Classes Informatique de Gestion

Programmation structurée et Langage C

O. BEN AHMED DAHO
Benahmed@unilim.fr

2000-2007 ***Quatrième version***

SOMMAIRE

Présentation générale du Langage C

I. Historique

Le langage C est un langage développé, aux environs des années 72 par un chercheur des laboratoires AT&T américains, du nom de Dennis Ritchie qui, avec Ken Thompson, travaillait à la réalisation d'un système d'exploitation connu ensuite, sous le nom de UNIX.

Le succès du C a été casuel, parce qu'à l'origine il avait été conçu non pas dans un but d'exploitation commerciale, mais plutôt pour des exigences liées à un environnement industriel, ou le but primaire était d'avoir à disposition un langage agréable dans l'utilisation, puissant et fiable.

Les ancêtres du C, c'est à dire, les langages qui l'ont inspiré, peuvent être ainsi résumés:

- ✓ ***Algol 60 :*** projeté en 1960 par un comité international.
- ✓ ***CPL : (Combined programming language)*** projeté en 1963 à l'université de Cambridge.
- ✓ ***BCPL : (Basic CPL)*** projeté en 1967 par Martin Richards à Cambridge.
- ✓ ***B :*** Projeté en 1970 par Ken Thompson chez les laboratoires Bell.
- ✓ ***C :*** Projeté en 1972 par Dennis Ritchie.

Le langage Algol conçu peu d'années après le Fortran, était beaucoup plus sophistiqué, et a influencé de façon plus ou mois sensible sur la plupart des langages de programmation suivants. Ses caractéristiques consistaient dans la régularité de la syntaxe et dans la structure par modules. A cause de son excessive abstraction et de sa généralité, ce langage n'a jamais eu une grande diffusion aux Etats Unis.

Avec le langage CPL, les chercheurs ont essayé de réduire les caractéristiques trop abstraites de l'Algol, cependant le CPL se révéla aussi un langage trop complexe, avec des caractéristiques et des attributs qui le rendaient difficile a comprendre.

Le BCPL a été une évolution simplifiée du CPL.

Le B, développé par Ken Thompson pour une première version de l'UNIX, est une autre simplification du CPL, très lié à la structure de l'ordinateur disponible a ce moment là par l'auteur.

Les deux langages ont trop poussé en avant la recherche de la simplicité et de la compacité, et il en est résulté des langages peu généralisés, efficaces seulement pour certains problèmes.

Dennis Ritchie a essayé avec le "C" de maintenir un certain niveau de généralité, surtout dans la gestion des types de données, sans sacrifier l'adhérence à la structure de l'ordinateur, qui avait été l'inspiration fondamentale du CPL.

Une autre caractéristique du "C", comme du "BCPL" et du "B", est la cohérence dérivante du fait d'avoir été conçu par la même personne.

Naturellement les langages projetés par la même personne reflètent l'expérience personnelle. En effet Dennis Ritchie travaillait dans la programmation de système, c'est à dire les systèmes d'exploitation, les langages, les compilateurs, etc., et le langage "C" se révèle surtout puissant pour ce type d'application.

Cela ne veut pas dire qu'il ne soit pas possible d'utiliser le "C" pour d'autres applications. En effet, à l'aide de sa structure par modules, il est possible de l'adapter à n'importe quel environnement, à condition, bien sûr, d'avoir déjà développé les routines nécessaires.

La souplesse du "C" consiste dans le fait qu'étant un langage de niveau relativement bas, elle permette d'optimiser le rendement des logiciels à cause de sa proximité à la structure matérielle de l'ordinateur.

En revanche, le "C" est un langage de niveau relativement haut, qui fait abstraction de l'architecture de l'ordinateur, et qui permet d'augmenter le rendement par le programmeur. Le "C" représente un compromis entre le BCPL et le B à un niveau plus bas, et l'ALGOL à un niveau plus haut.

C'est pourquoi quand on défini le "C" comme un langage soit de bas niveau, soit de haut niveau, on veut souligner que le "C" ne se situe pas exactement dans la hiérarchie des langages, parce qu'il est suffisamment près de la structure de l'ordinateur, pour laisser au programmeur un bon contrôle sur les détails qui peuvent accroître le rendement du logiciel, mais suffisamment de haut niveau pour faire abstraction de l'environnement hardware sur lequel on travaille.

Le langage "C" a suivi les évolutions de l'UNIX (V.5 en 1973, V.6 en 1974, V.7 en 1979), arrivé à la version « System V » en 1984. Aujourd'hui, il s'est démarqué de l'UNIX. Il a évolué vers la programmation orientée objet en 1983 : le C++ est le résultat du mariage de "C" et de *Simula*, le premier langage de programmation orientée objet, né en 1967.

Le langage "C" est utilisé sur de nombreux systèmes : MS-DOS, VMS, … Diverses versions du "C" existent : Microsoft C, Turbo C et Turbo C++ de Borland et QuickC de Microsoft sont les populaires.

Le but de ce cours, ce n'est pas de donner seulement une liste détaillée de toutes les commandes, parce qu'il existe de nombreuses publications bien plus complètes à ce sujet, mais d'analyser, hors du contexte syntaxique du langage, les caractéristiques essentielles du "C" et de mettre en évidence les principaux aspects.

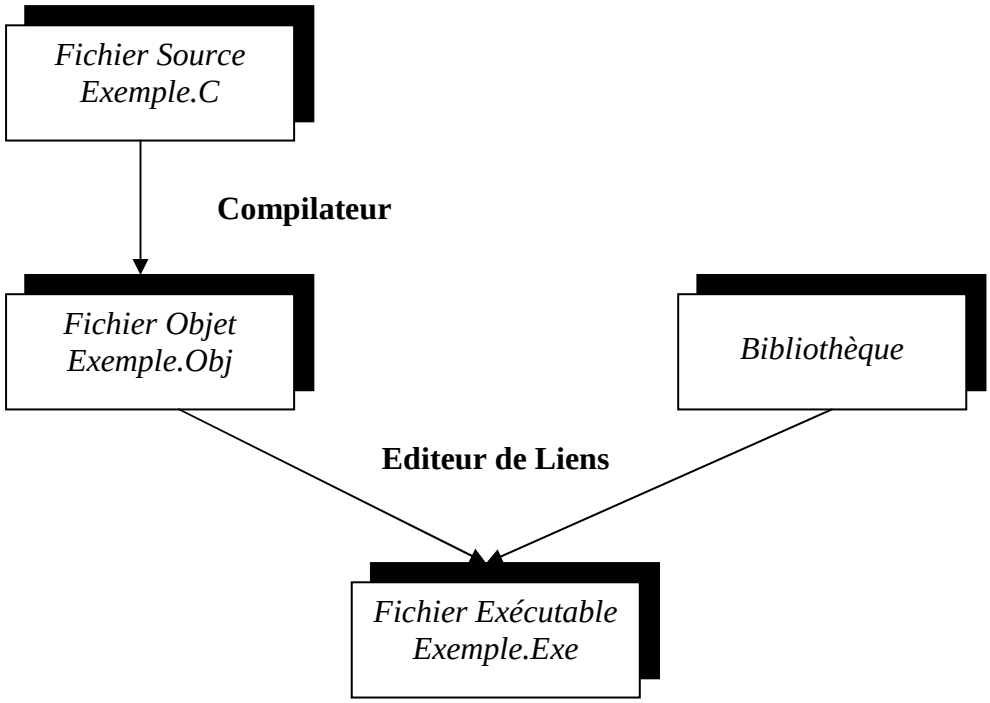
II. Structure d'un programme

1. Environnement de programmation

Le langage C n'est pas un langage interprété, c'est un langage compilé, c'est à dire traduit dans le langage du processeur pour pouvoir être exécuté.

Les compilateurs C mettent a la disposition du programmeur un environnement de développement ou EDI (lntegrated Development Environment) permettant l'écriture, la compilation, avec détection des erreurs, et l'exécution des programmes.

Un programme source écrit en C (**fichier.c**) est d'abord compilé (production d'un **fichier .obj**), puis "linké" par l'éditeur de liens (production d'un **fichier.exe**), avant de pouvoir être exécuté. Le langage C offre un certain nombre de fonctions prédéfinies qui résident dans la bibliothèque du C. L'éditeur de liens utilise cette bibliothèque pour générer le fichier exécutable.



Afin de rechercher les erreurs dans un programme, un débogueur est intégré. Il permet de faire fonctionner le programme pas à pas et de suivre à chaque étape ce qu'il réalise.

2. Aspect général d'un programme

D'une manière générale, un programme C est constitué de séquences d'objets de données et de corps de fonctions. Les séquences de description de données sont situées a l'extérieur des corps de fonctions. Les objets de données, externes aux fonctions, sont en fait des variables globales utilisables par les fonctions du programme, elles-mêmes considérées

comme externes. Tous ces éléments, variables externes et fonctions sont visibles entre eux au niveau de exécution du programme.

- **Les fonctions**

Une fonction est un sous-programme auquel est transmise une liste d'arguments (qui peut être vide) et qui retourne une valeur a la fonction appelante.

La notion de "procédure" n'existe pas en C, mais est équivalente à une fonction qui ne retourne pas de valeur ou a un appel de fonction qui n'affecte pas de variable.

Les fonctions peuvent posséder aussi leurs propres objets, dits variables locales, non visibles à l'extérieur des fonctions.

Les variables doivent toujours être déclarées avant leur utilisation. Leur déclaration se fait en tête du corps de la fonction ou en tête du programme.

Un programme C n'est donc constitué, mises à part les variables externes, que de fonctions pouvant apparaître dans n'importe quel ordre dans le fichier source.

- **Les fichiers**

Les fonctions peuvent être physiquement regroupées dans un même fichier source ou éparpillées dans différents fichiers.

Une directive d'inclusion de fichier source ***#include*** permet d'inclure, à tout endroit du fichier principal, des fichiers sources secondaires.

- **Le programme principal**

Que le programme C soit monolithique ou morcelé en plusieurs fichiers, une fonction particulière joue le rôle de fonction principale, appelée "*programme principal*":

Elle a pour nom ***main*** et est reconnue par l'éditeur de lien comme étant la fonction qui doit recevoir le contrôle de l'exécution après la phase de chargement.

Exemple :

```
main()
{

…

fonction (erg 1, arg2);

}

fonction (param1, param2)
{

...

}
```

- **Les commentaires**

Les commentaires en C sont placés entre **/*** et ***/** sans espace entre **/** et ***.**

Exemples :

```
/* Ceci est un commentaire correct */
```

ou encore,

```
/**********************************
***          exemple de commentaire   ***
***          sur plusieurs lignes          ***
**********************************/
```

Lorsqu'on ouvre un commentaire par /* ne jamais oublier de le fermer par */, car la séquence qui suit, ou la totalité du programme serait ignorée par le compilateur.

III. Les éléments de base du langage

Le langage C comporte les unités syntaxiques suivantes :

- **Les séparateurs**

Se sont constitués d'un ou plusieurs caractères "espace", ou d'un ou plusieurs caractères de tabulation.

- **Les identificateurs**

Se sont les différents noms utilisés pour définir les objets manipulés par un programme, notamment les constantes, les variables et les fonctions.

L'identificateur est composé d'une suite de lettres et de chiffres sans espace, le premier caractère étant obligatoirement une lettre. Le caractère souligné "_" est aussi utilisé.

En C, les lettres minuscules et majuscules ne sont pas équivalentes, ainsi annee, Annee et ANNEE sont trois identificateurs différents.

- **Les mots réservés**

Le langage C possède un ensemble de mots clés nécessaires à la sémantique du langage, ne pouvant pas être utilisés comme identificateurs. La liste des mots réservés est la suivante :

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

Outre les mots réservés figurant dans la liste ci-dessus, un certain nombre de fonctions sont intégrées dans la bibliothèque standard du C. Leur nom ne peut, bien entendu, être utilisé comme identificateur. Parmi les plus utilisées, citons ***printf*** et ***scanf***.

- **Les délimiteurs**

Les délimiteurs sont des caractères spéciaux permettant au compilateur de reconnaître les différentes unités syntaxiques du langage. Les principaux délimiteurs sont les suivants:

; termine une déclaration de variable ou une instruction.
, sépare deux éléments consécutifs dans une liste.
() encadre une liste d'arguments ou de paramètres.
[] encadre la dimension ou l'indice d'un tableau.
{} encadre un bloc d'instructions ou une liste de valeurs d'initialisation

- **Les constantes**

Il existe plusieurs types de constantes qui sont liées à l'architecture de la machine. Dans ce qui suit nous détaillerons leur déclaration et leur utilisation.

- **Les opérateurs**

Il existe trois types d'opérateurs:

- opérateurs unaires: précèdent un identificateur, une expression, une constante,
- opérateurs binaires: mettent en relation deux termes ou expressions,
- opérateurs ternaires: mettent en relation trois termes ou expressions.

L'ensemble de ces opérateurs sera détaillé dans le chapitre suivant.

