INTRODUCTION A LA PROGRAMMATION EN C

Formation des DES de médecine nucléaire. INSTN 1° année.

D.MARIANO-GOULART.
Service de médecine nucléaire. CHRU de Montpellier

OBJECTIF DU COURS :

❶ Etre capable de programmer de petits logiciels en langage C.
❷ Etre capable d'assurer la maintenance de logiciels dont on dispose des sources.

La programmation orientée objet ne fait pas partie des objectifs de ce cours d'initiation.
L'utilisation de cette police de caractères caractérise des instructions de C.

PLAN D'UN COURS DE 3 HEURES

1- Introduction

A- Présentation
B- Les étapes de la réalisation d'un programme exécutable

2- Le noyau du langage C

A- Eléments de syntaxe générale
B- Types de variables
C- Structures de contrôle
D- Pointeurs
E- Fonctions

3- Quelques fonctions de la bibliothèque

A- Entrées-sorties
B- Gestion des fichiers
C- Gestion de la mémoire

4- Travaux pratiques

1- INTRODUCTION

A- Présentation

Le langage C a été développé en 1972, par D.Ritchie et K.Thompson, pour écrire le code du système d'exploitation UNIX. Cette interfaçage à UNIX, sa souplesse d'utilisation, son efficacité et ses capacités d'accès à toutes les ressources matérielles ont rapidement suscité l'intérêt des programmeurs. Ce langage a ainsi pu se développer et est désormais très largement employé dans tous les domaines de la programmation :

① Programmation «système »
② Programmation scientifique
③ Programmation orientée «gestion de données »
④ Programmation orientée objet (C++, depuis 1982)
etc...

Par exemple, l'essentiel des applications fonctionnant sous le système d'exploitation Windows est écrit en C.

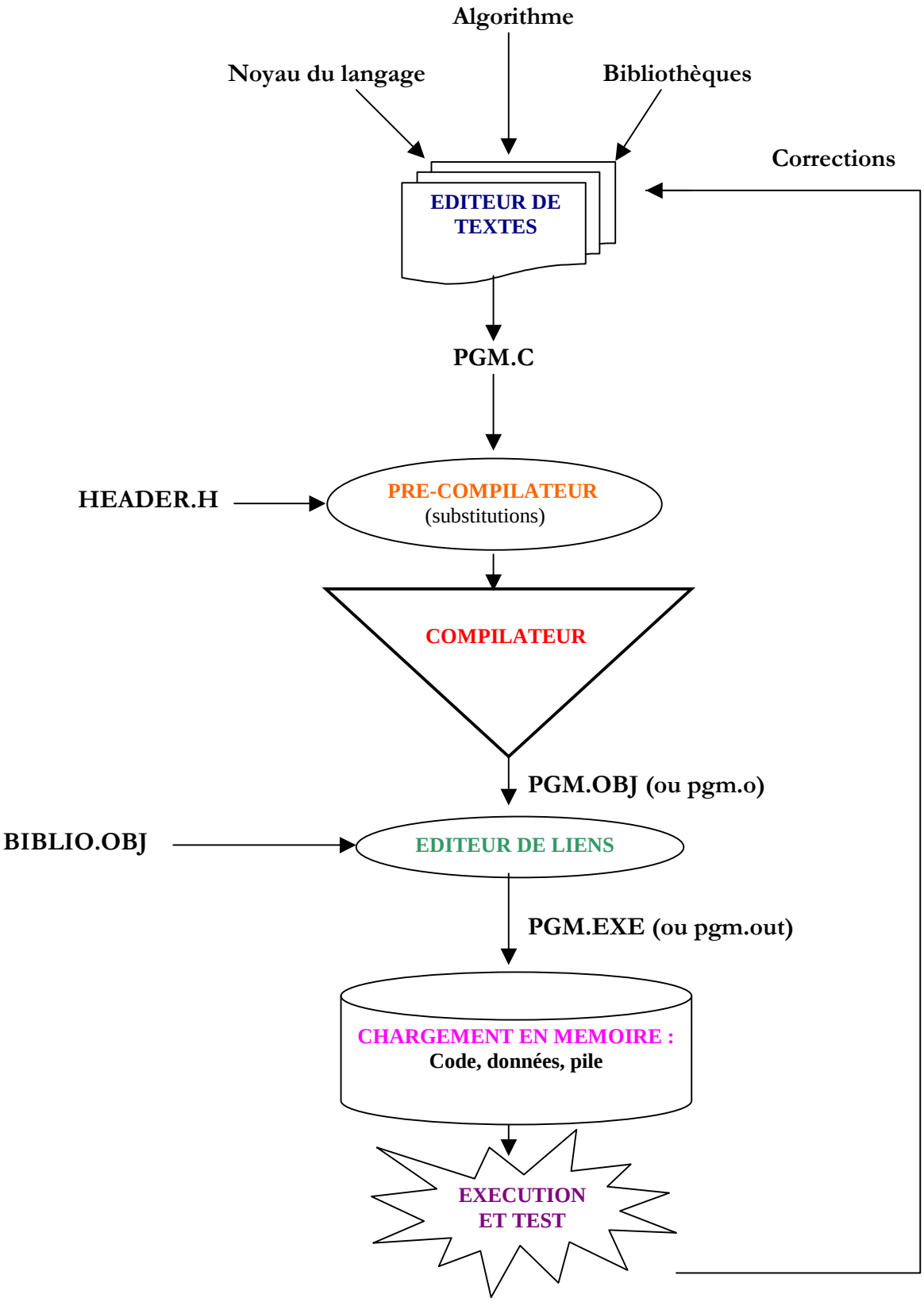
Le langage C peut-être utilisé à la fois à la manière d'un langage «de bas niveau » (type assembleur) et comme un langage «de haut niveau » (où l'on utilise directement des fonctions pré-programmées dédiées à une tache complexe et précise).

