SUPINFO
International University

Langage C

Essentiel

Objectifs :

Connaître l'histoire des langages de développement
Connaître le fonctionnement du langage
Ecrire ses premiers programmes en C

Version : 0.9
ID Booster : 664
Release Date : 05/09/2008

Sommaire

1. Notions d'informatique

Informatique traitement automatique de l'information.

Information → dans la vie courante et dans la théorie de l'information = levée de l'incertitude
→ en informatique : = données à traiter.
= instructions à suivre.
= résultats du programme.

Ordinateur système de traitement d'informations sous le contrôle d'un programme.

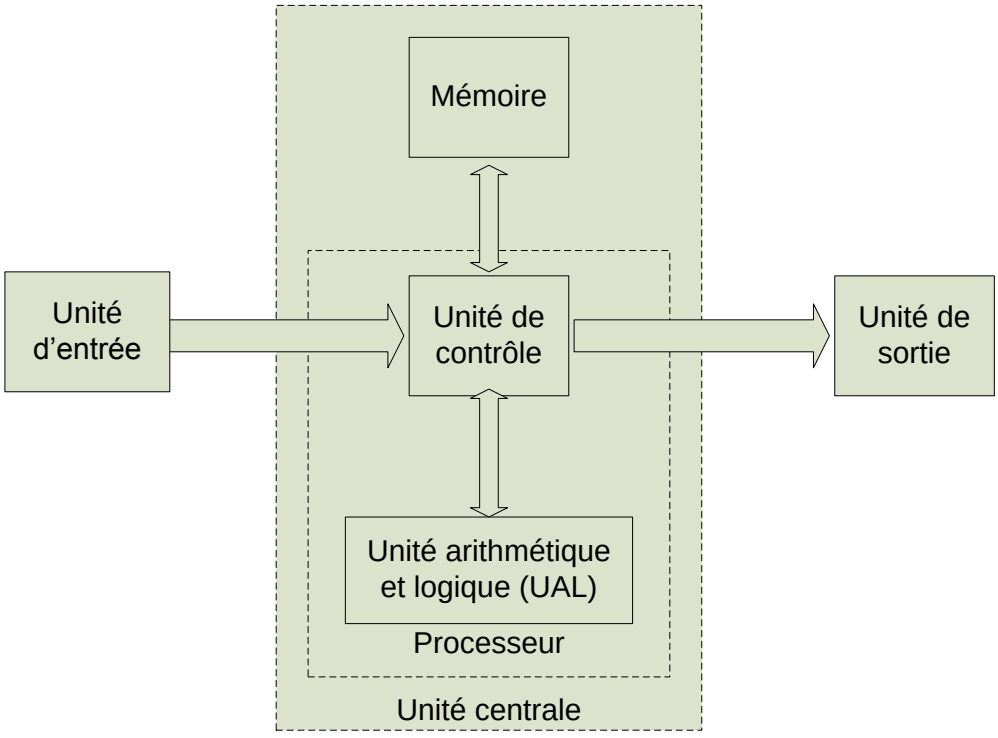
Programme représentation d'un algorithme en un langage de programmation.

Algorithme ensemble fini d'opérations qui mènent à la résolution d'un problème.

Hardware le matériel (circuits, dispositifs, équipements).

Software le logiciel (les programmes).

1.1. Description générale d'un ordinateur



Unité d'entrée = clavier, fichier, port → fait entrer les données et le programme.

Unité de sortie = écran, imprimante, fichier, port → reçoit les résultats du traitement.

Mémoire = centrale, opérationnelle, vive (accès rapide, capacité réduite) auxiliaire, de masse (accès plus lent, capacité importante).

UAL = Unité arithmétique et logique, circuits de traitement plus quelques registres de mémoire; exécute les instructions du programme.

Unité de contrôle = coordonne les différentes unités.

Le système d'exploitation (*operating system*) - SE, assure la coordination des activités de la machine. Ce sont des programmes (software) résidents dans la machine.

Un système d'exploitation se compose d'un SHELL (coquille) et d'un KERNEL (noyau).