IV. Les constantes

Les objets de base manipulés par les programmes sont les constantes et les variables pour lesquelles le langage C offre trois types de base :

char	pour caractère
int	pour entier
float	pour flottant

On peut utiliser la directive de compilation ***#define*** ou le mot clé ***const*** pour définir et utiliser une constante :

```
#define     nom     "Dupond"
#define     entier  99
```

ou :

```
const  char  nom = "Dupond" ;
const  int   entier = 100 ;
```

La directive ***#define*** initialise les constantes **nom** et **entier** et ne comporte pas de signe "=". L'instruction ne se termine pas par un " ;".

Dans l'utilisation du mot clé ***const***, le signe "=" définit une affectation. Les constantes **nom** et **entier** sont affectées d'une valeur d'initialisation qui ne pourra pas être modifiée.

La directive #***define*** peut aussi être utilisée pour définir des ***macros*** personnelles.

Exemple :

#***define*** lire ***scanf***

#***define*** ecrire ***printf***

Dans la suite du programme, on pourra remplacer : ***scanf*** par lire et ***printf*** par ecrire.

V. Les types de données et variables

Il existe essentiellement deux types de données en C : les entiers (integer) et les données exprimées en virgule flottante (floating point). Par ces deux types ont dérivés les caractères (character) et les données en double précision.

1. Les entiers

- **La déclaration d'un entier**

Un entier peut être positif ou négatif, et ses valeurs maximales et minimales dépendent du mot physique du processeur : pour un processeur 16 bits, ce sera 2 octets, pour un processeur 32 bits, ce sera 4 octets.

La déclaration des variables entières qui doit être à chaque début de code, a l'aspect suivant :

```
int a ;
int b ;
int c ;
…
```

ou bien, si nous voulons regrouper la déclaration de plusieurs variables :

```
int a, b, c ;
```

ou en utilisant d'autres identificateurs plus significatifs :

```
int annee, numero, age ;
```

Une fois une variable entière est déclarée, il est possible de lui assigner une valeur par l'intermédiaire de l'opérateur "=", comme dans l'exemple suivant :

```
int annee, numero, age ;
annee = 2000 ;
numero = 100 ;
age = 31 ;
```

- **Les entiers de type short et long**

Le Langage C disposent de deux variantes pour l'utilisation des entiers : l'entier long et l'entier court. La déclaration d'une variable de type long (***long integer***) est la suivante :

```
long int a ;
```

La variable long demande le double d'espace utilisé par un entier (l'espace alloué en mémoire dépend de la machine). L'usage des entiers de type long peut ralentir l'exécution d'un programme mais le champ des valeurs représentables est énormément élargi.

L'autre variante de l'entier est le ***short***. Cette variante peut être utilisée pour des variables dont le champ des valeurs est inférieur aux entiers ordinaires, et l'usage rend (théoriquement) l'exécution des programmes plus rapide. La dimension dépend aussi de la machine du compilateur considéré. La déclaration de l'entier court a la forme suivante :

short int a ;

Si la taille des entiers par défaut est 16 bits, celle des entiers courts serait la même, et celle des entiers long serait représentée par une séquence de 32 bits.

Le C permet, dans l'en-tête déclaratif, d'abréger les déclarations de ***short int*** et ***long int*** en ***short*** et ***long***.

- **Les entiers sans signe (unsigned)**

Si, à priori, une variable entière ne prend jamais des valeurs négatives, dans ce cas, cette variable peut être déclarée sous la forme de ***unsigned*** :

unsigned int a ;

A travers cette déclaration, le champ des valeurs que la variable peut prendre pour une taille de 16 bits, passe de (-32768 , +32767) à (0 , +65535).

2. Les caractères

En C, un caractère est représenté par son code numérique (généralement en ASCII, American Standard Code for Information Exchange) et peut donc être manipulé comme un entier. Un caractère peut donc être défini de deux façons.

```
char    carac = 'A' ;
char    carac = '\65' ;     /*** Code ASCII de A ***/
```

3. Les données en virgule flottante

Dans le langage C, une variable en virgule flottante (***floating point***), peut être représentée soit en format décimal soit en format exponentielle. Le mot clé qui permet de déclarer les variables flottantes est ***float*** :

```
float x,y,z ;

…

x = 54.123 ;
y = 0.9845 e 4 ;
z = 23486.02 e –5 ;
```

La précision de la mantisse varie en fonction du compilateur.

4. Les données en double précision

Une variable en double précision est tout simplement une variable en virgule flottante représentée par davantage de chiffres décimaux, et qui demande souvent le double d'espace utilisé par la variable de type ***float***. La déclaration des variables en double précision est la suivante :

double a ;

L'usage de ce type de données améliore la précision des opérations arithmétiques et réduit l'effet dû à l'approximation.

5. Format de données

Lors des opérations d'entrée et de sortie, notamment avec ***scanf()*** et ***printf()***, il faut, en général, préciser le format des variables affichées.

Exemple :

scanf (" %d %d %d ",&annee, &numero, &age) ;

printf (" Année %d, Numéro %d, Age %d \n ",annee, numero, age) ;

Les variables annee, numero et age sont de type entier. Dans les deux instructions, il y a autant de spécifications de format qu'il y a de paramètres à lire ou à afficher.

Pour chaque type de données est associé un format.

VI. Les structures de contrôle

Les structures de contrôle, dans un langage de programmation, permettent de spécifier quelles opérations doivent être exécutées, et dans quel ordre. Ces structures déterminent donc le flux de contrôle d'un programme.

1. Les sélections

- **IF Simple**

La forme la plus simple de cette structure est la suivante :

if (**<Condition>**) **<Instruction>** ;

Exemple *:*

if (a<b) ***printf*** ("a est inférieur à b "\n) ;

L'instruction **<Instruction>** n'est exécutée que si la condition **<Condition>** préalablement définie se trouve remplie. Dans le cas contraire, le programme saute l'instruction et passe à la ligne suivante. La condition est toujours entre parenthèses. L'instruction à exécuter peut être composée par une seule ligne de code, comme l'exemple

précédent, ou par un bloc de code contenant plusieurs instructions C. Dans ce dernier cas, l'usage des accolades est obligatoire, afin de marquer le début et la fin du code.

```
if      ( <Condition> )
 {
 instruction_1;
 instruction_2,
 …
 }
```

- **IF … ELSE …**

La structure de sélection peut être étendu par le complément ***else*** sous la forme suivante :

```
 if     ( <Condition> ) <Instruction_1> ;
else    <Instruction_2> ;
```

Exemple :
```
if ( a>b )    c = a – b ;
else          c = b - a ;
```

De nouveau, cette structure peut être étendue avec l'instruction "***else if***" qui permet d'avoir un nombre de choix illimité.

Exemple :
```
if ( a = = b )
      printf("a est égal à b\n");

else if ( a > b )
      printf("a est plus grand que b\n");

else if …
…
else …
```

Si une des conditions est vérifiée, toutes les autres sont sautées. La dernière instruction "***else***" n'est pas obligatoire et peut être utilisée comme choix par défaut.

La condition utilisée par l'instruction ***if*** doit être une **expression logique**. En général, on fait appel à des opérateurs de comparaison :

Opérateur	Signification	Exemple
= =	Egal à	***if*** (a = = b) … ;
!=	Différent de	***if*** (a != b) … ;
>	Supérieur à	***if*** (a > b) … ;
<	Inférieur à	***if*** (a < b) … ;
>=	Supérieur ou égal à	***if*** (a > = b) … ;
<=	Inférieur à	***if*** (a= < b) … ;

L'opérateur "==", qui teste l'égalité, ne doit pas être confondu avec l'opérateur d'affectation "=".

2. Les itérations

Les itérations permettent d'exécuter un bloc d'instructions aussi longtemps qu'une certaine condition est remplie.

- **L'itération *while***

  La syntaxe de cette instruction est la suivante :

  while (**<Condition>**) **<Instruction>** ;

La condition **<Condition>** est une expression logique testée avant l'exécution, et l'instruction **<Instruction>** peut être une seule ligne de code ou un bloc d'instructions.

Exemple :

```
nbr = 0 ;
while (nbr <10)
{
printf("Itération numéro %d \n", nbr) ;
nbr ++ ;
}
```

Il existe des situations où il est nécessaire d'interrompre la boucle sans attendre une nouvelle vérification de la condition : ceci est possible en utilisant l'instruction ***break***. Cette dernière est souvent associée à une condition exprimée avec la sélection ***if***.

Exemple :

```
main()
    {
    char carac ;
    while(1)                    /*** condition toujours vraie ***/
        {
        carac = getchar() ;     /*** renvoi du caractère saisi ***/
        if (carac = = '\n') break ;
        }
    }
```

Les instructions contenues dans cet exemple peuvent être regrouper de la façon suivante, tout en obtenant le même résultat.

```
main()
{
while(getchar() !='\n') ;
}
```

- **L'itération *do…while***

La syntaxe de cette boucle est la suivante :

```
do
{
<instruction> ;
}while ( <condition> ) ;
```

Contrairement à la boucle ***while***, ici la condition est testée après la première exécution de l'instruction, même si elle se révèle fausse.

- **L'itération *for***

L'exécution de l'instruction for nécessite la définition de trois paramètres :

- ✓ la valeur initiale du compteur,
- ✓ la condition à vérifier,
- ✓ l'expression utilisée pour calculer la nouvelle valeur du compteur.

La syntaxe de cette boucle est la suivante :

```
for ( <Initialisation> ; <Condition d'arrêt> ; <Incrémentation> )
{
<Instruction>
}
```

Exemple :
```
for ( nbr = 0 ; nbr <10 ; nbr++ )
printf("Itération numéro %d \n", nbr) ;
```

Le premier et dernier paramètres ne doivent pas forcément initialiser et augmenter (ou diminuer) le compteur, de même que le paramètre intermédiaire, qui ne doit pas forcément être une condition à évaluer. Chacun des trois paramètres peut être une quelconque instruction du langage C.

Exemple :
```
main()
{
for (printf("Tapez une suite de caractères \n : " ) ; getchar() !='\n' ; getchar() ) ;
}
```

L'usage de la boucle ***for*** peut aussi être étendu en incluant plus d'une expression, soit dans le champ des paramètres d'initialisation, soit celui des paramètres d'incrémentation (ou de décrémentation).

Exemple :
```
main ()
{
int  m , n ;
for (m = 1 , n = 15 ; m<= 10 && n >=1 ; m++ , n-- )
printf("%d - %d", m , n ) ;
}
```

De même que dans les autres itérations, il est possible d'interrompre la boucle avant que la condition soit satisfaite par utilisation de l'instruction ***break***.

❑ Les sélections *switch … case …* et *default*

L'instruction ***if*** devient lourd à utiliser si la condition à tester prend plusieurs valeurs possibles. L'instruction ***switch-case-default*** est prévue à cet effet. La syntaxe de cette instruction est la suivante :

```
switch ( <expression> )
{
case <valeur_1> : <instruction_1> ;
case <valeur_1> : <instruction_1> ;
…
case <valeur_n> : <instruction_n> ;
default : <instruction> ;
}
```

L'expression **<expression>** peut être une quelconque expression qui rend une valeur entière (expression entière, un caractère, ou un appel de fonction). Dés l'exécution de l'instruction switch, une comparaison entre l'expression **<expression>** et toutes les valeurs possibles est effectuée. Si la condition d'égalité est vérifiée, le programme exécute l'instruction qui suit le ***case***, et toutes les instructions des autres ***case*** qui suivent, y compris le cas par défaut (***default***). Pour éviter cela, on fait recours à l'instruction ***break***. La syntaxe devient la suivante :

```
switch ( <expression> )
{
case <valeur_1> : <instruction_1> ; break ;
case <valeur_1> : <instruction_1> ; break ;
…
case <valeur_n> : <instruction_n> ; break ;
default : <instruction> ;
}
```

L'utilisation de ***break*** pour le cas par défaut est superflu, puise que c'est le dernier.

Il arrive par fois qu'une instruction doit être exécuter pour un ensemble de valeurs possibles, pour cela il faut regrouper ces valeurs en répétant l'usage du case autant de fois qu'il y a de valeurs.

Exemple :

```
main ()
{
int nbr ;
switch (nbr)
        {
        case 1 :      case 2 :
                printf("je suis dans les choix 1 et 2 \n") ;
                break ;
        case 3 :
                printf("je suis dans les choix 3 \n") ;
                break ;
        default : printf("je suis dans choix par défaut \n") ;
        }
}
```

- **Etiquettes et branchements inconditionnels**

Une étiquette est un identificateur permettant de repérer des instructions dans un programme en vue de branchement au moyen de l'instruction ***goto***. La syntaxe est la suivante :

goto **<étiquette>** ;

L'étiquette **<étiquette>** doit être placée devant l'instruction sous la forme :

<étiquette> : <instruction> ;

L'usage des branchements inconditionnels rend difficile la lecture des programmes et leur compréhension. Il est donc préférable d'utiliser les instructions décrites au paravent.

VII. Les tableaux

Un tableau est un ensemble de données de même type. Les éléments d'un tableau sont indicés et accessibles individuellement.