Le langage C a été conçu, à l'origine, pour des spécialistes de la programmation. En conséquence :

① Une partie seulement du langage est normalisée (ANSI), donc strictement compatible et portable. Les autres fonctionnalités, de plus haut niveau, dépendent du compilateur.

② Le compilateur n'effectue que peu de vérifications : certaines manipulations peuvent s'avérer dangereuses.

B- Les étapes de la réalisation d'un programme exécutable



2- LE NOYAU DU LANGAGE C

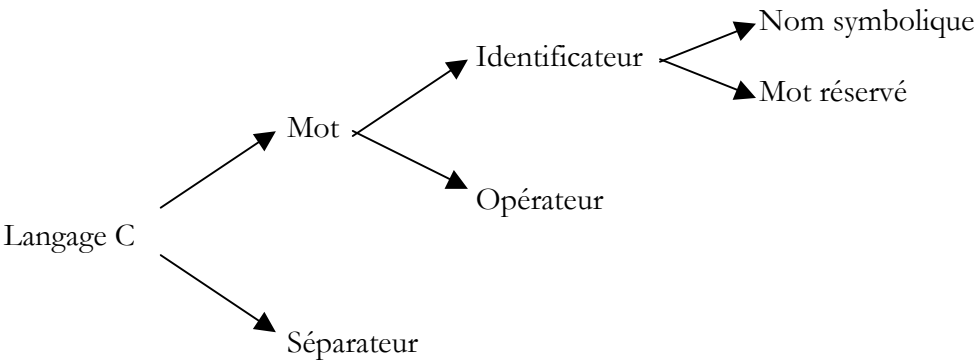
A- Eléments de syntaxe générale

① *Définitions*

Le langage C est constitué de mots individualisés par des séparateurs.
Un mot est un identificateur ou un opérateur agissant sur des identificateurs.
Un identificateur est un nom symbolique (de variable, de fonction, de sous-programme) ou un mot réservé du vocabulaire du langage.

Une instruction est une suite de mots et de séparateurs terminée par un point-virgule.
Une déclaration est une instruction particulière qui affecte un nom symbolique à une variable d'un type donné.