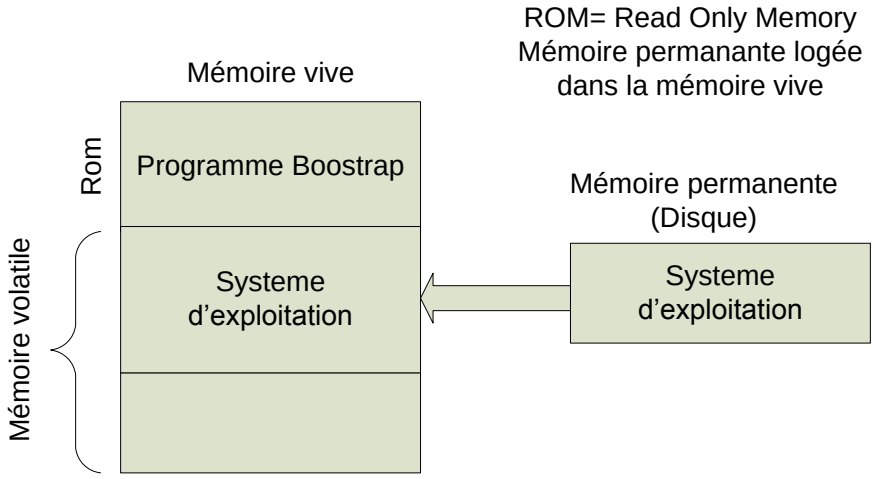
SHELL → interface entre le SE et l'utilisateur.

→ interface graphique - icônes (fichiers, programmes, etc.).

→ commandes avec la souris (mouse) ou avec les touches du clavier.

KERNEL → contient les programmes pour démarrer et pour coordonner.

1.2. Démarrage (bootstrapping ou booting)



Au démarrage le programme BOOTSTRAP s'exécute automatiquement et transfère le SE du disque vers la mémoire vive (volatile).

Une fois installé le SE peut coordonner :

- le placement et la recherche des fichiers (*file manager*)
- la communication avec les périphériques (*device drivers*)
- l'utilisation de la mémoire (*memory manager*)

1.3. La mémoire de masse (*mass storage*)

Stockage permanent des données.

1.3.1. Stockage sur disque magnétique (disquette ou disque dur)

Le nombre de tracés et de secteurs est déterminé par le formatage.

Disquettes (5 1/4 ou 3 1/2 inches) amovibles → 1.2 Mo.

Disque dur (5 à 10 disques sur le même axe) → 8 Go.

Vitesse d'accès (msec) > celle des circuits électroniques (nanosecondes) car il y a des parties mécaniques en mouvement.

1.3.2. Compact Disques (CD) - musique ou données

Traitement optique de l'information (laser).

CD-ROM (*compact disk read only memory*) avec information pré-enregistrée.

Environ 5 inches tracé en spirale qui donne une capacité d'environ 600 Mo.

1.3.3. Bande magnétique

Grande capacité (Go) mais temps d'accès long dû au déroulement de la bande.

1.4. Le stockage des données

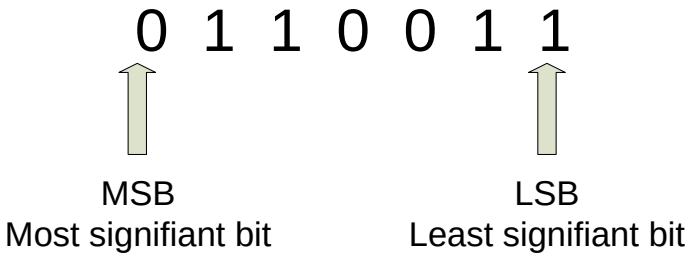
L'information dans la machine est représentée par des suites de bits (*binary digits*) de valeur 0 et 1.

Dispositifs électriques ou magnétiques à deux états : bascules (flip flop), portes logiques, etc.

Le stockage machine se fait sur la mémoire vive (*main memory*) ou sur la mémoire de masse (*mass storage*).

- La mémoire vive (ou « mémoire »):
  Suite de cellules à 8 bits (octet ou byte)
  Chaque cellule a une adresse.
  L'accès à chaque cellule est indépendant de sa place : *Random Access Memory* (RAM) - accès aléatoire.
  Le contenu d'une cellule (8 bits):

  Ex :



  Capacité de la mémoire = nombre de cellules (nombre d'octets).
  Exemple:
  1024 cellules = 1024 octets = 1 Ko = 2^{10} octets.
  4192304 cellules = 4 Mo (Mégaoctets) = 2^{20} octets.
  1024 Mo = 1 Go (Gigaoctet) = 2^{40} octets

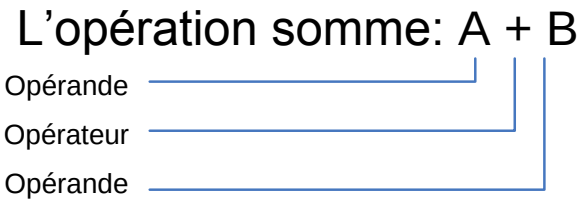
1.5. Langages accessibles à l'ordinateur

1.5.1. Langage machine

Codage des instructions en binaire.

