LANGAGE C

NOTIONS FONDAMENTALES

Michel NICOLLET

SERVICE D'AERONOMIE tour 15-14 5è case 102 mail:michel.nicollet@aero.jussieu.fr

1 STRUCTURE D'UN PROGRAMME C

1.1 PROGRAMME C

Comprend une ou plusieurs fonctions dont l'une doit s'appeler "**main**" stockées dans un ou plusieurs fichiers.
Les instructions du préprocesseur commencent par **#**

1.2 FONCTION

Entête défini par un type et un nom de fonction suivis d'une liste de types d'arguments entre parenthèses.

Le corps de la fonction délimité par les caractères **{ }** est composé d'instructions simples ou composées.

Toutes les instructions simples se terminent par **;**

Une instruction composée ou bloc est un ensemble d'instructions simples délimitées par des accolades **{}**.

Les commentaires sont encadrés par les délimiteurs **/*** et ***/** .

1.3 EXEMPLE

```
#include<stdio.h>     /* accès aux entrées sorties*/
#include<stdlib.h>    /* accès à la librairie mathématique*/
/*calcul de pi*/
void main(void)
{float caleat(void);
float x,y,d2,pi;
int np,nc,i;

printf("combien de points?");
scanf("%d",&np);

for(nc=0,i=1;i<=np;i++)
{x=caleat();              /*appel à une fonction*/
y=caleat();
d2=(x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
if(d2<=0.25)nc++;
}
pi=(4.0*nc)/np;
printf("estimation de pi avec %d points:%e",np,pi);/*affichage résultat*/
}

float caleat(void)              /*fonction de tirage de variables aléatoires*/
{
return ((float)rand()/(RAND_MAX+1.0));
}
```

2 COMPILATEUR ET EDITION DE LIENS

La source d'une application écrite en langage C peut être stockée dans un ou plusieurs fichiers dont le suffixe est **.c** .

La compilation de la source s'effectue par la commande **cc** . Cette commande enchaîne 3 étapes:

1. appel du préprocesseur **cpp**
2. appel au compilateur
3. appel à l'éditeur de liens.

La commande admet plusieurs options:

-E: permet de n'exécuter que la première étape.

-P: idem à -E mais la sortie est stockée dans un fichier dont le nom est celui du source suivi du suffixe **.i** .

-c : permet l'exécution des deux premières étapes et on récupère le module objet stocké dans un fichier dont le nom est identique à celui du source suivi du suffixe **.o**.

-o: permet de renommer le module exécutable dont le nom par défaut serait **a.out**.

Exemple: cc essai -c ess.exe

3 LES DECLARATIONS DE VARIABLES OU CONSTANTES

3.1 Les types de base

Le langage contient des types de base qui sont les entiers, les réels simple et double précision et les caractères, de plus il existe un type ensemble vide;
Les mots-clés des types permettent d'influer sur la taille mémoire des variables déclarées.

3.1.1. types entiers

déclaration	Taille(octets)	valeurs
int	4	-2e(31) à 2e(31)-1
unsigned int long int unsigned long int	4 4 4	0 à 2e(32)-1
short int unsigned short int	2 2	-32768-32767 0-65535

La taille d'un entier est par défaut soit 2 soit 4 (dépend de la machine). 4 octets est la taille la plus courante.
Les types short int et long int peuvent être abrégés en short et long.
Le mot clé unsigned indique si le bit de poids fort doit être ou non considéré comme un bit de signe.

3.1.2. types réels

declaration	Taille(octets)	valeurs	décomposition
float	4	3.4 e-38 à 3.4 e+38	exp(8bits) mantisse(23)
double	8	1.7 e-308 à 1.7 e+308	exp(11) mantisse(52)
long double	10	3.4 e-4932 à 3.4 e+4932	exp(15) mantisse(64)

Les opérations se font toujours sur des valeurs de type double.
Le tableau doit être complété de façon symétrique par les valeurs négatives

3.1.3. types caractères

déclaration	Taille(octets)	valeurs
char	1	-128 à 127
unsigned char	1	0 à 255

Le type char est considéré comme un entier qu'on pourra utiliser dans une expression arithmétique.

3.1.4 forme générale d'une déclaration

La forme générale est la suivante:
<type> variable, variable, …;
exemple: int i,k;
double x;

3.2 conversion de types

3.2.1 conversion implicite

Si un opérateur porte sur deux opérandes de types différents, il y a conversion du type le plus faible dans le type le plus fort (nombre d'octets).