La déclaration d'un tableau s'effectue comme tout autre variable selon la syntaxe suivante :

<type> <nom_de_variable> [<Nombre_max>] ;

<Nombre_max> est une valeur entière indiquant le nombre maximal des éléments du tableau.

Exemple : ***int*** **tab**[5] ; /* déclaration de la variable **tab** un tableau de 5 nombres entiers */

L'origine des indices est toujours égale à 0, et ne peut être déplacé. Ainsi, on accède aux 5 entiers du tableau **tab** par tab[0], tab[1], tab[2], tab[3] et tab[4].

Un tableau peut avoir plusieurs dimensions. Les tableaux à deux ou plusieurs dimensions sont déclarés en spécifiant le nombre d'éléments pour chaque dimension.

Exemple : ***int*** **tab** [10][20] ; /* déclaration d'un tableau à deux dimensions */

Dans cet exemple, le premier élément est donné par **tab**[0][0] et le dernier élément par **tab**[9][19].

Un tableau monodimensionnel peut être une chaîne de caractères. Dans ce cas, le dernier caractère est le caractère **NULL** dont la codification correspond à « **\0** ». Ce caractère est inséré automatiquement.

Lors de sa déclaration, un tableau peut faire l'objet d'une initialisation. Voici quelques déclarations avec initialisation.

Exemple : ***int*** premiers[4] = { 3,5,7,11} ;
```
int matrice[2][4]={
                    {1,2,3,4}
                    {5,6,7,8,}
                   } ;
char ville[12]= "Montpellier" ;
char adresse[]= "499, rue de la croix verte" ;
```

Variables et arithmétique

I. Déclaration et définition

La partie du programme qui dit au compilateur quel emplacement mémoire est requis et doit être référencé par un certain nom, est appelée **déclaration.** La forme la plus simple de déclaration consiste en un mot décrivant le type de mémoire qui est requis suivi du nom qui sera utilisé pour référencer cet emplacement :

***int* x**, **y**;

Ici, deux emplacements sont réservés du type ***int*** , qui seront référencés comme **x** et **y**.

L'emplacement mémoire de type "***int***" permet de stocker un entier. Ces emplacements mémoire sont souvent appelés variables car il est toujours possible de changer le nombre qui y est stocké. Le nom associé à l'emplacement mémoire est appelé identificateur. On dira **« la variable x »**, ce qui n'est pas tout à fait juste car il vaut mieux dire "*le nombre stocké à l'emplacement dont l'adresse est x*".

Toute variable utilisée dans le programme doit être déclarée. Toute déclaration de variable doit apparaître avant la première utilisation.

Lors d'une déclaration, il est important d'initialiser la variable par une valeur initiale. Dans le cas contraire, une valeur quelconque est affectée à la variable.

II. Conversions implicites et conversions explicites

- **La conversion implicite ou automatique**

Si on regroupe dans une même expression des types de données différents le compilateur procède à une conversion des types selon la règle suivante :

Le type dont le poids est le plus faible est converti au type dont le poids le plus fort. Ainsi un ***char*** en présence d'un ***int*** sera convertit en ***int*** (la valeur correspondante sera alors le code ASCII du caractère correspondant), un ***int*** en présence d'un ***float*** sera convertit en ***float***. Le type dont le poids est le plus fort est le ***double***.

Exemple:

char a = 'A';
int b = 32;
alors a+b sera un ***int*** contenant 97 car le char a été convertit en int.

Il est aussi important de noter que dans une expression d'affectation le compilateur procède à une conversion implicite en changeant le type de la variable à droite au type de la variable à gauche.

Exemple:

int a=12;
float b=a; alors b sera égale à 12.0
int c=b; alors c sera égale a 12 et la valeur décimale est tronquée.

La conversion explicite ou forcée

Nous avons vu que le compilateur procède à une conversion quand cette conversion lui paraît logique. Cependant il est tout à fait possible de procéder à la conversion d'un ***float*** en un ***int*** par exemple.

Cette conversion est dite une conversion forcée ou explicite. Une conversion forcée peut être achevée en utilisant un ***cast***, la syntaxe d'un ***cast*** est la suivante:

(type souhaité) expression

Exemple: (***int***)(5.31/5) retournera un ***int*** (valeur tronquée), en effet 5.31/5 retourne un ***float*** mais puisque un ***cast*** est utilisé avant l'expression le ***float*** sera convertit en ***int***.

III. Entrées et sorties de données

Entrées des données

La fonction ***scanf***()est chargée d'effectuer différentes tâches :

- ✓ déterminer quels touches ont été appuyées,
- ✓ traduire le nombre saisi dans la représentation interne de l'ordinateur,
- ✓ stocker la valeur dans l'emplacement mémoire choisi.

En fait, ***scanf***() lit depuis "l'entrée standard" qui est habituellement le clavier mais peut-être un flux de donnée en provenance d'un fichier.

Chaque appel à la fonction ***scanf***() est fait avec deux arguments séparés par une virgule: ***scanf*** (**<argument_1>**, **<argument_2>**) ;

Exemple : ***int*** x ;
scanf("%d", &x) ;

<argument_1> : "%d" chaîne de caractères définissant les règles à appliquer pour la **conversion** des touches que l'utilisateur a appuyé en une représentation interne à l'ordinateur. Ces règles sont nécessaires car quand un nombre tel que "45" est tapé au clavier, deux caractères '4' et '5' sont transmis à l'ordinateur, il faut donc les regrouper pour former le nombre 45 dans la représentation binaire de l'ordinateur.

"%d" : Le '%' indique que le caractère suivant 'd' correspond à une commande de conversion.

<argument_2> : **&x** c'est l'adresse de l'emplacement mémoire où stocker le nombre lu et converti.

Sorties des données

La syntaxe de la fonction ***printf***() est la suivante :

printf (**<argument_1>**, **<argument_2>**) ;

<argument_1> : Ce paramètre est une **chaîne de format** ou **combinaison d'éléments** à copier directement sur la sortie standard et de **spécifications de conversion.**

<argument_2> : C'est une expression dont la valeur est **convertie d'une représentation interne** à l'ordinateur vers une **représentation externe** conformément à la **spécification de conversion indiquée** dans la chaîne de format.

Différents types de conversion existent, elles **sont toujours** notées par un caractère '%' suivi d'un ou plusieurs caractères.

Exemple : La spécification de conversion "%d" indique une conversion de la représentation interne de l'ordinateur vers une **valeur entière.**

Quand une valeur est convertie de sa représentation interne par la fonction ***printf***(), l'ensemble des positions occupées par la forme externe est appelée un **champs de sortie**. Le nombre de caractères composant ce champs est appelé la **taille** du champs de sortie.

Exemple : La taille du champs de sortie pour une valeur entière peut être définie en insérant entre le caractère '%' et le caractère 'd' un nombre.

printf (" le nombre est : %3d ",x) ;

La taille du champs de sortie ne peut jamais être inférieure à la taille spécifie.

IV. Les opérateurs arithmétiques et d'affectation

Un opérateur est un symbole qui représente une opération exécutable à partir des valeurs des données. Ainsi, une expression est constituée d'un certain nombres d'opérateurs et :

- ✓ des nombres (constantes) et/ou,
- ✓ des identificateurs de variables et/ou,
- ✓ des expressions, auquel cas la valeur de l'expression est utilisée.

Les opérateurs arithmétiques sont les suivants :

Opérateur	Signification
-	Moins unarie
+	Addition
-	Soustraction
*****	Multiplication
/	Division
%	Modulo, le reste d'une division

L'opérateur affectation "=" est utilisé pour affecter une valeur donnée à une variable. L'expression **x = 9 ;** se traduit par : la variable x reçoit la valeur 9.

L'expression **x = x + 1 ;** est exécutée de la façon suivante : on évalue l'expression **x+1**, la valeur trouvée sera stockée dans l'emplacement mémoire désigné par x.

Le langage C dispose d'autres opérateurs qui sont des combinaisons de l'opérateur d'affectation avec des opérateurs arithmétiques ou des opérateurs bit à bit. Les instructions suivantes :

<variable> = **<variable>** **<opérateur>** **<expression>** ;

deviennent :

<variable> **<opérateur>** = **<expression>** ;

Exemple :

x = x +9 ; devient x += 9 ;
x = x * 2 ; devient x *= 2 ;

V. Les opérateurs Logiques bit à bit

Nous distinguons cinq opérateurs logiques manipulant des bits, et qui sont :

Opérateur	Signification
&	Et bit à bit
|	Ou inclusif bit à bit
^	Ou exclusif bit à bit
~	Négation bit à bit
>>	Décalage à droite
<<	Décalage à gauche

Soient var1 et var2 sont deux variables logiques, les tables de vérités des quatre premiers opérateurs sont les suivantes :

var1	var2	var1&var2	var1|var2	var1^var2	~var1
1	1	1	1	1	0
1	0	0	1	0	0
0	1	0	1	0	1
0	0	0	0	1	1

Exemple : opérateurs de décalages

Décalage à gauche :

11011101 << 1 donne 10111010
11011101 << 3 donne 11101000

Décalage à droite :

11011101 >> 1 donne 01101110
11011101 >> 3 donne 00011011

VI. Les opérateurs Logiques

Le type booléen donnant aux variables la valeur vrai ou faux n'existe pas en langage C. Il est remplacé par le type ***int*** permettant de donner la valeur 1 ou 0 correspondant à vrai ou faux à une condition.

Exemple : ***int*** **condition** ;
condition = (a < b) ;
condition prendra la valeur 1 si la variable a et inférieure à la variable b.

Les opérateurs de combinaison d'expressions conditionnelles sont les suivants :

Opérateur	Signification
&&	Et logique, conjonction
||	Ou logique, disjonction
!	Négation

Les tables de vérités des trois opérateurs sont les suivantes :
var1 et var2 sont deux variables logiques.

var1	var2	var1&&var2	var1| |var2	! var1
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Exemple :

if ((x = = y) | | (x = = z)) ***printf***("il y a égalité") ;

else if (x > y) && (x > z)) ***printf***("x est le plus grand des trois nombre") ;

VII. Les opérateurs d'incrémentation et de décrémentation

L'opérateur de d'incrémentation ++ et celui de décrémentation --, ont comme effet d'augmenter ou de décrémenter la variable affectée. La position de ces deux opérateurs peuvent être , soit avant ou après la variable. Dans les deux cas, la variable est incrémentée ou décrémentée.

Si l'opérateur ++ se trouve devant le nombre, l'incrémentation précède l'affectation. Au contraire si l'opérateur ++ se trouve derrière le nombre, l'incrémentation suit l'affectation.

Exemple : ***int*** x = 5 ;
int y ;

L'exécution de l'instruction ***y = x++ ;*** (ou de ***y = x- - ;***) donne comme résultat : y = 5 et x = 6 (y = 5 et x = 4). Tandis que l'exécution de l'instruction ***y = ++x ;*** (ou de ***y = - -x ;***) donne comme résultat : y = 6 et x = 6 (y = 4 et x = 4).

VIII. Priorité et Associativité des opérateurs

Le langage C dispose d'un ensemble de règles qui déterminent la séquence d'exécution des différentes opérations. Il est nécessaire de suivre ces règles concernant la priorité des opérateurs, afin d'aboutir à des expressions correctes. La priorité est relative à l'ordre par lequel le langage C évalue les opérateurs et l'associativité se rapporte à l'ordre par lequel le langage C évalue les opérateurs qui ont la même priorité.

Afin d'aboutir aux résultats attendus, il faut faire usage des parenthèses qui assurent l'ordre dans lequel sera évaluée l'expression par le compilateur. Les parenthèses ont la plus grande priorité.

Le tableau suivant présente les priorités de quelques opérateurs dans le l'ordre décroissant :

()
++ -- ! ~
*/ %
+ -

Exemple :

```
main()
{
int    x=4, y =5 ;
int    u, v ;
u=v=5 + x++ * 10 – (y-=4) ;
}
```

Dans cette expression, les parenthèses possèdent la plus haute priorité, l'affectation étendue est donc exécutée en premier. Le résultat de cette affectation est 1. Par ordre de priorité décroissante, nous avons l'opérateur ++. Comme cet opérateur est placé après la variable, x prendra donc la valeur 5 après l'évaluation de toute l'expression.

Le reste de l'expression est évalué selon les priorités des différents opérateurs. Les valeurs en fin de l'exécution de cette instruction sont : x=5 , y=1, v=44 et u=44 ;

IX. Expressions Conditionnelles

L'opérateur conditionnel ***? :*** s'exécute sur la base de trois opérandes. Sa syntaxe est la suivante :

<condition> ? **<expression_1>** : **<expression_2>**

L'interprétation de cette instruction est la suivante : si la condition est vraie alors la valeur rendue par l'instruction est l'expression **<expression_1>**, sinon la valeur rendue sera l'expression **<expression_2>**.

Exemple :

printf("%c",(c>' ') ? c : '?') ;

Cette instruction imprime à l'écran la valeur affectée à la variable c si telle valeur est majeure à un espace (code ASCII 32), sinon elle imprime un point d'interrogation.