Un commentaire est une suite de caractère comprise entre les symboles /* et */ et qui est ignorée par le compilateur.



② *Les principaux séparateurs sont* :

{ }	bloc d'instruction, déclaration de structure, initialisation de tableau
( )	priorité, conversion, arguments d'une fonction
[ ]	indice d'accès dans un tableau
,	sépare les éléments d'une liste d'arguments
;	terminateur d'instruction
.	accès à un élément d'une structure
:	termine la mise en place d'une étiquette (switch, goto)

③ *Les principaux mots réservés concernent* :

❶ Types utilisés lors de la déclaration des variables
❷ Structures de contrôle : les boucles

④ ***Les différents opérateurs sont*** **:**

❶ Opérateurs généraux :

#	directive d'inclusion au pré-compilateur
=	Affectation
&	Adresse d'une variable
*	Contenu d'un pointeur sur une variable
->	Accès à un membre d'une structure pointeur

❷ Opérateurs arithmétiques :

-	inversion de signe
++	incrémentation de 1 (-- : décrémentation de 1)
+,-,*,/	addition, soustraction, multiplication, division
%	modulo (reste de la division entière)

ex :
```
int i , j, k ;
i = 9 ;
j = 3 ;
i++ ;            /* 10 = 9 + 1 est affecté à la variable entière i */
k = i%j ;        /* 1 = 10%3 est affecté à la variable entière k */
```

❸ Opérateurs logiques :

!	NON
==	égalité
!=	différent
>	relation d'ordre (>, >=, <, <=)
&&	ET
||	OU

Par convention, le résultat de l'évaluation d'une expression à l'aide d'opérateurs logiques est 1 si ce résultat est vrai, 0 s'il est faux.

Ex :
```
int i=1, j=2 ;
int r ;
r = ( (i+j)==3 ) ;/* 1 est affecté à la variable entière r */
```

❹ Opérateurs binaires :

~	complément à 1 (inverse la valeur de chaque bit)
<<n	décalage à gauche de n bits
>>n	décalage à droite de n bits
&	ET bit à bit
|	OU bit à bit
^	OU exclusif bit à bit

B- Types de variables

Une variable (qui contient une donnée nécessaire au programme) est caractérisée par :

- Un *identificateur* ou nom symbolique, qui la caractérise.
- Un *type* qui définit la taille de l'emplacement mémoire occupé par la variable.
- Une *adresse* qui correspond à l'emplacement du premier octet de la zone de stockage en mémoire centrale de l'ordinateur. Un octet est un regroupement ordonné de 8 bit (b). Sur un octet, il est donc possible de coder 256 objets différents. Un mot est un regroupement ordonné de 2 ou 4 octets, suivant l'ordinateur utilisé.

Toutes les variables utilisées dans un programme c doivent être *déclarées* avant leur utilisation (le plus souvent en début de programme). La déclaration d'une variable est donc une instruction qui précise, dans l'ordre, le type puis l'identificateur associés à cette variable. Lors de la déclaration, le compilateur se charge de réserver en mémoire une adresse et zone de stockage suffisante suffisante commençant à cette adresse.

Les caractères d'imprimerie et les séquences d'échappement (ou de contrôle telles que le retour en arrière, le saut de ligne ou de page etc.) sont associés à un code variant de 0 à 255 appelé le code ASCII. Caractères et séquence d'échappement sont ainsi manipulées sous la forme de simples variables numériques de type "unsigned char". A ce titre, les opérations arithmétiques usuelles sont possibles sur les caractères.

Exemple :
```
unsigned char a = 'A';  /* définit la variable a comme la lettre A */
                        /*  ou  de  façon  équivalente  comme  l'entier  65,
                        code ASCII de A */
unsigned char b;        /* définit la variable b comme un caractère */
a = b-1;              /* affection à la variable a la valeur 65 */
                        /* 65 est le  code ASCII de A */
```

les variables numériques peuvent être codées en utilisant une place mémoire variable suivant la précision et l'étendue de la gamme de valeurs nécessaires. Elles peuvent être entières ou réelles. Dans ce dernier cas, elles sont codées en machine sous la forme d'une mantisse et d'un exposant. La précision est de 7 chiffres significatifs pour un "float", 15 pour un "double" et 19 pour un "long double".