3.2.2 conversion forcée

Utilisation de fonctions de conversion appelées opérateur de cast.
Syntaxe:

```
(float) n
(int) x
(float)(n/p)
(double) n/p
```

3.3 déclaration de constantes symboliques

Soit à l'aide du préprocesseur
#define
exemple: #define x 100
Toute valeur de x dans le programme aura la valeur 100

```
#define entier int
#define debut {
```

Soit par l'attribut **const**

```
Const int n=10;
```

n est une constante entière qui ne peut être modifiée dans la fonction où elle est déclarée.

3.4 définir un nouveau type

Au moyen du constructeur **typedef**
Le constructeur typedef permet de donner un nouveau nom à un type déjà existant.

Syntaxe:

```
typedef <déclaration>
```

Exemple:

```
typedef int entier
typedef  float reel
typedef  double variable
```

Rem: Attention à la syntaxe différente de l 'instruction **define** où il y a affectation tandis que pour **typedef** il y a synonyme

4 ECRITURE DES CONSTANTES NUMERIQUES

4.1 entières

-décimales
-octales: nombre écrit en base 8 précédé du chiffre 0
-héxadécimales: nombre écrit en base héxadécimale précédé des caractères 0x ou 0X

Possibilité de gérer des constantes sur 32 bits en ajoutant L ou l à la suite de sa valeur.
Possibilité de gérer des constantes non signées en ajoutant u ou U à la suite de sa valeur.

4.2 réelles

-flottantes: doivent comporter un point.
-exp: lettre e sans espace dans l'exposant.

Possibilité de gérer des constantes de type float en faisant suivre son écriture de la lettre F ou f .

4.3 caractères

-directement entre apostrophes ' ' : soit 's'; '+';'10'.
-soit à partir du code ASCII sous forme octale: '\012'
-soit à partir du code ASCII sous forme héxadécimale: '\x7'

remarque générale:
Les constantes sont converties par le préprocesseur dans les calculs en type double.

5 OPERATEURS ET EXPRESSIONS

Une expression est constituée de variables et constantes (littérales ou symboliques) reliées par des opérateurs.
Lorsque les types des opérandes sont différents, il y a conversion implicite dans le type le plus fort.

5.1 opérateurs arithmétiques

+, -, *, /, %
Pas d'opérateur de puissance; appel à la fonction **pow**.
A priorité identique: associativité de gauche à droite.

5.2 opérateurs relationnels

<, <=, >, >=, ==(identique), **!=** (différent)
Le résultat d'une comparaison est un entier: 1 si vrai 0 si faux.

5.3 opérateurs logiques

&& (et), || (ou), **!** (négation)
Les opérateurs logiques acceptent tout opérande numérique avec: 0 correspond à faux
(renvoi 0) et toute valeur numérique !=0 à vrai (renvoi 1).

5.4 opérateurs d'incrémentation et de décrémentation

++, -- ce sont des opérateurs unaires permettant d'ajouter ou de retrancher **1** au contenu de leur opérande.
Cette opération est effectuée après ou avant l'évaluation de l'expression suivant que l'opérateur suit ou précède son opérande

Exemple: ++i pré incrémentation de i de 1
i++ post incrémentation de i de 1

5.5 opérateurs d'affectation logique

Opérateur suivi d'une affectation: i+=k équivalent à i=i+k
i*=(i-3) équivalent à i=i*(i-3)

Ceci peut s'employer avec beaucoup d'autres opérateurs.
+=, -=, *=, /=, %=

Syntaxe générale:
ivalue=ivalue opérateur expression, équivalent à ivalue opérateur=expression

5.6 opérateur conditionnel

?: opérateur ternaire
Syntaxe:
exp1? exp2 : exp3

La valeur de l'expression exp1 est interprétée comme un booléen. Si elle est vraie c'est à dire non nulle seule l'expression 2 est évaluée sinon c'est l'expression 3 qui est évaluée.
Exemple: (i<0)?0:10;

5.7 opérateur séquentiel

,
Possibilité de plusieurs séquences: i++,j=i+k;

5.8 opérateur d'adressage

&
Cet opérateur appliqué à un objet renvoie l'adresse en héxadécimale de cet objet.

5.9 opérateur de taille

sizeof

Renvoie la taille en octets de son opérande. L'opérande est soit une expression soit une expression de type.
Exemple: sizeof i;
sizeof (int)

5.10 opérateurs de manipulation de bits

5.10.1 opérateurs arithmétiques bit à bit

Ils correspondent aux 4 opérateurs de l'arithmétique booléenne:
~ non logique
& et logique
| ou logique
^ ou exclusif
Les opérandes sont de type entier et les opérations s'effectuent bit à bit suivant la logique binaire.