Les Fonctions

I. Introduction

1. Déclaration et définition

La définition d'une fonction demande la spécification des opérations que cette fonction devra effectuer, et les règles par lesquelles elle pourra échanger des informations avec d'autres fonctions. Une fonction est constituée de deux parties : une entête et un corps.

L'entête de la fonction, qui précède toujours le corps, sert à définir les règles d'échange d'information (les paramètres avec lesquelles elle effectue les traitements), et le corps décrit les opérations que la fonction doit exécuter. La syntaxe générale est la suivante :

```
<type> <nom_fonction> ( <liste_des_arguments>)
        {
        …
        }
```

Exemple : ***int*** somme (***int*** debut, ***int*** fin)

Le type **<type>** désigne le type du résultat retourné par la fonction. Le nom de la fonction **<nom_fonction>** est un identificateur. Même si la liste des arguments est vide, la présence des parenthèses est obligatoire.

Le corps de la fonction somme qui calcule la somme des nombres compris entre debut et fin est le suivant :

```
int  somme (int debut, int fin)
{
int i, som =0;        /* se sont les variables locales de la fonction */
for (i = debut  ; i <= fin ; ++i) som+=i ;
return (som) ;        /* valeur à renvoyer */
}
```

Le résultat de la fonction est retourné par le l'instruction ***return***;

Une fonction peut ne pas renvoyer de valeur, il faut pour cela introduire le nouveau type appelé ***void*** dans sa déclaration. Ce type permet de ne pas réserver de place en mémoire.

Exemple :

```
char nom[30] ;
void message()
{
printf("Bonjour, \n") ;
printf("Donner votre nom :") ;
}
main()
{
message() ;          /* appel de la fonction*/
scanf("%s", nom) ;
printf("Vous vous appelez %s\n", nom) ;
getch() ;
}
```

Les fonctions doivent obligatoirement être définies à l'extérieur de toute autre fonction. Aucune imbrication de déclaration ne peut avoir lieu.

Les fonctions peuvent faire l'objet d'une déclaration de leurs entête, et être appelées, puis vient la déclaration de leurs corps. L'exemple précédent pouvait être réécrit de la façon suivante :

```
char nom[30] ;
void message() ;
main()
{
message() ;
scanf("%s", nom) ;
printf("Vous vous appelez %s\n", nom) ;
getch() ;
}
void message()
{
printf("Bonjour, \n") ;
printf("Donner votre nom :") ;
}
```

2. Passage d'information entre fonctions

Les fonctions en langage C sont complètement indépendante les unes des autres. Il est donc possible d'effectuer un passage d'informations entre les fonctions. Ce passage d'informations s'effectue par les paramètres d'entrée, et par la valeur à la sortie.

Le programme suivant utilisant l'exemple vu au paragraphe précédent illustre bien ce passage d'information.

Exemple :

```
main()
{
int val1, val2, total ;
printf("Entrez deux nombres entiers :");
scanf("%d%d",&val1,&val2);
total=somme(val1, val2);
printf("la somme de %d …à …%d = %d\n",val1, val2, total);
}
```

Les paramètres envoyés à la fonction lors de son appel, val1 et val2, sont appelés paramètres actuels, et doivent correspondent avec la position et le type des paramètres formels de la fonction, et qui sont debut et fin. Ici, val1 correspond à debut, et val2 correspond à fin.

II. Les Classes d'allocations

La classe d'allocation d'une variable définit en général la façon dont la variable doit interagir avec les différents **modules** du programme. En effet, les variables ne sont pas seulement définies par leur type, mais aussi par la classe d'allocation.

Les classes d'allocations ont une attribution par défaut ; si la classe n'est pas spécifiée dans l'entête déclaratif, le compilateur choisit comme classe celle déductible par le contexte. La classe d'appartenance d'une variable, détermine son **emplacement** dans la mémoire centrale ou dans un registre, et définie en outre sa **durée de vie** et sa **visibilité** par les différentes fonctions.

1. La classe *automatic*

La déclaration d'une variable avec la classe ***automatic*** s'effectue par la mot clé ***auto***.

Exemple : ***auto int*** d ;
auto char carac ;
auto float x ;

Si une variable est déclarée dans un bloc d'instructions (en général au début d'une fonction), et si aucune classe n'est indiquée, alors le compilateur donne par défaut la classe ***automatic***.

L'usage d'une variable de classe ***auto*** est limité dans le bloc de code (ou fonction) où elle est présente. Cette variable sera présente et active dans tous les sous blocs imbriqués. En revanche, quand l'exécution du bloc (ou fonction) où elle se trouve est terminée, le contenu de la variable est effacé de la mémoire.

2. La classe *register*

Les variables qui appartiennent à la classe ***register*** peuvent être utilisées comme celles de la classe ***automatic***. La différence entre une variable déclarée comme ***register*** (registre), et une autre en classe ***automatic***, relève de la façon par laquelle le compilateur mémorise la variable.

En effet, pour les variables de classe ***register***, le compilateur essayera de les stocker dans un registre de l'ordinateur. Ainsi l'accès à ces variables sera plus rapide.

L'usage de cette classe d'allocations nécessite la profonde connaissance de l'environnement de travail, car le nombre de registres ainsi que leur dimension varient pour chaque type d'ordinateurs. Une utilisation intelligente de cette classe d'allocation peut augmenter la vitesse d'exécution des programmes.

3. La classe *static*

Les variables de classe ***static*** sont comme celles des variables de classe ***automatic***. Elles sont locales par rapport à la fonction où elles ont été déclarées. Mais la différence consiste dans le fait que les variables ***static*** une fois déclarées et initialisées, restent actives en dehors de toute fonction, mais accessibles uniquement là où elles ont été déclarées. La syntaxe est la suivante :

static int a ;
static float x=5.23 ;

L'exemple suivant illustre la différence entre des variables de classe ***automatic*** et celles de la classe ***static***.

```
ajout()
{
        int x = 0 ;
        x += 5 ;
        printf("x= %d \n", x) ;
}
main()
{
        ajout() ;
        ajout() ;
}
```

La variable locale x est déclarée en classe ***automatic***. Le résultat affiché à l'écran est x=5 puis x=5.

```
ajout()
{
        static int x = 0 ;
        x += 5 ;
        printf("x= %d \n", x) ;
}
main()
{
        ajout() ;
        ajout() ;
}
```

Ici, la variable locale x est déclarée en classe ***static***. Le résultat affiché à l'écran est x=5 puis x=10.

4. La classe external

Les variables de classe ***external*** sont des variables globales, déclarées hors de toute fonction. Elles sont toujours disponibles et accessibles, et leurs valeurs sont persistantes.

S'il y a un conflit entre les noms des variables globales et les variables locales, ce sont ces dernières qui l'emportent.

Exemple :

```
int  x = 15 ;
main()
{
        int x = 26 ;
        printf("x= %d \n", x) ;
}
```

Le résultat de l'affichage à l'écran est x=26.

Il existe certaines règles quant à l'usage de la déclaration des variables de classe ***external*** : chaque fonction utilisant une variable ***external***, celle-ci peut être précédée du mot clé ***extern***. L'exemple précédent peut être réécrit de la manière suivante :

```
int  x = 15 ;
main()
{
        extern int x ;
        printf("x= %d \n", x) ;
}
```

Le résultat de l'affichage à l'écran est x=15.

III. Structures de blocs

Un bloc de code est un ensemble de déclarations et d'instructions délimité par des accolades. L'exemple suivant illustre l'utilisation des blocs et des fonctions qui peuvent remplacer les blocs pour une meilleure lisibilité.

Exemple :
```
main()
{
        char c = 'A' ;
                {
                char c = 'B' ;
                        {
                                char c = 'C' ;
                                printf("%c \n ", c) ;
                        }
                printf("%c \n ", c) ;
                }
        printf("%c \n", c) ;
}
```

Le même exemple peut être écrit en utilisant des fonctions :

```
affiche2() ;
{
        char c = 'C' ;
        printf("%c \n", c) ;
}
affiche1() ;
{
        char c = 'B' ;
        affiche2() ;
        printf("%c \n", c) ;
}
main()
{
        char c = 'A' ;
        affiche1() ;
        printf(" %c \n", c) ;
}
```

IV. Inclusion de fichiers

L'instruction du **préprocesseur** qui permet l'inclusion de fichiers a la forme suivante:

#include « nom_fichier »

Cette déclaration comporte l'insertion du fichier, dont le nom est spécifié entre parenthèses, à la position même de la ligne de commande.

Un fichier de ce genre peut être une déclaration de constantes et de variables que nous désirons inclure par la suite, ceci pour limiter la dimension des programmes, et pour rendre le code plus facilement lisible.

Certains fichiers, que l'on appelle plus communément fichiers de ***include***, avec une extension "**.h**", contiennent les déclarations des constantes et des variables qui nous permettent d'utiliser la plupart des fonctions du langage C ; par exemple toutes les fonctions d'entrée-sortie demandent l'inclusion de la bibliothèque <***stdio.h***>.

V. compilation conditionnelle

Il est possible que le compilateur, en suivant des directives spécifiées par le programmeur, saute la compilation de certaines partie du code. Ceci peut être réalisé au moyen des commandes du préprocesseur :

#ifdef <nom_macro>
…
bloc de code
…
#endif

Si le nom de la macro a été défini, le bloc de code est élaboré, autrement il n'est considéré ni par le préprocesseur ni par le programme. L'exemple suivant illustre l'utilisation de la compilation conditionnelle.

#ifdef UNIX
…
code à éxécuter pour l'environnement UNIX
…
#else
…
code à exécuter pour l'environnement DOS
…
#endif

La macro UNIX est déclarée de la façon suivante : ***#define*** UNIX
Si cette déclaration est présente alors il y aura compilation du bloc de code UNIX, sinon c'est le bloc de code DOS.

Une autre variante de la condition existe : ***#ifndef …#endif***. L'instruction ***#ifndef*** teste si le nom de la macro n'est pas définie.

VI. La récursion

La récursion a lieu lorsqu'une fonction s'appelle elle-même. La récursion est une technique de programmation souvent employée en C.

Un simple exemple de programmation récursive est celui qui calcule le factoriel d'un chiffe. Le factoriel d'un chiffre n, représenté par le symbole : n! (n factoriel), est exprimé par une série de multiplications répétitives : n ! = n*(n-1)*(n-2)*…*3*2*1.

```
main()
{
      int i = 0 ;
      while (++i < 6) printf("%d ! est %d \n", i, factoriel(i)) ;
      getch() ;
}
int factoriel(int x)
{
if (x = = 1) return (1) ;
else
return  (x* factoriel (x-1)) ;
}
```

Tableaux et Pointeurs

I. Les pointeurs

Une caractéristique du langage C est l'utilisation des pointeurs. Un pointeur est une variable qui contient l'adresse d'une autre variable.

Nous avons vu qu'une variable est définie par son nom et son type. L'exemple suivant déclare la variable entière x et l'initialise à la valeur 5.

int x = 5 ;

Le compilateur réserve un espace mémoire équivalent à la taille d'un entier et affecte le nom x à cet espace. **L'adresse** de cet espace mémoire où est stockée la variable x peut affecté à une variable appelé pointeur. L'adresse de x s'écrit alors **&**x. L'opérateur unaire & permet de définir l'adresse d'une variable.

Connaissant l'adresse d'une variable on obtient son **contenu** à l'aide de l'opérateur unaire ***** qui ne s'applique qu'aux pointeurs. La déclaration d'un pointeur est la suivante :

<type> * **<nom_du_pointeur>** ;

<nom_du_pointeur> est un identificateur.

Exemple :

```
int x = 5;
int *ptr ;          /*** déclaration d'un pointeur à une valeur entière ***/
ptr = &x ;          /*** affectation de l'adresse de la variable x au pointeur ptr ***/
*ptr = 10 ;         /*** affectation de la valeur 10 à x puisque x et ptr accèdent***
                    *** aux mêmes emplacements mémoires ***/
```

II. Les tableaux et Les pointeurs

Les éléments d'un tableau sont stockés dans des emplacements mémoires contiguës. Le nom du tableau cité tout seul donne l'adresse de base où est stocké ce tableau. Soit la déclaration du tableau d'entiers suivant :

int tab [10] ;

Le langage C traite le mot tab comme un pointeur dirigé sur le 1er élément du tableau. Et donc tab n'est autre que &(tab[0]). Par conséquent, on peut accéder à la valeur du premier élément par l'expression (*tab).

Exemple :

```
main()
{
    int  i  = 0 ;
    int tab [] = {1,2,3,4,5} ;
    while ( i< 5 )
    {
printf ("La valeur de l'élément, %d, est %d, stocké à l'adresse %u \n", i , *(tab+i), tab+i) ;
    }
}
```

L'exécution donne les valeurs stockées dans le tableau suivies par leurs adresses en mémoire.

