxe est la suivante:

long ftell(FILE * Pointeur_Fichier)

Exemple:

```
long Taille;
int nbr_enrg;
fseek(fp,0L,2);
Taille=ftell(fp);
nbr_enrg=Taille/sizeof(pers);
```

- **Feof:**

Elle vérifie si le pointeur du fichier est pointé sur la fin du fichier ou non, si oui elle retourne 1, sinon elle retourne 0.

int feof(FILE * Pointeur_Fichier)

Exemple:

```
if (feof(fp))
     printf("Fin du fichier");
```