5.10.2 opérateurs de décalage

>> décalage à droite
<< décalage à gauche
Exemple: n<<2
Le motif binaire est décalé du nombre de bits indiqué. Dans le cas d'un décalage à gauche les bits les plus à gauche sont perdus. Les positions binaires vacantes sont remplies par des 0.

Pour un décalage à droite, les bits le plus à droite sont perdus. Si l'entier décalé est non signé les positions binaires vacantes sont remplies par des 0, s'il est signé le remplissage se fait à l'aide du bit de signe (0 ou 1).

6 SORTIES-ENTREES

6.1 LIEN

Faire appel au préprocesseur par la commande:

#include<stdio.h>

6.2 SORTIES

6.2.1 syntaxe

printf("codes format",liste d'arguments);

-code format: constante chaîne à afficher contenant ou non un code format précédé par **%** . Le tout entre quotas.
-liste d'arguments: suite de variables séparées par des virgules;

6.2.2 code format

a -entier
%d libre
%nd n: nombre de caractères à afficher avec cadrage à droite.
%-nd idem avec cadrage à gauche
%nld entier long, ou %ld ou %Ld
%u entier sans signe et %lu pour sans signe et long
%+d pour afficher le signe

b -réel flottant
%f 6 chiffres après le point décimal avec cadrage à droite
%lf flottant double
%Lf flottant long double
%nf n: nombre total de caractères à afficher avec toujours 6 chiffres décimaux
%n.pf p: indicateur de précision correspondant au nombre de chiffres décimaux à garder

c -réel exponentiel
%e 13 caractères en tout affichés dont 3 pour la puissance et 6 chiffres décimaux
14 caractères si la valeur est négative
%le exp double
%Le exp long double
%n:pe n: nombre total de caractères et p nombre de chiffres décimaux

d -précision variable
%n.* affichage en flottant ou * signifie que la précision est fournie dans la liste d'arguments

e -octal et héxadécimal
%o valeur en octal
%x valeur en héxadécimal

f -caractère
%c (voir aussi d'autres fonctions telles que **putchar()**

6.2.3 format de mise en page

\t tabulation
\n passage à la ligne

autonome ou avec un code format: %d\t; \%td

6.3 ENTREES

6.3.1 syntaxe

scanf("format",liste d'arguments);

-format: codes format entre quotas
-liste d'arguments: adresse de variables (variables précédées de &) et séparées par des virgules

6.3.2 codes format

Constitués du caractère % suivi d'une spécificité de format

-a entier	%d ; %u (entier non signé)
-b réel flottant	%f accepte indifféremment un nombre écrit sous forme entière, décimale ou exponentiel.
-c réel exponentiel	%e idem " "
-d caractère	%c attention tout espace est considéré comme un caractère. Voir aussi les macros **getchar()** et **fgetchar()**
-e octal	%o
-f hexadécimal	%x

7 INSTRUCTIONS DE CONTROLE

Ces instructions sont les choix, les boucles et les branchements.

7.1 INSTRUCTIONS DE CHOIX

7.1.1 instruction if

Syntaxe

```
if (expression)          /* expression logique ou relationnelle)
{
        bloc1;
}
else
{
        bloc2;
}
```

Le bloc1 sera exécuté si l'expression est non nulle, sinon c'est le bloc2 qui sera exécuté.
Le **else** est facultatif if (expression) {bloc;}
Si plusieurs choix sont impliqués le **else** se rapporte au **if** le plus interne.

7.1.2 instruction switch

Permet d'effectuer un branchement à une étiquette en fonction de la valeur d'une expression.
Syntaxe

```
switch(expression)    /*expression numérique entière*/
{case étiq1: instruction simple ou composée;
case etiq2:     "       "       "       "
-------------------------------------------------
case etiqn:     "       "       "       "
default:        "       "       "       "
}
```

Les étiquettes doivent être des expressions **constantes entières.**

Le programme envoie à l'étiquette correspondant à la valeur de l'expression et exécute toutes les instructions suivantes jusqu'à la fin du bloc switch. Il existe une possibilité de forcer la sortie par l'instruction **break**.

7.2 INSTRUCTIONS DE BOUCLE

7.2.1 instruction for

Syntaxe

```
for(exp1;exp2;exp3)
```

exp1: est évaluée une seule fois au début de la boucle, permet d'initialiser les paramètres de boucle.
exp2: est évaluée et testée avant chaque passage dans la boucle (condition logique).
exp3: est évaluée après chaque passage, permet l'incrémentation.