La déclaration d'une chaîne de caractères revient à déclarer un tableau dont les éléments sont des caractères. Soit donc la déclaration suivante :

char mot[]= "ville" , *ptr = mot;

Le premier caractère est mémorisé à l'adresse (mot+0). Pour accéder à un caractère, par exemple 'i', on peut utiliser soit l'expression *(mot+1) soit l'expression mot[1] . Avec le pointeur, il est possible aussi d'utiliser un indice, le même caractère est obtenu par les deux expressions *(ptr+1) et ptr[1].

Les pointeurs contiennent des adresses et ces dernières sont des nombres sur lesquels peuvent être effectués des calculs.

Exemple :

```
int tab [5], *ptr ;
ptr = tab ;
++ptr ;      /*** ptr pointe sur l'élément tab[1] ***/
ptr +=3 ;    /*** ptr pointe sur l'élément tab[4] ***/
```

III. Tableaux multidimensionnels

Comme nous l'avons dit précédemment, cela n'est pas directement possible en C de déclarer des tableaux multidimensionnels. Mais il existe un moyen détourné pour palier le problème : il suffit de définir un tableau de tableaux d'un type quelconque. Par exemple, si vous avez à créer un tableau de 2*3 cellules, il vous faut alors déclarer un tableau de deux entrés. Puis, pour chacune de ces entrées il faudra déclarer un tableau à trois entrées. Une telle définition se fait de manière compacte, comme le montre l'exemple suivant :

```
int      tab[3][2] ;
/* Un tableau à deux entrées contenant chacune un autre tableau à trois entrées */
```

En terme de références, cela se traduit par un pointeur sur un autre pointeur qui lui pointe sur le premier élément de la première ligne.

IV. Mode de passage des paramètres

En théorie des langages, il existe plusieurs modes de passage de paramètres au travers d'une fonction. En absolu, C en propose un unique : le passage par valeur. Cependant, l'utilisation de variables pointées permet de réaliser un passage par référence. Pour comprendre les choses regardons plus en détail chacun de ces deux modes de passage de paramètres.

1. Le passage par valeur

Ce mode de passage fonctionne en établissant une copie exacte de la valeur qui doit être passée en paramètre. Ceci implique un résultat important : le contenu d'une fonction ne peut

en aucun cas changer la valeur d'une variable passée en paramètre. Cela est logique car l'on en réalise une copie.

Exemple :

```
void essai(int valeur)
{
        valeur++;
}
void main()
{
        int a=10;
        essai(a);
        printf("La valeur de a vaut : %d\n", a);
}
```

Le résultat en sortie est : `La valeur de a vaut : 10.`

2. Le passage par adresse (ou par référence)

Contrairement au passage par valeur, le passage par adresse, n'effectue pas de copie du paramètre : il passe une référence sur celui-ci. En conséquence, une fonction passant des paramètres par référence ne pourra jamais prendre (en paramètre) une constante, mais uniquement une variable. Ce mode de passage peut donc être intéressant, surtout si un paramètre est important en terme de taille (un tableau, une structure, ...).

Comme dit précédemment, le langage C ne propose que le passage par valeur. Cependant, il propose aussi, la possibilité d'utiliser des pointeurs. En conséquence, on arrive, indirectement, à effectuer un passage par référence, via la notion de pointeurs. Dans tout les cas, la valeur du pointeur (une adresse mémoire) sera, elle, passée par valeur.

Exemple :

```
void essai(int *p)
{
  *p+=2;      /* On change la valeur pointée */
  p++;        /* On change le pointeur */
  *p+=3;      /* On change la nouvelle valeur pointée */
}

void main()
{
  int tab[2] = { 10,11 }, *p=&tab;
  essai(p);
  printf("tab : [%d,%d]\n", *p, *(p+1));
}
```

Le résultat en sortie est : `tab : [12,14].`

V. Allocation et libération dynamique de mémoire : fonctions malloc() et free()

Elle permet d'indiquer la taille de l'emplacement à réserver en mémoire. La syntaxe générale de cette fonction est la suivante :

```
#include <stdlib.h>
void *malloc (size_t taille ) ;
```

Taille correspond au nombre d'octets à allouer. La fonction ***malloc***() renvoie un pointeur ***void***. Cela permet de convertir automatiquement au type du pointeur à gauche de l'opérateur d'affectation.

Si l'affectation de mémoire échoue, la fonction ***malloc***() renvoie un pointeur nul. Cela survient quand le programme ne dispose pas d'un espace mémoire suffisant. C'est pourquoi if faut vérifier le pointeur de retour de la fonction ***malloc***().

La mémoire n'est pas une ressource inépuisable, il est important de libérer tout espace mémoire alloué après utilisation. L'utilisation de la fonction free() permet de libérer l'espace mémoire alloué. La syntaxe est la suivante :

```
void        free( void *ptr_adresse) ;
```

Exemple :

```
int     *ptr ;
ptr = malloc (10 * sizeof (int)) ;
if (ptr !== NULL)       {
        printf ("Allocation réussie") ;
        free(ptr) ;
        }
else
{
printf("Echec d'allocation de mémoire") ;
}
```

L'opérateur ***sizeof*** permet de connaître la taille (en octets) d'un type donnée. Ceci est particulièrement intéressant pour écrire des programmes portables d'une machine à une autre, en particulier lorsqu'on réalise des allocations dynamiques d'espaces mémoire.

Pointeurs et Fonctions

I. Pointeurs et Fonctions

Tous les types de variables disponibles dans le langage C, exception faite pour les variables de type **"register"** (dont les valeurs sont contenues dans les registres de l'ordinateur), sont associés à une adresse qui identifie la position en mémoire dans laquelle leur valeurs sont mémorisées. Ainsi, le langage C, au moyen des pointeurs, permet d'accéder à toutes ces variables.

Rappelons que ce qui distingue une fonction dans le langage C est la présence des deux parenthèses. Chaque fois que le compilateur rencontre un nom symbolique suivi de parenthèses, il lui associe la signification d'une fonction. Par la suite, chaque citation du nom de la fonction sans parenthèses est interprétée par le compilateur comme une référence à l'adresse de la fonction.

Il est donc possible en C de faire appel à la fonction au moyen d'un pointeur. L'exemple suivant illustre l'utilisation de pointeurs de fonctions.

```
char *message ()        /*** fonction qui renvoie un pointeur de type char ***/

main()
{
        char *(*ptr_message)() ;
 /*** pointeur à une fonction qui renvoie un pointeur de type char ***/

        ptr_message = message ;
 /*** affectation de l'adresse de la fonction au pointeur ***/

        printf(" %s ", message()) ;
        printf(" %s ",(*ptr_message)()) ;
}

char * message()
{
        return ("Bonjour") ;
}
```

Les deux instructions d'affichage, affichent à l'écran le même message "Bonjour". La première instruction est réalisée par l'appel de la fonction à partir de son nom. Quant à la deuxième instruction, elle est réalisée par l'appel du pointeur de fonction.

II. Tableaux de pointeurs

Les pointeurs sont des variables, il est alors possible de déclarer un tableau de pointeurs. Dans l'exemple suivant, on déclare un tableau de pointeurs sur les jours de la semaine.

Exemple :

```
char *semaine[] = {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche" } ;
```

Chaque nom de jour est une chaîne de caractères terminée par le caractère nul ‘\0’.

L’accès aux éléments du tableau se fait de la manière suivante : semaine [0] pointe sur "lundi", semaine [4] pointe "vendredi".

III. Tableaux de fonctions

Le concept de pointeur de fonction est prévu dans le langage C pour des programmes qui ne peuvent pas connaître à priori quelle fonction devra être appelée, parce que l’exécution de telle fonction dépend du contexte du programme même. L’exemple typique est celui d’un programme qui prévoit le choix de la fonction à appeler sur la base des valeurs en entrées.

Dans ce cas, tous les pointeurs des fonctions susceptibles d’être utilisées sont stockés dans un tableau pouvant utiliser les données en entrées pour décider quelle fonction appeler.

Types de données structurées

I. Les structures

Pour regrouper des données appartenant à différents types de données, et pouvoir y accéder comme à une entité unique, nous disposons dans le langage C d'un type de donnée appelé : **structure**. La déclaration d'une structure est faite suivant la syntaxe suivante :

```
struct <nom_de_structure>
{
        <type>    <membre_1> ;
        <type>    <membre_2> ;
        <type>    <membre_3> ;
        …
} ;
```

Exemple :

```
struct personne
{
        char nom[20] ;
        char adresse[30] ;
        int age ;
} ;
```

Les champs ou membres de la structure **personne** sont **nom**, **adresse** et **age**.

La déclaration d'une structure n'occupe pas de place en mémoire parce qu'il s'agit seulement d'un modèle qui décrit les caractéristiques de la structure. Une fois la structure définie, il est possible de déclarer des variables de type structure.

Pour l'exemple précédent, la déclaration de variables de type structure de personne a la syntaxe suivante :

```
struct  personne  per1, per2, per3 ;
```

L'accès aux membres de la structure est réalisé de la manière suivante :

```
per1.nom = « dupond » ;
per2.adresse = « Paris » ;
per3.age = 30 ;
```

La déclaration de ces trois variables peut être fait dès la définition de la structure de la façon suivante :

```
struct personne
{
        char nom[20] ;
        char adresse[30] ;
        int age ;
} per1, per2, per3;
```

```
struct
{
        char nom[20] ;
        char adresse[30] ;
        int age ;
} per1, per2, per3;
```

Dans ce cas, il n'est pas nécessaire de spécifier le nom la structure.

II. Les tableaux de structures

Dans l'exemple précédent, nous avons déclaré trois variables de type structure. Ces trois variables peuvent être déclarées comme éléments d'un tableau. La nouvelle variable sera un tableau de structures.

Exemple :

```
struct personne
{
        char nom[20] ;
        char adresse[30] ;
        int age ;
} ;
main ()
{
struct personne tab[3] ;
        tab[0].nom = "Dupond" ;
        tab[1].adresse = "Paris" ;
        tab[2].age = 30 ;
}
```

III. Pointeurs sur structures

La déclaration d'un pointeur de structure est la suivante :

struct <nom_de_structure> *<nom_pointeur> ;

L'utilisation des pointeurs et l'accès aux membres de la structure sont illustrés dans l'exemple suivant :

```
struct personne
{
        char nom[20] ;
        char adresse[30] ;
        int age ;
} ;
main ()
{
struct personne tab[3] , *ptr;

        tab[0].nom = "Pierre" ;
        tab[0].adresse = "Paris" ;
        tab[0].age = 20 ;

        tab[1].nom = "François" ;
        tab[1].adresse = "Montpellier" ;
        tab[1].age = 25 ;
```

```
        tab[2].nom = "Claude" ;
        tab[2].adresse = "Limoges" ;
        tab[2].age = 30 ;
        printf ("\n\t nom\t adresse\t age  \n") ;
        for (ptr = tab ; ptr < =tab+2 ; ++ptr)
        printf("\t %s \t %s \t %d \n", ptr->nom, ptr->adresse,ptr->age) ;
        getch() ;
}
```

L'affichage à l'écran est le suivant :

```
Pierre      Paris           20
François    Montpellier     25
Claude      Limoges         30
```

IV. L'opérateur typedef

L'instruction ***typedef*** permet de définir de nouveaux types de données. Ainsi, l'utilisateur peut définir ses propres types.

Exemple :

```
typedef     char        caractere ;
typedef     char        chaine[20] ;
typedef     int         entier ;

caractere   c = 'r' ;
chaine      nom, prenom ;
entier      age ;
```

On peut également définir des types énumérés ou des structures par le mot clé ***typedef***.

Exemple :

```
typedef     enum        {LUN, MAR, MER, JEU, VEN, SAM, DIM} MOIS ;

typedef     struct      personne
            {
                char nom[20] ;
                char adresse[30] ;
                int age ;
            } PERSONNE;
```

Les déclarations des variables peuvent être effectuées de la manière suivante :

```
MOIS        var_mois ;
struct      personne    per1, per2, per3 ;

PERSONNE                per1, per2, per3 ;
```

V. Les unions

L'union permet de regrouper plusieurs éléments de natures diverses sous un même nom logique, mais un seul de ces composants sera actif à un moment donné. Tous les composants de l'union sont stockés dans le même espace mémoire, et débutent à la même adresse. Le compilateur réserve la taille correspondant au plus long des éléments.

Cela permet d'économiser de la place en partageant la mémoire entre différents données. L'affectation d'une valeur à un élément entraîne la destruction de l'ancienne valeur, éventuellement d'un autre composant. Une variable de type union ne peut être initialisée à sa déclaration.

Exemple :

```
union  exemple_union
{
        char carac [ 3 ] ;
        int ent ;
} ;
```

La déclaration d'une variable s'effectue de la manière suivante :

```
union exemple_union  mot ;

mot.ent = 546 ;

mot.carac [ 0 ] = 'e' ;
mot.carac [ 1 ] = 'u' ;
mot.carac [ 2 ] = 'i' ;
```

La dernière affectation écrase la valeur entière 546.