Par défaut, toutes les variables sont signées et un bit de la zone de stockage est utilisé pour coder le signe (bit de signe). Ces variables varient donc entre deux extrêmes centrées au mieux autour de 0. Il est cependant possible d'utiliser des variables non signées en les définissant au moyen du spécificateur de type unsigned. Dans ce cas, le bit de signe est disponible pour coder la valeur numérique de la variable, ce qui permet de coder des valeurs deux fois plus grandes.

Exemple pour une variable codée sur un octet, soit 8 bits :
```
char a;             /* a varie de -128 à 127 */
unsigned char ua;       /* ua varie de 0 à 255 */
```

① *Types de données* :

Les différents types de données sont résumés dans le tableau suivant :

Type	Nom	Unix/Windows	DOS	Limites
char	caractère	8 b	8 b	-E à E-1 où E = 2^B / 2 = 128
short	entier court	16 b	16 b	-E à E-1 où E = 32768
int	entier	32 b	16 b	celles de short ou long
long	entier long	32 b	32 b	-E à E-1 où E = 2 147 483 648
float	réel	32 b	32 b	$\pm 3{,}4.10^{-38}$ à $\pm 3{,}4.10^{+38}$
double	réel double précision	64 b	64 b	$\pm 1{,}7.10^{-308}$ à $\pm 1{,}7.10^{+308}$
long double	réel très grande précision	80 b	80 b	$\pm 3{,}4.10^{-4932}$ à $\pm 1{,}1.10^{+4932}$

Exemple de déclarations :

```
char car = 'A' ;
short i = -32768 ;
int i = 32767 ;
int i = 2147483647 ;
float pi = 3.14159 ;
double e = 2.7  ;
```

Remarque : Le type int correspond à la taille du "mot machine" de l'ordinateur utilisé, c'est-à-dire à la taille des registres de son processeur.

Représentation des variables: La valeur numérique d'une variable peut-être donnée en base 2 (binaire), 8 (octal), 10 (décimal) ou 16 (hexadécimal). On indique au compilateur qu'un entier est définit en octal en le faisant commencer par 0, et par 0x pour une définition en héxadécimal. Par exemple, dans un programme C, toutes les affectations suivantes aboutiront à donner à la variable b la valeur 66 en décimal, code ASCII du caractère 'B' :

```
unsigned char b;
b = 66;
b = 0102;          /* début par 0 ⇨ octal, i.e base 8 */
b = 0x42 ;         /* début par 0x ⇨ hexadécimal, i.e base 16 */
```

Le mot réservé **void** est utilisé pour signifier qu'une fonction ne renvoie aucune variable.

Tableaux : La définition de tableaux contenant un nombre prédéfini d'éléments se fait à l'aide de crochets. La numérotation des éléments d'un tableau commence à 0. Par exemple :

```
int entier[9] ;        /* tableau de 10 entiers : entier[0], entier[1], ... entier[9] */
char mois[9] ;         /* tableau de 10 caractères : mois[0], mois[1],...,mois[9] */
```

Un tableau à une dimension peut aussi être utilisé pour stocker un mot ou une chaine de caractères :

```
int nom[13]="Albert_Camus" ;  /* attention: pas d'espaces */
```

② *Modificateurs de types* :

const
```
const int i = 2 ;          /* i ne pourra pas être modifiée dans le programme */
```

enum
```
int i,j ;
enum jours { lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche } ;
i = lundi ;                /* i prend la valeur de l'entier 0 */
j = mardi ;                /* j prend la valeur de l'entier 1 */
```

(un)signed
```
unsigned int i = 65535 ;
signed int j ;             /* équivaut par convention à int j ; */
```

③ *Regroupement de variables de types différents* :

struct
```
struct calend
        {
        int jour  ;
        char mois[10]                /* ce champ est un tableau de 10 char */
        int annee ;
        } date;                      /* date est une variable de type calend */
date.jour = 20 ;
date.mois= « avril » ;
date.annee=1992 ;
```

En dehors de l'accès aux éléments d'une structure, la seule opération possible sur ce type de données est la copie de l'intégralité des champs d'une structure dans une autre (=).