Possibilité de créer plusieurs boucles imbriquées à condition qu'elles ne se chevauchent pas.

7.2.2 instruction while

Syntaxe:

```
while (expression)
{
        bloc;
}
```

répétition du bloc tant que l'expression est vraie; la boucle peut n'être parcourue aucune fois.

7.2.3 instruction do while

Syntaxe:

```
do
{
        bloc;
}while(expression);
```

Le test est effectué après le corps de boucle qui sera exécuté au moins une fois.

7.3 INSTRUCTIONS DE BRANCHEMENT

7.3.1 instruction break

Permet de quitter une boucle ou de sortir d'une instruction de test. Renvoie à la première instruction qui suit la boucle ou le test.

7.3.2 instruction continue

Elle permet de passer à l'itération suivante dans une boucle.

7.3.3 instruction go to

Permet le branchement en un point quelconque du programme
Syntaxe:

```
go to etiquette;
```

etiquette fait référence à une instruction étiquetée.
Exemple:

```
for(     )
{
if(  ) goto sortie;
}
sortie: instruction;
```

7.3.4 instruction d'interruption

Un programme peut être interrompu par l'instruction **exit**
Syntaxe: **exit(exp);** exp doit être un entier indiquant le code de terminaison du programme.
Exemple: **exit(1);**

8 VECTEURS (tableaux)

8.1 VECTEURS UNE DIMENSION

Syntaxe: **t[i]** t: nom du vecteur; i indice; t[i] est la valeur du vecteur dont la position est fournie par l'indice i.

Déclaration: **int t[n]; /*vecteur de n éléments*/**
float tab[n];
double vect[10];

n est le nombre total d'éléments du tableau (taille) qui doit être déclaré avant, en particuliers par une constante symbolique.
La taille peut être vide, c'est alors l'initialisation qui définit le nombre d'éléments.
Le premier élément d'un vecteur porte l'indice 0. l'indice varie donc de 0 à n-1.

8.2 VECTEURS DEUX DIMENSIONS

Syntaxe: **t[i][j]**

Déclaration: **t[3][2]** vecteur de 6 éléments ou 3 vecteurs de 2 éléments.

Arrangement en mémoire: t[0][0]
t[0][1]
t[1][0]
t[1][1]
t[2][0]
t[2][1]

8.3 INITIALISATIONS

a) int t[4]={10,20,30,40};
ou **int t[]={10,20,30,40};** vecteur de 4 éléments

b) **int t[3][2]={1,2,3,4,5,6};**
ou **int t[3][2]={{1,2},**
{3,4},
{5,6}};

9 FONCTIONS

9.1 PROPRIETES GENERALES

- Toutes les fonctions sont indépendantes les unes des autres et sont compilées séparément.
-Toutes les fonctions doivent en principe être déclarées avant leur définition au moyen d'un prototype.
(Le compilateur doit connaître le type de la valeur retournée ainsi que le nombre et les types des arguments transmis).
-Le nom de la fonction est précédé d'un type indiquant le type de la valeur qu'elle retourne et est suivi d'arguments ou paramètres formels. Ces arguments sont passés soit par valeur soit par adresse.
-Le retour d'une valeur se fait par l'instruction **return.**
-Si la arguments sont transmis par valeur, ils peuvent être modifiés par la fonction sans être modifiés dans la fonction appelante. Les variables de la fonction sont alors locales et un **nouvel espace mémoire est alloué à chaque appel de la fonction**. Elles ne gardent donc pas leur valeur d'un appel à l'autre.
-Si les variables doivent garder leur valeur d'un appel à l'autre elles devront être déclarées comme statiques.
-Si plusieurs fonctions doivent partager les mêmes variables celles ci devront être déclarées comme variables globales ou transmises par adresse.

9.2 PASSAGE ARGUMENTS

9.2.1 par valeur

```
void main(void)
{ float som(int);      /*declaration ou prototypage*/
int a=10;
float d;
d=som(a);              /* appel de la fonction avec argment par valeur*/
}
float som(in n)        /*definition de la fonction et argument formel*/
{
float s=10;
s+=n;
return s;              /* valeur retournee à la fonction main*/
}
```

9.2.2 passage d'un vecteur

Un vecteur est considéré comme une contante symbolique. Lorsqu'un vecteur est passé comme argument en mentionnant son nom sans crochet ni indice, **le nom du vecteur est interprété comme un pointeur dont la valeur est l'adresse de son premier élément (passage par adresse).**
Passer un vecteur comme argument , c'est donc transmettre l'adresse du début du code machine constituant le tableau.
-conséquence: la fonction peut modifier les valeurs, dans la fonction appelante, du vecteur transmis.