Les Fichiers de Données

I. Introdcution

Les données stockées en mémoire sont perdues dès la sortie du programme. Les fichiers sur support magnétique (bande, disquette, disque) sont par contre conservables, mais au prix d'un temps d'accès aux données très supérieur. On peut distinguer les fichiers séquentiels (on accède au contenu dans l'ordre du stockage) ou à accès direct (on peut directement accéder à n'importe quel endroit du fichier). Les fichiers sont soit binaires (un *float* sera stocké comme il est codé en mémoire , d'où gain de place mais incompatibilité entre logiciels), soit formaté ASCII (un *float* binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre). Les fichiers étant dépendants du matériel, ils ne sont pas prévus dans la syntaxe du C mais par l'intermédiaire de fonctions spécifiques.

II. Type de fichiers

1. Fichiers binaires

C'est la méthode la plus efficace et rapide pour stocker et récupérer des données sur fichier (mais aussi la moins pratique). On accède au fichier par lecture ou écriture de blocs (groupe d'octets de taille définie par le programmeur). C'est au programmeur de préparer et gérer ses blocs. On choisira en général une taille de bloc constante pour tout le fichier, et correspondant à la taille d'un enregistrement physique (secteur, cluster...). On traite les fichiers par l'intermédiaire de fonctions, prototypées dans stdio.h (ouverture et fermeture) et dans io.h (les autres), disponibles sur la plupart des compilateurs (DOS, UNIX) mais pas standardisés.

La première opération à effectuer est d'ouvrir le fichier. Ceci consiste à définir le nom du fichier (comment il s'appelle sous le système) et comment on veut l'utiliser. On appelle pour cela la fonction :

```
int open(char *nomfic, int mode);
```

nomfic pointe sur le nom du fichier (pouvant contenir un chemin d'accès). Mode permet de définir comment on utilisera le fichier. On utilise pour cela des constantes définies dans fcntl.h :
O_RDONLY lecture seule, O_WRONLY écriture seule, O_RDWR lecture et écriture. On peut combiner cet accès avec d'autres spécifications, par une opération OU (|) :
O_APPEND positionnement en fin de fichier (permet d'augmenter le fichier), O_CREAT crée le fichier s'il n'existe pas, au lieu de donner une erreur, sans effet s'il existe (rajouter en 3ème argument S_IREAD | S_IWRITE | S_IEXEC déclarés dans stat.h pour être compatible UNIX et créer un fichier lecture/ écriture/ exécution autorisée, seul S_IWRITE utile sur PC), O_TRUNC vide le fichier s'il existait, O_EXCL renvoie une erreur si fichier existant (utilisé avec O_CREAT).
Deux modes spécifiques au PC sont disponibles : O_TEXT change tous les \n en paire CR/LF et inversement, O_BINARY n'effectue aucune transformation.
La fonction rend un entier positif dont on se servira par la suite pour accéder au fichier (HANDLE), ou -1 en cas d'erreur. Dans ce cas, le type d'erreur est donné dans la variable errno, détaillée dans errno.h.

On peut ensuite, suivant le mode d'ouverture, soit lire soit écrire un bloc (l'opération est alors directement effectuée sur disque) :

int write(int handle, void *bloc, unsigned taille);

On désigne le fichier destination par son handle (celui rendu par open), l'adresse du bloc à écrire et la taille (en octets) de ce bloc. Le nombre d'octets écrits est retourné, -1 si erreur.

int read(int handle, void *bloc, unsigned taille);

lit dans le fichier désigné par son handle, et le met dans le bloc dont on donne l'adresse et la taille. La fonction retourne le nombre d'octets lus (<=taille, <si fin du fichier en cours de lecture, 0 si on était déjà sur la fin du fichier, -1 si erreur).

int eof(int handle)

dit si on se trouve (1) ou non (0) sur la fin du fichier.

Lorsque l'on ne se sert plus du fichier, il faut le fermer (obligatoire pour que le fichier soit utilisable par le système d'exploitation, entre autre mise à jour de sa taille :

int close(int handle)

fermeture, rend 0 si ok, -1 si erreur.

Le fichier peut être utilisé séquentiellement (le "pointeur de fichier" est toujours placé derrière le bloc que l'on vient de traiter, pour pouvoir traiter le suivant). Pour déplacer le pointeur de fichier en n'importe que autre endroit, on appelle la fonction :

long lseek(int handle, long combien, int code);

déplace le pointeur de fichier de combien octets, à partir de : début du fichier si code=0, position actuelle si 0, fin du fichier si 2. La fonction retourne la position atteinte (en nb d'octets), -1L si erreur.

long filelength(int handle);

rend la taille d'un fichier (sans déplacer le pointeur de fichier).

2. Fichiers bufférisés

Les opérations d'entrée / sortie sur ces fichiers se font par l'intermédiaire d'un "buffer" (bloc en mémoire) géré automatiquement. Ceci signifie qu'une instruction d'écriture n'impliquera pas une écriture physique sur le disque mais dans le buffer, avec écriture sur disque uniquement quand le buffer est plein.

Les fichiers sont identifiés non par un entier mais par un pointeur sur une structure FILE (définie par un typedef dans stdio.h). Les fonctions disponibles sont prototypées dans stdio.h.

III. Opérations sur les fichiers

1. Ouverture et fermeture d'un fichier

a. Ouverture d'un fichier (fopen) :

- **Syntaxe :**

```
#include <stdio.h>

FILE *fopen(const char *nom , const char *mode);
```

- **Arguments :**
  nom : nom (ou chemin) externe du fichier à ouvrir,
  mode : mode d'ouverture.
- **Description :**
  Hormis les flux déjà ouverts, avant de lire ou d'écrire dans un fichier il faut "ouvrir le fichier". Cette opération permet d'associer (si vous en avez le droit) un flux au fichier désigné par son nom physique nom.
- **Mode d'ouverture d'un fichier :**

Mode	Description
r	Ouverture en lecture seule sur un fichier existant
w	Ouverture en écriture seule. Si le fichier existe il est détruit.
a	Ouverture pour écriture à la fin du fichier. Création du fichier s'il n'existe pas
r+	Ouverture en lecture ou écriture sur un fichier existant.
w+	Ouverture en lecture ou écriture. Si le fichier existe il est détruit. Création du fichier s'il n'existe pas.
a+	Ouverture en lecture ou écriture à la fin du fichier. Création du fichier s'il n'existe pas.

Remarques :

- Contrairement à MSDOS, il n'existe pas sous UNIX de différence entre un fichier binaire et un fichier texte. Ainsi, il existe sous MSDOS des modes d'ouvertures rb, wb, ab, r+b, w+b, a+b permettant d'ouvrir des fichiers en mode binaire et des modes d'ouvertures rt, wt, at, r+t, w+t, a+t permettant d'ouvrir des fichiers en mode texte.
- Sur PC, on peut rajouter t ou b au mode pour des fichiers texte (gestion des CR/LF, option par défaut) ou binaires, ou le définir par défaut en donnant à la variable _fmode la valeur O_TEXT ou O_BINARY.
- Le nombre maximum de fichiers ouvrables par un processus est défini par la pseudo-constante **FOPEN_MAX**.

- **Valeur retournée par fopen:**
  Si elle aboutit, cette fonction retourne un pointeur sur le flux qui vient d'être ouvert; sinon, elle retourne *NULL* et positionne la variable *errno*.

Exemple : ouverture en création

```
#include <stdio.h>

main() {
  FILE *stream;

  if ( (stream = fopen("toto","w")) == NULL) {
     perror("fopen");       /* affichage d'un message d'erreur    */
                            /* correspondant a la valeur de errno */
     exit(1);
  }
}
```

b. Fermeture d'un fichier (fclose) :

- **Syntaxe :**
  ```
  #include <stdio.h>

  int fclose(FILE *stream);
  ```
- **Description :**
  cette fonction ferme le fichier stream désigné, après avoir vidé le tampon qui lui est associé.
- **Valeur retournée :**
  Elle retourne 0 si elle réussit, dans le cas contraire elle retourne *EOF* (constante de fin de fichier valant -1) et positionne la variable *errno*.

***Remarque* :** la fin du processus ferme automatiquement tous les fichiers ouverts.

Exemple :

```
#include <stdio.h>

main() {
 FILE *stream;

 fclose(stdin);            /* fermeture de l'entre standard      */
 if ( (stream = fopen("toto","w")) == NULL) {
  perror("fopen");    /* affichage d'un message d'erreur    */
                  /* correspondant a la valeur de errno */
  exit(1);
 }
 fclose(stream);           /* pas de test des erreurs ...        */
}
```

2. Lecture/écriture binaire dans un flux

a. Ecriture binaire dans un flux (fwrite) :

- **Syntaxe :**
  ```
  #include <stdio.h>
  size_t fwrite(void *ptr, size_t taille, size_t n, FILE *stream);
  ```
- **Description :**
  Ecriture dans le flux stream de *n* objets ayant chacun une longueur de *taille* octets et placés dans une zone pointée par *ptr*. *size_t* correspond à un *unsigned int*.
- **Valeur retournée :**
  Elle retourne le nombre d'objets (et non le nombre d'octets) réellement écrits. Le nombre total d'octets écrits est : *n * taille*.
- **Exemple 1 :**
  ```
  #include <stdio.h>
  #include <string.h>   /* strlen() */
  #include <stdlib.h>   /* exit() */
  main() {
   FILE *stream;
   char *msg = "123456789012345";
   int n;
                        /* OUVERTURE EN CREATION */
   if ( (stream = fopen("toto", "w")) == NULL) {
  ```

```
        perror("fopen");      /* affichage d'un message d'erreur    */
                              /* correspondant à la valeur de errno */
        exit(1);
      }
      n = strlen(msg);  /* ECRITURE DANS LE FICHIER */
      if ( fwrite(msg, sizeof(char), n, stream) != n ) {
    /* écriture dans le flux stderr */
        fwrite("Erreur a l'ecriture\n", 20, 1, stderr);
        exit(2);
      }
                          /* FERMETURE DU FLUX ET VIDAGE DU TAMPON */
      fclose(stream);       /* pas de test des erreurs ...          */
    }
```

- **Exemple 2 :**
  - Ecriture d'un entier :

    ```
    int i=10;
    fwrite( &i, sizeof(int), 1, stream);
    ```
  - Ecriture d'un tableau d'entiers :

    ```
    int tab[100];
    fwrite( tab, sizeof(tab[0]), 100, stream);
    ```
  - Ecriture d'une structure et d'un tableau de structures :

    ```
    stuct s_fiche { char nom[20];
                    int  numero;
                  } fiche, tabfiche[100];

    fwrite( &fiche, sizeof(struct s_fiche), 1, stream);
    fwrite( tabfiche, sizeof(struct s_fiche), 100, stream);
    ```

b. Lecture dans un flux (fread) :

- **Syntaxe :**

  ```
  #include <stdio.h>
  size_t fread(void *ptr, size_t taille, size_t n, FILE *stream);
  ```
- **Description :**
  Lecture dans le flux stream de *n* objets ayant chacun une longueur de *taille* octets et rangement de ces éléments dans la zone pointée par *ptr*.
- **Valeur retournée :**
  Cette fonction retourne le nombre d'objets réellement lus, qui peut être inférieur au nombre n. Elle retourne la valeur 0 si la fin du fichier est rencontrée ou s'il y a une erreur de lecture.
- **Exemple :**

```
/* lecture du fichier créé par l'exemple précédent */
#include <stdio.h>
#include <stdlib.h>   /* exit() */

main() {
  FILE *stream;
  char buf[11];
  int nlec;
```

```
 /* ouverture en lecture du fichier créé par l'exemple précédent */
 if ( (stream = fopen("toto", "r")) == NULL) {
    perror("fopen");        /* affichage d'un message d'erreur    */
                            /* correspondant à la valeur de errno */
    exit(1);
 }
 do {
  /* lecture de 10 caractères dans le fichier */
  nlec = fread(buf, sizeof(char), 10, stream);
  if (nlec != 0) {  /* affichage des nlec caractères lus */
     fwrite(buf, sizeof(char), nlec, stdout);
     putchar('\n');
  }
 } while ( nlec == 10 );
 fclose(stream);
}

/*-- résultat de l'exécution ----------------------
1234567890
12345
-------------------------------------------------*/
```

c. Ecriture des tampons (fflush) :

- **Syntaxe :**
  ```
  #include <stdio.h>
  int fflush(FILE *stream);
  ```
- **Description :**
  La mémoire tampon associée à un flux est vidée lorsqu'elle est pleine ou à la fermeture du flux. La fonction *fflush* permet d'écrire sur disque tous les tampons associés au flux *stream*.
- **Valeur retournée :**
  Elle retourne *EOF* (-1) lorsqu'une erreur est détectée et positionne la variable *errno*.
- **Remarque :**
  si *stream* est un pointeur *NULL*, *fflush* vide sur disque les tampons de tous les flux ouverts.
- **Exemple :**

```
#include <stdio.h>
main() {
 FILE *stream = fopen("/tmp/essai", "w+");

 fflush( stdin );  /* vidage des tampons associés au flux stdin */
 fflush( stream );
 return 0;
  }
```

d. Lecture/écriture de caractères dans un flux

➢ *Lecture d'un caractère (fgetc) :*

- **Syntaxe :**
  ```
  #include <stdio.h>
  ```

```
int fgetc(FILE *stream);
int getc(FILE *stream); /*macro identique à la fonction fgetc */
```