Union
```
union U         {
                int i ;
                char c[2] ;
                } varunion ;          /* varunion est une variable de type U */
char car ;
varunion.i = 7687;                    /* 7687 = 0x3007 : initialisation */
car = varunion.c[0] ;                 /* car prend la valeur 0x07 */
car = varunion.c[1] ;                 /* car prend la valeur 0x30 */
```

L'union est donc une variable qui permet d'affecter une zone de mémoire à des objets de types différents au cours de l'exécution d'un programme. L'accès aux éléments d'une variable de type union est identique à celui utilisé pour les structures.

④ *Synonyme d'un type* :

typedef :
```
typedef int entier ;
entier i;
```

⑤ *Modificateurs de la portée d'une variable* :

Une variable définie à l'intérieur d'un bloc d'instruction (d'une fonction) est dite *locale*. Une variable locale est stockée dans la pile. Elle n'est connue et accessible qu'à l'intérieur de ce bloc. Au contraire, une variable définie hors de tout bloc d'instruction est dite *globale* : elle est alors stockée en zone de données et utilisable à tout endroit du code source.

```
register int i ;        /* le compilateur essaye de stocker i dans un registre du processeur*/
extern int i ;          /* i est définie à un autre endroit du code source */
static int i ;          /* i est une variable locale mais elle garde sa valeur d'un appel à l'autre de
                        la fonction où elle est définie */
```

⑥ *Taille d'un type de variable* :

Le mot réservé sizeof() renvoie la taille d'un type, exprimée en octet :

```
i = sizeof(char) ;      /* i prend la valeur 1 (octet) */
```

⑦ *Directives de substitution* :

Elles sont introduites par un # et ne sont pas terminées par un point-virgule en fin de ligne. Déclarées (en général) en début de programme, elles permettent la substitution de symboles (#define)et l'inclusion de fichiers headers (#include).

```
#define PI 3.14159
#define carre(x) x*x
#include « header_perso.h »
#include <stdio.h>
```

Cette dernière ligne permet au pré-processeur d'inclure au source le fichier /usr/lib/include/stdio.h qui contient les prototypes des fonctions standards d'entrées-sorties.

Des directives de compilation conditionnelles sont également disponibles afin d'incorporer ou d'exclure certaines portion du code source selon le résultat de l'évaluation de certaines conditions.

Exemple : compilation d'un code source en environnement DOS seulement :

```
#if(sizeof(int)==2)
        taille = 16 ;
#endif
```

C- Structures de contrôle

Ces structures permettent de contrôler, par test de différentes conditions, le déroulement séquentiel d'un bloc d'instruction.

① ***Boucles*** **:**

if :
```
if(condition) { instructions ; }
else { instructions ; }
```

while :
```
while (condition) { instructions ; }
do { instructions ; } while (condition) ;
```

Exemple :

```
int i=2, j=7, k ;
if(j<=i) { k=i ; i=j ; j=k ; }        /* permutte les valeurs de i et j si i ≥ j */
while(i<j) { i++ ; j-- ; }            /* affecte 5 à i et 4 à j */
```

for :
```
for( expression1, expression2, expression3) { instructions ; }
```

Le fonctionnement d'une boucle for est le suivant :

❶ l'expression 1, d'initialisation, est exécutée
❷ l'expression 2 est évaluée
❸ si l'expression 2 est fausse, on passe aux instructions situées après le bloc d'instructions entre accolades.
❹ sinon, les instructions entre accolades sont exécutées
❺ l'expression 3 est exécutée
❻ on recommence à l'étape ❷

exemples :

```
for (i=1 ; i<3 ; i++)  a[i] = i ;                    /* a[1] = 1 et a[2]=2 */
for(i=2, j=4 ; i<5 && j>2 ; i++, j--) a[i]=i+j ;     /* affecte 6 à a[2] et a[3] */
```

switch :
```
switch( varaible_test)
        {
        case 1 :        a=1 ;   break ;
        case 2 :        a=4 ;   break ;
        case 3 :        a=5 ;   break ;
        default :       a=0 ;
        }
```

② ***Sorties de boucles*** **:**

break : permet de sortir d'une boucle (for, while, do et switch).