Exemple:

```
#define N 4
void main(void)
{ int tab[N]={1,5,8,10};
int som(int []),som;
som=som(tab);
}
```

```
int som(int t[])
{int i,som;
for( i=0,som=0;i<N;i++)
som+=t[i];
return som;
}
```

9.3 RECURSIVITE

-récursivité directe: une fonction comporte un appel à elle même.

-récursivité croisée: une fonction appelle une autre fonction qui appelle la fonction initiale.

10 CLASSES D'ALLOCATION OU CLASSES DE MEMORISATION

Il faut distinguer la durée de vie d'une variable, temps pendant lequel la variable a une existence en mémoire, et son emplacement en mémoire. Ces deux notions sont définies par l'emplacement de la déclaration des variables.

10.1 CLASSES AUTOMATIQUES: classes auto

Par défaut d'attribut de classe mémoire dans une fonction une variable est de classe auto: variable interne à la fonction et temporaire.
Propriétés:
-la valeur de la variable n'est pas conservée d' un appel à l'autre.
-la durée de vie de la variable est limitée à celui de la fonction.
-un nouvel espace mémoire est alloué dynamiquement dans la pile à chaque entrée dans la fonction. (pile de type LIFO)
-initialisation obligatoire.
-les variables peuvent être déclarées par **auto.** Exemple **auto int i;**

10.2 CLASSES STATIQUES

Déclaration de type **static**.
Possibilité de conserver un emplacement mémoire permanent et de conserver ainsi la valeur d'un appel à l'autre.
Par défaut initialisation de la valeur à 0 sinon à l'aide d'expressions constantes.
La durée de vie est celle du programme.

Exemple: float som(int n)
{ static float s;

}

10.3 CLASSES GLOBALES

Plusieurs fonctions pourront partager des variables communes
Sa déclaration se fait à l'extérieur des fonctions. La portée de la variable est alors l'ensemble du programme source à partir de l'endroit où elle est déclarée. Donc seules les fonctions définies après la déclaration des variables externes peuvent y accéder.
Si une déclaration dans une fonction est faite avec un nom homonyme de la variable externe la variable locale automatique est prioritaire.
Par défaut initialisation de la valeur à 0 sinon à l'aide d'expressions constantes.

Les fichiers source sont traités de façon indépendante par le compilateur; si l'on a besoin de partager une variable entre plusieurs fichiers on devra allouer son emplacement mémoire dans un seul de ces fichiers et faire précéder sa déclaration par **extern** .

Déclaration de type **extern**.

Exemple
```
extern float x;
void main(void)
{
-------------
}
double f(int n)
{
-------------
}
```

Au contraire si l'on doit cacher la variable aux autres fichiers sources il faut faire précéder sa déclaration par static . **static float x;**

10.4 FONCTIONS

Les fonctions suivent les mêmes règles de classes d'allocation que les variables.

11 POINTEURS

11.1 DEFINITION

Manipulation d'adresses par l'intermédiaire de variables nommées pointeurs. Un pointeur est une variable dont la valeur est une adresse.
L'adresse est indissociable du type de la variable, on définira donc des pointeurs sur des entiers, des caractères, des réels ou sur des objets plus complexes.

11.2 DECLARATION

Syntaxe: **type *nom;**
nom de la variable pointeur
type de la donnée que désigne le pointeur

```
Exemple:  int *pt;        /*pt est un pointeur sur des entiers*/
          int n,u;
          n=10;
          pt=&n;          /*affecte à pt l'adresse de n*/
          u=*pt;          /*contenu de la variable pointée soit u=10*/
```

Ne pas confondre le caractère * dans la définition du pointeur avec le même caractère ***** de la dernière ligne qui est un opérateur **d'indirection** qui évalue le contenu de l'adresse pointée.

11.3 OPERATIONS SUR LES POINTEURS

11.3.1 opérations sur les adresses

Si pt est un pointeur sur des entiers longs on peut travailler sur les adresses ainsi:
pt+3 :décalage de 3*4(octets)=12 octets de décalage d'adresse dans la pile.
++pt: incrémentation de 4 octets.

Tous les opérateurs d'addition et de soustraction sont admis ainsi que les opérateurs de comparaison à condition qu'ils portent sur des pointeurs de même type.