- **Description :**
  Lecture d'un caractère depuis le pointeur courant du flux *stream*.
- **Valeur retournée :**
  Retourne le caractère lu ou retourne *EOF* (-1) si la fin du fichier est atteinte.En cas d'erreur de lecture, elle retourne aussi *EOF* et positionne la variable *errno*.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>   /* strlen() */

int main()  {
 FILE *stream;
 char str[] = "123456789012345";
 char ch;

 stream = fopen("/tmp/essai", "w+");
 fwrite(str, strlen(str), 1, stream);

 fseek(stream, 0, SEEK_SET);      /* se repositionne au début du fichier */
 do {
  ch = fgetc(stream); /* lecture caractère par caractère du flux */
  putchar(ch);              /* et affichage sur stdout du caractère lu */
 } while (ch != EOF);
 fclose(stream);
 return 0;
}
```

➢ *Ecriture d'un caractère (fputc) :*

- **Syntaxe :**

```
#include <stdio.h>

int fputc(int ch, FILE *stream);
int putc(int ch , FILE *stream);  /* macro identique à fputc */
```

- **Description :**
  Ecriture du caractère *ch* dans le flux *stream*.
- **Valeur retournée :**
  Retourne le caractère écrit ou retourne *EOF* en cas d'erreur.
- **Exemple :**

```
/*********************************************/
/* PROGRAMME : COPY_CAR_A_CAR.C              */
/*                                           */
/* COPIE D'UN FICHIER CARACTERE PAR CARACTERE */
/*********************************************/
#include <stdio.h>

main() {
  FILE *in , *out;
          /* Ouverture du fichier en lecture */
```

```
  if ( (in = fopen("ESSAI.C" , "r")) == NULL) {
    perror("fopen en lecture");
    exit(1);
  }
            /* Ouverture du fichier en création */
  if ( (out = fopen("TEMPO.$$$" , "w")) == NULL) {
    perror("fopen en création");
    exit(1);
  }
  while (!feof(in)) /* tant que la fin du fichier n'est pas atteinte */
    putc(getc(in) , out);
  fclose(in);
  fclose(out);
}
```

e. Lecture d'une chaîne (fgets) :

- **Syntaxe :**

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
```

- **Description :**
  Lecture de *n* caractères du flux *stream* et place le résultat à l'endroit pointé par *s*. La lecture de la chaîne s'arrête si la fin du fichier est atteinte, ou si un caractère '\n' est lu, ou si n - 1 caractères ont été lus.
- **Valeur retournée :**
  Elle retourne la chaîne et *NULL* en cas de fin de fichier ou d'erreur.
- Remarques :
  - Le caractère d'interligne '\n' (s'il est lu) est conservé dans la chaîne.
  - Un caractère nul ('\0') est placé en fin de la chaîne pointée par s.
- **Exemple :**

```
#include <stdio.h>
#include <string.h>    /* strlen() */
#include <stdlib.h>    /* exit() */

main() {
   FILE *stream;
   char buf[80];

   if ( (stream = fopen("/tmp/essai", "r")) == NULL ) {
     perror("fopen");
     exit(1);
   }

   /* lecture d'une chaine */
   fgets( buf, sizeof(buf), stream);
   printf("%s", buf);
   fclose(stream);
   return 0;
}
```

f. Ecriture d'une chaîne (fputs) :

- **Syntaxe :**
  ```
  #include <stdio.h>

  int fputs(const char *s, FILE *stream);
  ```
- **Description :**
  Ecriture de la chaîne pointée par *s* dans le flux *stream*.
- **Valeur retournée :**
  S'il n'y a pas d'erreur, cette fonction retourne le dernier caractère écrit; sinon, c'est la valeur *EOF* qui est retournée.
- **Exemple :**
  ```
  #include <stdio.h>

  main() {
     fputs("Hello world\n", stdout);
     return 0;
  }
  ```

3. Lecture/écriture formatée dans un flux

a. Ecriture formatée dans un flux (fprintf) :

- **Syntaxe :**
  ```
  #include <stdio.h>

  int fprintf(FILE *stream, const char *format, argument ...);
  ```
- **Description :**
  Ecriture formatée dans le fichier stream.
  argument ... est une liste d'expressions dont les valeurs seront converties suivant les spécifications de la chaîne format avant d'être écrites dans le flux *stream*.
  Cette fonction utilise les mêmes spécifications de format que *printf*.
- **Valeur retournée :**
  fprintf retourne le nombre d'octets écrits ou *EOF* (-1) en cas d'erreur.
- **Exemple :**
  ```
  #include <stdio.h>

  main() {
     FILE *stream;
     int i = 1234;
     char c = 'C';
     float f = 1.234;

     stream = fopen("/tmp/essai", "w");
     fprintf(stream, "%d %c %f", i, c, f);
     fclose(stream);
     return 0;
  }
  ```

b. Lecture formatée dans un flux (fscanf) :

- **Syntaxe :**

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format, pointeur ...);
```

- **Description :**
  Lecture formatée dans le flux *stream*.
  pointeur ... est une liste d'expressions de type pointeur sur des objets qui recevront les valeurs converties suivant les spécifications de la chaîne *format*.
- **Valeur retournée :**
  Cette fonction retourne le nombre de champs d'entrée correctement lus, convertis et mémorisés ou *EOF* en cas d'erreur.
- **Exemple :**

```
#include <stdio.h>

main() {
 FILE *stream;
 int i;
 char c;
 float f;

 /* lecture du fichier cree par l'exemple precedent */
 stream = fopen("/tmp/essai", "r");
 if (fscanf(stream, "%d %c %f", &i, &c, &f) == 3)
  printf("Lecture reussie: %d %c %f\n", i, c, f);
 fclose(stream);
 return 0;
}
```

4. Position du pointeur de fichier

a. Positionnement du pointeur de fichier (fseek) :

- **Syntaxe :**

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int methode);
```

- **Description :**
  Par défaut, les lectures/écritures s'effectuent à partir de la position courante du pointeur de fichier. Ce dernier est incrémenté automatiquement du nombre de caractères lus ou écrits après chaque opération de lecture/écriture (accès séquentiel). Il est possible de modifier la position courante par la fonction *fseek*, permettant ainsi l'accès direct à une information.
  fseek positionne le pointeur de fichier du flux *stream* pour la prochaine opération de lecture/écriture.
- **Arguments :**
  methode :
  - SEEK_SET (0) : Positionnement à *offset* octet(s) du début du fichier.
  - SEEK_CUR (1) : Positionnement à la position courante *offset* octet(s).

  - SEEK_END (2) : Positionnement à la fin du fichier + *offset* octet(s). (*offset* peut-être négatif !)
- **Valeur retournée :**
  Cette fonction retourne 0 si le pointeur a pu être déplacé et une valeur non nulle sinon.
- Remarques :
  - L'appel : *fseek(stream,0L,SEEK_SET);* revient à se positionner au début du fichier, il est équivalent à l'instruction *rewind(stream);*
  - Le positionnement dans un flux ouvert en mode *append* ne sert à rien, car il y a positionnement à la fin du fichier avant chaque écriture .
- **Exemple :**

```
#include <stdio.h>
#include <string.h>   /* strlen() */

int main()  {
 FILE *stream;
 char str[] = "123456789012345";
 char ch;

 stream = fopen("/tmp/essai", "w+");
 fwrite(str, strlen(str), 1, stream);

 /* se repositionne au debut du fichier */
 fseek(stream, 0, SEEK_SET);

 do {
  ch = fgetc(stream); /* lecture caractere par caractere du fichier */
  putch(ch);          /* et affichage sur stdout du caractere lu    */
 } while (ch != EOF);
 fclose(stream);
 return 0;
}
```

b. Position courante du pointeur de fichier (ftell) :

- **Syntaxe :**

```
#include <stdio.h>

long ftell(FILE *stream);
```

- **Description :**
  Cette fonction retourne la position courante du pointeur de fichier ou la valeur *-1L* quand il y a une erreur.
- **Exemple :**

```
/************************************************/
/* FONCTION QUI COMPTE LE NOMBRE DE LIGNE D'UN    */
/* FICHIER TEXTE A PARTIR DE LA POSITION COURANTE */
/************************************************/

int nb_ligne(FILE *f)
{  int nl = 0;
   long position;
```

```
  position = ftell(f);
  while( fgets(ligne, 255, f) )
    ++nl;
  fseek(f, position, 0);  /* retour à la position initiale */
  return nl;
}
```

5. Fonctions diverses de gestion du flux

a. Tester la fin de fichier (feof) :

- **Syntaxe :**

```
#include <stdio.h>

int feof(FILE *stream);
```

- **Description :**
  Cette fonction retourne une valeur non nulle si la fin de fichier désigné par stream est atteinte. Elle permet donc de distinguer une erreur de lecture ou d'écriture d'une fin de fichier.
- **Exemple :**

```
#include <stdio.h>

main() {
 FILE *stream;
 char ch;

 stream = fopen("/tmp/essai", "r");
 ch = fgetc(stream);   /* lecture d'un caractère dans le flux */
 if (feof(stream))     /* teste la fin du fichier */
  printf("fin du fichier atteinte\n");
 fclose(stream);
 return 0;
}
```

b. Gestion des erreurs (ferror et clearerr) :

- **Syntaxe :**

```
#include <stdio.h>

int ferror(FILE *stream);
int clearerr(FILE *stream);
```

- **Description :**
  La fonction *ferror* retourne une valeur non nulle après une erreur sur le fichier stream. L'erreur reste positionnée jusqu'à l'appel de la fonction *clearerr* ou à la fermeture du fichier.
- **Exemple :**

```
#include <stdio.h>
```

```
main() {
  FILE *stream = fopen("/tmp/essai", "w");

  (void) getc(stream);   /* erreur: flux ouvert en ecriture seule */

  if (ferror(stream)) {     /* teste si une erreur c'est produite */
    fprintf(stderr, "Erreur sur le fichier\n");
    clearerr(stream);          /* RAZ de l'erreur et du flag EOF */
  }
  fclose(stream);
  return 0;
}
```

c. Fonctions diverses (remove, rename) :

- **Syntaxe :**

```
#include <stdio.h>

int remove(const char *nom);
int rename(const char *anciennom , const char *nouveaunom);
```

- **Description :**
  La fonction *remove* supprime le fichier désigné. Elle retourne 0 si l'opération a réussi et une valeur non nulle si elle a échoué.
  La fonction *rename* modifie le nom d'un fichier. *anciennom* est remplacé par *nouveaunom*.
- **Valeur retournée :**
  Elles retournent 0 si l'opération a réussi et une valeur non nulle si elle a échoué.
- **Exemple :**

```
#include <stdio.h>

main() {
  char nf[80];

  printf("Fichier a effacer: ");  gets(nf);
  if (remove(nf) == 0)
    printf("Effacement de %s\n", nf);
  else
    perror("remove");
  return 0;
}
```

- **Exemple :**

```
#include <stdio.h>

main(int argc, char *argv[]) {

  if ( argc != 3)
    fprintf(stderr,"Usage: %s ancien_nom nouveau_nom\n", argv[0]);
  else
    if (rename(argv[1], argv[2]) != 0)
```

```
    perror("rename");
  return 0;
}
```

d. Redéfinir un flux (freopen) :

- **Syntaxe :**
  #include <stdio.h>

  FILE *freopen(const char *filename, const char *mode, FILE *stream);

- **Description :**
  La fonction **freopen** permet de changer le fichier associé à un flux déjà ouvert. *freopen* commence par fermer le flux stream. Elle effectue ensuite l'ouverture d'un nouveau flux associé au fichier de nom *filename* et selon le mode décrit par le paramètre *mode*. Si l'ouverture réussit, le nouveau flux est placé dans le descripteur stream.
  Le flux original *stream* est refermé, même si l'ouverture du nouveau fichier échoue.
- **Valeur retournée :**
  En cas de succès *freopen* retourne le flux fourni en argument. Si erreur, la fonction retourne *NULL*.
- **Exemple 1 :** redirection de la sortie standard

```
#include <stdio.h>

main() {

    /* redirection de stdout dans un fichier */
    if ( freopen("tempo.log", "w", stdout) == NULL)
       fprintf(stderr, "erreur a la redirection de stdout\n");

    /* Cet affichage va dans le fichier  tempo.log */
    printf("Cet affichage va dans le fichier\n");

    /* fermeture du flux stdout */
    fclose(stdout);

    return 0;
}
```

Gestion des processus

I. Introduction

Un système tel qu'Unix est un système qui permet d'avoir plusieurs processus en même temps. On donne ainsi à l'utilisateur l'impression d'avoir la possibilité de faire plusieurs choses à la fois. En fait, au niveau machine, si celle-ci n'a qu'un seul processeur, il n'en est rien. Mais on peut ainsi notamment utiliser les temps morts de certaines applications pour les autres. C ayant été créé pour Unix, il doit donc permettre de gérer les processus.