Exemple :

```
int i=3 ;
while (1)                    /* boucle « potentiellement » infinie */
        {
        if ( !—i) break ; /*  après décrémentation, on quitte la boucle si i=0*/
        }
```

Continue : permet de passer directement à l'occurence suivante de la boucle sans exécuter les instructions de fin de boucle. Exemple :

```
a[5]=0
For(i=1 ;i<=10 ;i++)
        {
        if(i==5) continue ;
        a[i] = i ;
        }
```

goto : permet un branchement inconditionnel.

```
goto etiquette ;
etiquette : instruction ;
```

Comme dans les autres langages structurés, l'utilisation de cette instruction est déconseillée car elle rend les programmes moins lisibles (et donc plus difficiles à modifier).

③ ***Retour d'une fonction*** **:** return (cf. infra le paragraphe relatif aux fonctions).

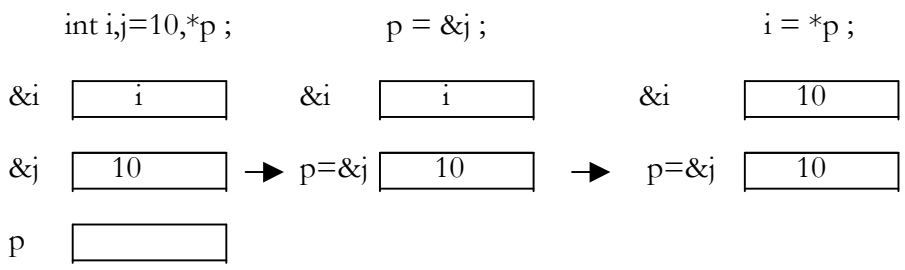
D- Pointeurs

Un pointeur est une variable qui contient l'adresse (en mémoire) d'une autre variable. Nous venons de voir comment manipuler une variable à l'aide de son identificateur. Celui-ci permet d'accéder à la table des symboles du programme (où à chaque identificateur correspond une adresse en mémoire). Un pointeur permet d'accéder directement à l'adresse en mémoire.

Déclaration et initialisation :

```
short i,j=10 ;
short *p ;        /* p est un pointeur sur un entier court */
p = &j ; /* la variable p contient l'adresse de j (pointe sur j) */
i = *p ;          /* affecte à i le contenu de p, soit 10 */
```

L'opérateur d'indirection (ou de déréférencement) * signifie «contenu de ».



En cas d'incrémentation d'un pointeur, l'unité d'incrémentation est la taille du type de la variable pointée. Dans l'exemple précédent, (p+1) pointe 2 octets (16 bits) après p dans la mémoire.

Nous avons vu que la référence à un tableau peut se faire à l'aide de crochets. Par exemple :

```
int i ;
int  a[3]={1,2,3} ;                  /* tableau de 3 entiers: a[0]=1, a[1]=2 et a[2]=3 */
for(i=0 ;i<3;i++) a[i]=i ;            /*ou bien : for(i=0 ;i<3 ;i++) *(a+i)=i ; */
```

Dans cet exemple, le nom du tableau (i.e. la variable a) est en fait un pointeur qui pointe sur le premier des 3 entiers du tableau. Dans ces conditions, on peut indifféremment accéder à ces entiers par l'expression a[i] ou par *(a+i).

Un pointeur peut aussi pointer sur une structure. L'accès aux champs de cette structure peut se faire de deux façons, soit habituellement à l'aide d'un identificateur du type (*pointeur).champ, soit sous la forme équivalente pointeur->champ.