11.3.2 opérations sur les valeurs

Toutes les opérations sont possibles sur les valeurs du contenu d'un pointeur. Il est conseillé d'utiliser des parenthèses:

```
u=(*pt)+5; /*u=15*
```

11.4 POINTEURS ET VECTEURS

Les pointeurs sont étroitement liés aux tableaux dont les noms sont les adresses du premier élément.
Le langage C ne fait pas la distinction entre les pointeurs et les vecteurs. Un vecteur est accessible en prenant comme adresse la position du premier élément.
Les notations **int t[10]** et **int *t** sont identiques. t pointeur est équivalent à &t{0]. Ceci est utile pour la gestion des tableaux dynamiques.
Ainsi **t[i]** équivalent à ***(t+i)** ; i est un décalage d'adresse (offset).t+i représente &t[i] et *(t+i) est le contenu de cette adresse.

Dans la fonction **scanf** l'opérateur d'adressage **&** est superflu pour un tableau dont le nom constitue en soi une adresse.

11.5 POINTEURS ET FONCTIONS

11.5.1 vecteur comme argument d'une fonction

Passage en argument de l'adresse du tableau de sorte que la fonction pourra effectuer toutes les manipulations voulues à partir de cette adresse. Un tableau passé en argument à une fonction est traité comme un pointeur sans qu'il soit nécessaire de faire précéder son nom par l'astérisque.
Exemple:

```
int t[10];              ou int *t;
double fct(int []);     ou double fct(int *);
fct(t);                 /*transmission de l'adresse de t[0]*/

double fct(int tab[])  ou double fct(int tab)      /définition de la fonction*/
```

Une fonction peut travailler avec un tableau de taille quelconque à condition de lui transmettre la taille en argument. On peut également passer qu'une partie d'un tableau à une fonction, il suffit de lui indiquer l'adresse du i élément de début.
Exemple: **fct(&t[5]);**

11.5.2 fonction comme argument d'une fonction

Le nom d'une fonction est une constante symbolique dont la valeur est un pointeur sur la première instruction exécutable du code machine de la fonction.
Passer une fonction en argument, c'est transmette l'adresse de la fonction.
Exemple:

double fct(double(*f)(int n),) /* définition de la fonction. f est un pointeur sur une fonction quelconque acceptant un argument entier*/
équivalent à double fct(double f(int n),) /* f est le nom d'une fonction précise*/

prototype équivalent plus court :

double fct(double(*)(int),) /*déclaration de la fonction*/
équivalent à double fct(double f(int),)

L'appel se fait en passant simplement le nom de la fonction.

11.6 OBJETS PLUS COMPLEXES

-Une fonction peut renvoyer un pointeur; cela se déclare en insérant un ***** devant le nom de la fonction.

int *fct(double); /*la fonction fct renvoie une adresse sur des entiers*/
ou **int *(*f)(double); /* f est ici un pointeur sur une fonction*/**

-pointeur de pointeur

char ** pt; /*pointeur de pointeur de caractère*/
contient l'adresse d'un pointeur sur des caractères

-vecteurs multidimensionnels

int (*t)[10]; /*t est un pointeur sur un ensemble de vecteurs de 10 éléments*/
t pointe sur la première ligne du premier vecteur.

12 STRUCTURES

12.1 DEFINITION

Une structure est un ensemble hétérogène permettant de définir des objets de nature et de type différents.

12.2 DECLARATION

12.2.1 syntaxe

La déclaration est faite en dehors de toute fonction suivant la syntaxe:

```
struct nom
{
liste de déclarations;
};
```

Les objets de la liste de déclaration sont appelés champs. Les champs peuvent être des vecteurs, des variables, des pointeurs, d'autres structures, etc qui doivent être déclarés.
Lors de la définition d'une structure, des variables dites structurées peuvent être déclarés, qui seront du type associé à la structure.
Exemple: dans une fonction:

```
struct nom s1,s2;
```

s1 et s2 sont des variables de type structure (variable dite structurée) dont le modèle est le nom de la structure les définissant.

Autre exemple:

```
struct nom
{
int n;
double x,y;
float tab[10];
int *pt;
}s1,s2;
```

12.2.2 typedef

typedef permet de donner un nouveau nom à un type existant.
Ainsi**: typedef struct nom s_enreg;**

s_enreg défini une structure s'appelant nom.