II. Création d'un processus

La création d'un processus en C se fait à partir de la fonction fork. Celle-ci renvoie -1 si elle échoue. Si elle réussit, elle crée un processus *fils* du premier. Celui-ci dispose d'un environnement semblable à celui de son père. Le fils poursuit l'exécution du programme après le fork() tout comme son père. La seule différence est dans le résultat de la commande fork(). Pour le père, on trouve le numéro de processus affecté au fils. Pour le fils, on obtient 0.

Exemple :

```
main()
{
        int t,i;
        t = fork();
        if (t == -1)
                {
                printf("Erreur : creation fils a echouee\n");
                return 1;
                }
        if (t)
                {
        printf("On est dans le processus pere\n");
                printf("Et on vient de creer un fils numero %d\n", t);
                for (i = 0; i <= 200 ; printf("%4d",i++));
                }
        else
                {
        printf("On est dans le processus fils\n");
        printf("cree par un pere numero %d\n",getppid());
                for (i = 400; i <=600 ; printf("%d",i++));
                }
}
```

Dans cet exemple, si le père se termine avant le fils, celui-ci est adopté par le processus *init*, d'identificateur 1. Pour éviter cela, Il faut que le processus père attende la mort de son fils. Cela se fait grâce à la fonction wait par un :

r = wait(&id);

où id est l'identificateur du processus dont ont attend la mort. r est tel que :

- r / 256 = paramètre de exit ;
- r mod 256 = numero signal si processus tué par un signal sans création d'un fichier core ;

- r mod 256 = 128 + n du signal si processus tué par un signal avec création d'un fichier core.

Dans le cas où l'on souhaite qu'un processus lance un autre programme, mais sans que cela soit un autre processus, il faut utiliser les fonction execl ou execv, ou une de leur variante. execl suppose que le nombre de paramètre est connu : on passe le nom du programme en paramètre, puis tous les paramètres du programme sous forme de chaînes de caractères. Dans le cas d' execv, on passe un tableau de chaînes de caractères. Dans les deux cas, le premier paramètre est toujours le nom du programme.

Exemple : lancement du programme prog2 sans paramètre. execl("prog2", "prog2");
Les fonctions execle et execve jouent le même rôle que execl et execv, mais un dernier paramètre est un tableau envp qui permet de passer un environnement.

Enfin, les fonctions execlp et execvp sont similaires à execl et execv, mise à part le fait qu'elles permettent d'exécuter un programme situé dans un répertoire quelconque de la variable d'environnement PATH.

III. Communication entre processus

Si l'on désire avoir plusieurs processus qui collaborent à une tâche donnée, il faut qu'ils puissent communiquer. Il existe plusieurs moyens pour faire communiquer des processus. Ici, seule la communication par *pipes* (tubes) est présentée.

Un pipe est une implantation d'une file. Il a une taille limitée. Un processus qui essaie d'écrire dans un pipe plein est bloqué jusqu'à ce qu'il y ait une place de libérée.

La *production* et la *consommation* au niveau d'un pipe se font par l'intermédiaire de descripteurs de fichier (un par opération), que le processus doit connaître. Un tube peut bien évidemment être fermé par un processus. Dans la mesure où plusieurs processus peuvent partager un même tube à la fois en lecture et en écriture, un tube ne transmettra un EOF (End Of File) que lorsque tous les processus auront fermé le tube en écriture.

La création d'un pipe se fait au moyen de la commande n = pipe(p), où p est un tableau de 2 entiers, et n vaut 0 si l'opération s'est passée sans problème.
L'écriture dans un tube utilise la commande write(p[1], &c, n) où &c est un pointeur sur caractère, et n est le nombre de caractère à transmettre.

La lecture dans un tube se fait par la commande read(p[0], &c, n). Si le tube est vide, elle renvoie zéro. Un processus peut fermer les accès en lecture ou écriture dans un tube au moyen respectivement des commandes close(p[0]) et close(p[1]).
Plutôt que d'utiliser les fonctions read et write pour accéder à un pipe, on peut utiliser les fonctions classiques sur les fichiers telles que fprintf et fscanf. Cela nécessite cependant d'obtenir les pointeurs sur fichier nécessaires au moyen de la fonction fdopen :

```
lecture = fdopen(p[0], "w");
ecriture = fdopen(p[1], "r");
...
fprintf(lecture, "%d ",n);
...
fscanf(ecriture, "%d", &m);
...
fclose(lecture);
fclose(ecriture);
```

Annexes

I. Les arguments de la ligne de commande

1. Introduction

Dans de nombreux systèmes, les programmes sont appelés par l'intermédiaire d'une *ligne de commande*, sur laquelle on tape le nom du programme, auquel on peut associer des paramètres. Exemple : lorsque l'on tape `cp toto1 toto2`, `toto1` et `toto2` sont des paramètres du programme `cp`. Il faut donc que le programme puisse récupérer ses paramètres. Le langage C ayant été conçu notamment dans le but d'écrire Unix et les programmes qui tournent dessus, il a donc été prévu de pouvoir récupérer ces paramètres.

2. Les paramètres argc et argv

La fonction que l'ont déclare généralement sous la forme `main()`, a en fait le profil suivant : `int main (int argc, char *argv[])`, où `argc` est un entier qui indique le nombre de paramètres de la ligne de commande (nom du programme y compris), et `argv` est un tableau de chaîne de caractères, une par argument. À noter cependant que le premier élément du tableau `argv`, d'indice zéro, est le nom de la commande.

Exemple :

```
main(int argc, char *argv[])
{
        int i;
        printf("Vous avez lance le programme : %s ",argv[0]);
        switch (argc)
                {
                case 1 :
                printf("sans argument \n");
                        break;
                case 2 :
                printf("avec l'argument %s \n", argv[1]);
                        break;
                default :

                printf("avec les arguments suivant%s\ n");
                        for ( i = 1; i <= argc ; i++)
                        printf("%s\n", argv[i]);
                }
}
```

II. Fonctions liées aux chaînes de caractères

1. Introduction

Comme on l'a vu, il n'existe pas en C à proprement parler de type *chaîne de caractères*. Ceci fait que le langage n'a pas d'opérateurs où mots-clefs réservés pour le traitement de celles-ci. Cependant ce manque est en partie comblé par des fonctions de la libraire de base, que l'on peut utiliser en incluant le fichier `string.h`.

2 .Quelques fonctions

On se limitera ici aux fonctions les plus utiles :

- char ***strcpy**(char *dest, char *src) : Copie la chaîne *src* dans la chaîne *dest*, y compris le caractère nul de fin de chaîne. De plus, cette fonction retourne *dest*. L'allocation pour *dest* doit avoir été faite avant.
- char ***strdup**(char *src) : même rôle que strcpy, mais fait l'allocation mémoire.
- char ***strncpy**(char *dest, char *src, size_t n) : comme strcpy, mais copie au plus *n* caractères. Complète par des caractères nuls si la chaîne n'est pas assez longue.
- size_t ***strlen**(char *src) : renvoie la longueur de la chaîne *src*.
- char ***strcat**(char *dest, char *src) : rajoute la chaîne *src* à la chaîne *dest*, et retourne cette dernière. N.B. : il existe aussi strncat.
- int ***strcmp**(char *ch1, char *ch2) : compare les chaînes *ch1* et *ch2* en utilisant l'ordre alphabétique. Renvoie un nombre négatif si **ch1 < ch2**, un nombre positif si **ch1 > ch2**, et zéro si **ch1 = ch2**. N.B. : il existe aussi strncmp.
- char ***strcstr**(char *ch, char *ss_ch) : retourne un pointeur sur la première occurence de *ss_ch* dans *ch*. Si elle n'est pas présente, renvoie le pointeur NULL.

III. La programmation modulaire en C

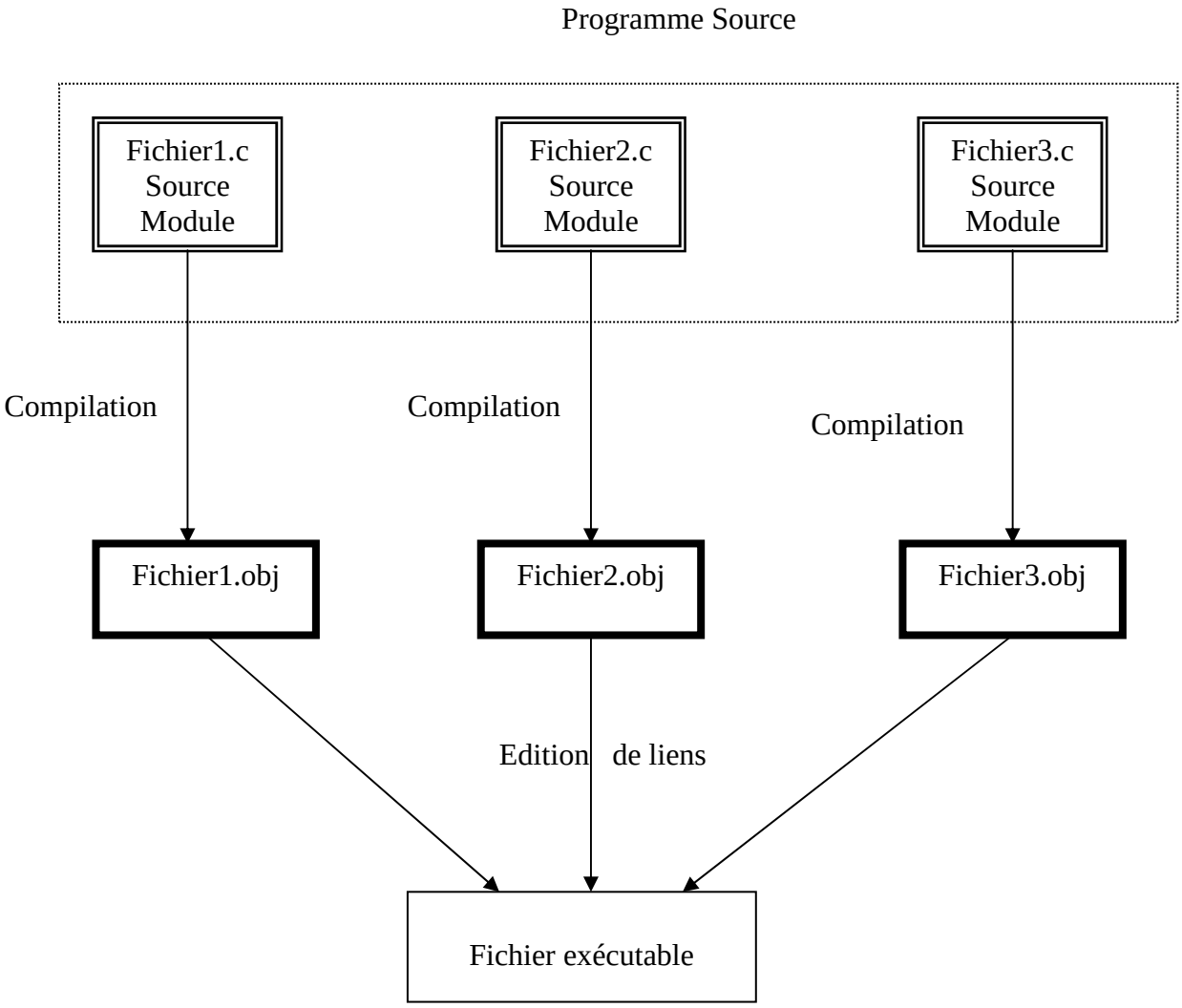
1. Pourquoi la programmation modulaire

Qui dit *programmation modulaire* dit programmation en modules. En C, cela correspond à diviser les sources d'un programmes en plusieurs fichiers. Cela offre plusieurs avantages :

- réutilisabilité ;
- facilité de compréhension ;
- travail en équipe;
- compilation fractionnée ;
- ...

Toutefois, lorsqu'une fonction d'un fichier fait appel à une fonction d'un autre fichier, il faut, à la compilation, connaître certaines choses sur la fonction appelée, notamment :

- être sûr qu'elle existe ;
- connaître son profil.



2. Rôle des fichiers d'en-tête

Ces fichiers servent à préciser le profil des fonctions d'un fichier source accessibles depuis d'autres. Il consistera principalement en la déclaration d'en-tête de fonctions. On trouvera aussi dans ce fichier la définition de certaines structures de données. Il constitue la *partie publique* de ce qu'il définit, la partie privée étant dans le fichier en ``.c" associé. Par conséquent, le fichier ``.h" doit aussi contenir tous les commentaires nécessaires pour comprendre à quoi ça sert, et comment utiliser les fonctions qui y sont définies.

Toutefois, on ne peut pas définir dans deux sources d'un même projet une même fonction. Or on peut avoir besoin, dans deux sources différents, de faire appel aux fonctions d'un même troisième fichier source. Pour éviter alors ce problème de multiple déclaration des fonctions, un fichier d'en-tête fait en général usage des directives #ifndef et #endif.