Ex :
```
struct calend { int jour  ; int annee ; } *date ;
                                  /*  date pointe sur une variable de type calend */
(*date).jour = 20 ;
date->annee=1992 ;
```

E- Fonctions et procédures

De part la possibilité de programmation structurée et modulaire qu'elles impliquent, les fonctions expliquent en partie le succès des langages de programmation récents. Ainsi, un programme en C est constitué d'un ensemble de modules appelés fonctions et liés entre eux par différentes structures de contrôle. Comme les variables, une fonction se définie en précisant le type de la variable retournée à la fonction appelante, l'identificateur de la fonction et ses variables d'entrée.

Exemple d'un premier programme (qui calcule la somme de 1 et 2 !...) :

```
int somme(int i, int j) ;          /* prototype de la fonction somme */
                                   /* cette fonction a deux entiers en entrée */
                                   /* et retourne un autre entier */

int somme(int a, int b)            /* code de la fonction somme */
        {
        int c ;                    /* variable locale */
        c = a+b ;
        return c ;                 /* retour de l'entier c à la fonction appelante */
        }

void main()                        /* fonction principalle (qui ne retourne rien) */
        {
        int i,j,k ;                /* déclarations */
        i=1 ;                      /* initialisations */
        j=2 ;
        k=somme(i,j);   /* appel à somme() et affectation du résultat à k */
        }
```

Une fonction peut aussi ne rien retourner et opérer donc comme un sous programme. Une telle fonction sera de type *void*. Dans ce cas, cette fonction doit pouvoir modifier des variables locales définies dans la fonction appelante. Ceci est impossible si la fonction appelée ne connaît que l'identificateur de ces variables. En effet, modifier une variable nécessite de connaître l'adresse en mémoire de celle-ci de manière à pouvoir y écrire la valeur de la variable après modification. C'est donc l'adresse des variables à modifier qu'il faudra passer comme variables d'entrée à un sous-programme. Par exemple, le programme suivant ne modifiera pas la valeur de la variable k (qui restera égale à 0) :

```
I       void somme(int i, int j, int k) ;  /* prototype de la fonction somme */
                                           /* cette fonction a trois entiers en entrée */
                                           /* et ne retourne rien */

        void somme(int a, int b, int c)    /* code de la fonction somme */
                {
                c = a+b ;
                }
```

```
void main()                    /* fonction principalle (qui ne retourne rien) */
        {
        int i,j,k ;            /* déclarations */
        i=1 ;                  /* initialisations */
        j=2 ;
        k=0 ;
        somme(i,j,k);          /* appel à somme() */
        }
```

Par contre, le programme suivant affectera bien 3 à la variable k :

```
I       void somme(int i, int j, int *k) ;      /* prototype de la fonction somme */
                                                /* cette fonction a 2 entiers et 1 pointeur */
                                                /* sur un entier en entrée */
                                                /* et ne retourne rien */
        void somme(int a, int b, int *c)        /* code de la fonction somme */
                {
                *c = a+b ;                      /* la valeur a+b est mise à l'adresse c */
                }

        void main()                             /* fonction principalle (qui ne retourne rien) */
                {
                int i,j,k ;                     /* déclarations */
                i=1 ;                           /* initialisations */
                j=2 ;
                k=0 ;
                somme(i,j,&k);                  /* appel en envoyant l'adresse de l'entier k */
                }
```

Reste désormais à pouvoir afficher les résultats produits par un programme. On utilise pour cela (entre autre) les bibliothèques de fonctions du compilateur. Nous allons en étudier les principales.

3- QUELQUES FONCTIONS DE LA BIBLIOTHEQUE

A- Entrées-sorties

① Affichage formaté à l'écran, renvoie le nombre d'octets affichés (ou EOF)

Prototype : int printf(const char *format, arguments)
Exemple :

```
#include <stdio.h>
void main()
        {
        int i=1,j=2,k ;
        char texte[5]="plus" ;
        char c='=' ;
        k=i+j ;
        printf("\n %d %s %d %c %d \n",i,texte,j,c,k) ;
        }
```

② Récupération formatée de caractères tapés au clavier, renvoie le nombre de champs lus.