La déclaration des variables structurées s'écrira:

```
s_enreg s1,s2;
```

12.2.3 classes d'allocation

Les variables structurées suivent les mêmes règles que les variables. La portée dépend de l'emplacement de la déclaration de la variable structurée.

12.3 ACCES AUX MEMBRES(CHAMPS)

Seules les variables structurées permettent d'accéder aux champs par l'opérateur de champ: **.** placé entre le nom de la variable structurée et le champ.
Exemple:

s1.n=8; on affecte au champ n la valeur 8

Tous les opérateurs peuvent être utilisés: **s1.n++, &s1.n**

12.4 INITIALISATION DE STRUCTURES

Il s'agit de l'initialisation des champs appartenant à la structure. Cette initialisation se fait soit pour chaque champ comme précédemment soit par une déclaration statique ou globale des variables structurées. Dans ce cas les champs sont initialisés à 0.

12.5 EXEMPLES DE STRUCTURES

a)
```
struct point
{
int x[10];
int  y[10];
}l;
```
l.x[3] désigne le quatrième élément du tableau x[]

b)
```
struct point
{
char nom;
int x;
int  y;
};
 struct point l[10];
```
l est un tableau de 10 structures

12.6 STRUCTURE EN ARGUMENT DE FONCTION

12.6.1 par valeur

```
struct nom
{ int nb;
float valeur;
};
void main(void)
{struct nom x;              /*déclaration de la variable structurée*/
void fct(struct nom y);     /*structure en argument*/
----------------------
fct(x);                     /* passage à la fonction de la variable structurée x*/
}
void fct(struct nom s)
{
----------          /* fct travaille sur s, il n'y aura pas de modification des
                    champs dans main*/
}
```

12.6.2 par adresse

```
struct nom
{ int nb;
float valeur;
};
void main(void)
{struct nom x;              /*déclaration de la variable structurée*/
void fct(struct nom *);     /*pointeur sur une structure*/
----------------------
fct(&x);                    /* passage de l'adresse de la variable structurée x*/
}
```

Pour accéder aux champs dans la fonction à partir le l'adresse de la structure il faut faire appel à un nouvel opérateur: -> ou utiliser l'écriture pointeur **(*s).champ**

```
void fct(struct nom *s)
{
s->nb=8;            ou (*s).nb=8;
}
```

Les champs ayant même position mémoire, toute modification de leur valeur dans l'une ou l'autre des fonctions se répercute dans l'autre.

13 CHAINES DE CARACTERES

Il n'existe pas de variable de type chaîne
Une chaîne de caractères est un tableau de caractères (suite d'octets) codé en ASCII et terminé par un octet de code nul. Ainsi une chaîne de n caractères occupe n+1 octets.

13.1 INITIALISATION

Syntaxe: **char ch[10];**
ch="analyse";

ch est considéré comme une constante pointeur. Elle contient l'adresse de début de la constante chaîne: "analyse". |a|n|a|l|y|s|e|\o|

↑

ch

Autres syntaxes:

```
char ch[10]="analyse";

char  *ch;
ch="analyse";
```

On peut généraliser à un tableau de pointeurs:

```
char *jour[7]={"lundi","mardi",          ,     };
```

Ainsi jour[0]="lundi"

13.2 ENTREE-SORTIE DE CHAINES

code **%s** dans printf et scanf : printf("%s",ch); scanf("%s",ch);

Il existe des fonctions spécifiques telles que **gets(ch)** pour l'entrée et **puts(ch)** pour l'impression.
Pour entrer une chaîne caractère par caractère dans une boucle on peut utiliser les fonctions: **getchar()** et **putchar(ch[i]).** Ne pas oublier en entrée que le dernier caractère doit être de code nul soit: **'\0'.**

13.3 FONCTIONS DE MANIPULATION DE CHAINES

Ces fonctions se trouvent dans la librairie standard **string** d'ou nécessité de faire un lien avec le préprocesseur:

#include<string.h>

Toutes les fonctions commencent par **str** ainsi **strcat(ch1,ch2)** concatène ch2 à la suite de ch1. Attention il faut que l'emplacement réservé à ch1 soit suffisant.

14 FICHIERS

- Deux types de fichiers: binaire ou formaté.
- Deux types d'accès: séquentiel(flux) ou direct.

Création d'une mémoire tampon dans laquelle sont enregistrées les données avant d'être transférées de la mémoire centrale au fichier.

14.1 OUVERTURE ET FERMETURE D'UN FICHIER

14.1.1 déclaration

Syntaxe: **FILE * fichier_pt;**

FILE est une structure prédéfinie servant à déclarer la mémoire tampon dans <stdio.h>
fichier_pt est l'adresse de départ ou pointeur de flux.

14.1.2 ouverture

Associé un nom à la mémoire tampon et précisé son mode d'utilisation (lecture, écriture, lecture-écriture).

Syntaxe: **fichier_pt=fopen(nomfichier,mode);**

nomfichier est une chaîne de caractères précédemment déclarée.
mode est "wb" (écriture binaire) ou "rb" (lecture binaire)

fopen renvoie un pointeur de code NULL si le fichier ne peut être ouvert
if (fichier_pt==NULL) printf("impossible d'ouvrir");

14.1.3 fermeture

Après utilisation le fichier doit être fermé:

fclose(fichier_pt);

14.2 ECRITURE ET LECTURE BINAIRE

Syntaxe: **fwrite(&n,taille d'un bloc,nombre de blocs,fichier_pt);**
ou

fread(&n,taille d'un bloc,nombre de blocs,fichier_pt);

fwrite et fread fournissent respectivement le nombre de blocs écrits ou lus.

- n est la variable lue ou écrite
- taille d'un bloc est la taille en octets de la variable sizeof(n).
- nombre de blocs est le nombre de variables que l'on désire lire ou écrire
- fichier_pt est le pointeur de flux

14.3 FIN DE FICHIER

Syntaxe: **feof(fichier_pt)**

Retourne une valeur non nulle (vraie) si une fin de fichier a été trouvée sinon zéro (faux)

14.4 EXEMPLE DE PROGRAMME

```
#include<stdio.h>
void main(void)
{char nomfich[20];
int n;
FILE *entree;
printf("nom du fichier");
scanf("%20s",nomfich);
entree=fopen(nomfich,"rb");
if(!entree){printf("erreur d ouverture");
            exit();}
do{
fread(&n,4,1,entree);
if(!feof(entree))printf("\n%d",n);
}while(!feof(entree));
fclose(entree);
}
```

14.5 ECRITURE ET LECTURE FORMATEES

Fichier de type texte ou chaque octet représente un octet codé en ASCII

fprintf(fichier_pt,format,liste d'expressions);

fscanf(fichier_pt,format,adresses des expressions);

14.6 ACCES DIRECT

En mode séquentiel ,après chaque opération, le pointeur du fichier est incrémenté du nombre d'octets transférés, on peut également accéder au fichier en n'importe quel point: accès direct.

fseek(fichier_pt,taille*(num-1),par);

taille est le nombre d'octets de la variable
num numéro de la variable où on veut se placer. Exemple: 2*(num-1) pour des entiers courts.
par est un paramètre: 0 déplacement depuis le début du fichier
1 déplacement a partir de la position courante
2 déplacement à partir de la fin du fichier

fteel(fichier_pt) ;/*position courante du pointeur en octets*/

Les instructions suivantes donnent la taille d'un fichier:
fseek(fichier_pt,0,2);
taille=fteel(fichier_pt);

15 PREPROCESSEUR

Le préprocesseur exécute des fonctions particulières appelées "directives" et effectue un prétraitement du programme source avant sa compilation.
Les fonctions directives sont identifiées par le caractère # placé en tête. Ces fonctions peuvent contenir plusieurs lignes, chaque ligne à continuer étant terminé par le caractère \.

15.1 PRINCIPALES DIRECTIVES

15.1.1 #define

Permet de définir des pseudo constantes et des macros ou pseudo fonctions. Le préprocesseur remplace tous les mots du fichier source identique à la pseudo constante par la substitution.
Syntaxe: **#define pseudo_constante substitution**

Exemple: **#define N 100 /*N pseudo constante*/**

#define CARRE(x) (x)*(x) /*CARRE macro*/

15.1.2 #include

Permet d'insérer le contenu d'un fichier appelé en tête dans un autre. Ces fichiers sont en principe suffixés par **.h** .
Syntaxe: **#include <nom du fichier>**
#include "nom du fichier"

Le fichier entre guillemets est recherché dans le répertoire courant. Celui spécifié par <> est recherché dans le répertoire: **/usr/include.**
Quelques fichiers en tête:

stdio.h	(entrées sorties)
string.h	(chaînes de caractères)
time.h	(manipulation date et heure)
stdarg.h	(fonction avec un nombre variable d'arguments)
math.h	(fonctions mathématiques)

15.1.3 autres possibilités

Il existe des tests de compilation conditionnelle tels que **#ifdef** et **#endif** qui permettent de tester l'existence d'une pseudo constante.