Prototype : int scanf(const char *format, arguments)
Exemple :

```
#include <stdio.h>
void main()
        {
        int i,j,k ;
        printf("\n Entrez deux entiers : ") ;
        scanf("%d %d",&i,&j) ;
        k = i+j ;
        printf("\n Leur somme vaut %d",k) ;
        }
```

③: Renvoie la valeur ASCII d'un caractère entré au clavier.

Prototype : int getchar(void)
Exemple :

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

B- Gestion des fichiers

Les différentes opérations sur les fichiers se font par l'intermédiaire d'un pointeur sur une structure complexe prédéfini (dans stdio.h). Ce pointeur, de type FILE, est retourné par la fonction fopen au moment de l'ouverture d'un fichier.

① *Ouverture et fermeture d'un fichier*

Prototype : FILE *fopen(const char *nom_fichier, char *type)
int fclose(FILE *pointeur)
Exemple :

```
#include <stdio.h>
void main()
        {
        FILE *fichier ;
        fichier=fopen("c:\mon_texte.txt","rb") ;
        if(fichier==NULL) exit(1) ;
        fclose(fichier) ;
        }
```

② *Lecture et écriture dans un fichier*

Prototype : int fprintf(FILE *pointeur,const char *format, arguments)
int fscanf(FILE *pointeur,const char *format, arguments)
Exemple :

```
#include <stdio.h>
void main()
        {
        char chaine[20] ;
        int entier ;
        float reel ;
        FILE *fichier ;
        fichier=fopen("c:\mon_texte.txt","rb") ;
        if(fichier==NULL) exit(1) ;
        fscanf(fichier,"%d %f %s\n",&entier,&reel,chaine) ;
        fclose(fichier) ;
        fichier=fopen("c:\mon_texte2.txt","wb") ;
        if(fichier==NULL) exit(1) ;
        fprintf(fichier,"%d %f %s\n",entier,reel,chaine) ;
        fclose(fichier) ;
        }
```

③ *Repositionnement du pointeur de fichier*

Prototype : int fseek(FILE *pointeur, long décalage, int origine)

Ou origine vaut SEEK_SET (décalage depuis le début du fichier), SEEK_CUR (depuis la position courante) ou SEEK_END (depuis la fin du fichier).

C-Gestion de la mémoire

① *Allocation de mémoire*

Prototype : void *malloc(size_t taille)
size_t est un synonyme de unsigned long, défini dans stdlib.h.

② *Libération de mémoire*

Prototype : void free(void *pointeur)

③ *Lecture/ecriture de blocs de données*

Prototype : size_t fread(void *pointeur, size_t taille, size_t nb_elts, FILE *pointeur)
size_t fread(void *pointeur, size_t taille, size_t nb_elts, FILE *pointeur)

Exemple : programme de seuil du bruit de fond d'une image.

```
#include <stdio.h>
#include <stdlib.h>
void main()
        {
        int i ,taille=128*128;
        short int *pixels ;
        FILE *fichier ;

        /* Allocation d'un pointeur sur les pixels */
        pixels = malloc(taille*sizeof(short int)) ;
        if(pixels==NULL) exit(1) ;

        /* Ouverture du fichier contenant l'image */
        fichier=fopen("c:\image.raw","rb") ;
        if(fichier==NULL) exit(1) ;
        fread(pixels,taille,sizeof(short int),fichier) ;
        fclose(fichier) ;

        /* Seuillage de l'image */
        for(i=0 ;i<taille ;i++)  if( *(pixels+i)<10)  *(pixels+i) = 0 ;

        /* Ecriture de l'image seuillée */
        fichier=fopen("c:\seuil.raw","wb") ;
        if(fichier==NULL) exit(1) ;
        fwrite(pixels,taille,sizeof(short int),fichier) ;
        fclose(fichier) ;

        /* libération de mémoire */
        free(pixels) ;
        }
```