Instruction = opération élémentaire composée d'un code pour l'opération à effectuer ainsi que des adresses des opérandes.

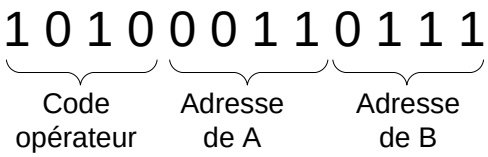
Exemple :



Code de l'addition : 1010

Adresse de A : registre 3 (0011 en binaire)

Adresse de B : registre 7 (0111 en binaire)



Inconvénients:

- Difficultés de programmation en binaire surtout pour des instructions complexes.
- Lié au type de machine (non transportable).

Avantages :

- Utilisations maximales des potentialités de la machine (vitesse et occupation mémoire).

1.5.2. Langage assembleur (symbolique)

Codage symbolique (et non pas binaire) des instructions.

Codes opérations faciles à retenir (ex : ADD pour addition).

Adresses en base 10 ou 16 (ou bien un nom symbolique).

Utilisation d'étiquettes pour les « ruptures de séquences » (choix de plusieurs alternatives)

Exemple : l'instruction « additionner A et B » devient :

ADD A B

Nécessité d'un programme de décodage qui traduit les instructions en langage machine.

Avantages :

- Plus simple à manipuler (aussi bien pour écrire que pour modifier et déboguer).
- Exploite bien les potentialités de la machine.

Inconvénients:

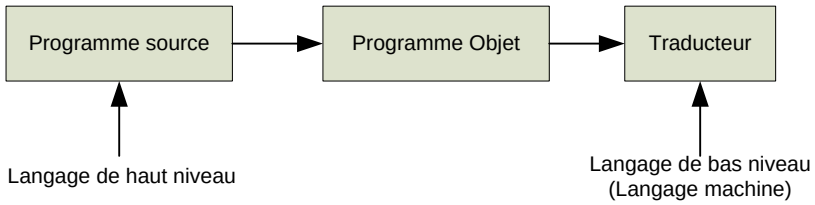
- Lié au type de machine (non transportable)
- Oblige à décomposer le programme en instructions élémentaires.

L'évolution des langages de programmation s'est faite dans deux directions principales :

- Libérer le programmeur des contraintes machine.
- Approcher le langage de l'algorithme de résolution et donc du raisonnement.

1.5.3. Langages de programmation

- Plus proches des langages naturels (haut niveau).
- Rigueur syntaxique → traduction sur différentes machines.
- Gain de productivité dû aux temps d'écriture plus réduit, ainsi qu'à une portabilité accrue et une meilleure lisibilité.



Traducteur :

- Compilateur sert à la traduction du programme en bloc et sert aussi à rendre le programme exécutable (indépendant de la machine).
- Interpréteur - l'exécution est liée à la machine (instruction par instruction). Il est plus lent mais interactif.

1.6. Classification des langages

1.6.1. Selon le domaine d'utilisation

Langages :

- orientés problèmes scientifiques (Fortran & Algol)
- orientés problèmes de gestion (Cobol)
- universels (Pascal, Ada, C)
- orientés objet (C++, Smalltalk)
- pour L'IA (Prolog, Lisp)

1.6.2. Selon la structure interne

Langages :

- procéduraux (C, C++, Pascal, etc.)

  Le programme est une suite de procédures (instructions)
- déclaratifs - logiques (Prolog)

  Le programme est une suite de propositions logiques
- fonctionnels (Lisp)

  Le programme est une suite de fonctions

1.6.3. Selon la chronologie d'élaboration

1ère génération (1950-1960) :	Fortran, Cobol (types de variables simples et peu de structures de contrôle.
2ème génération (1960 – 1970) :	Algol (structures de contrôle évoluées) et Lisp (fonctionnel).
3ème génération (1970 – 1980):	Pascal, C (1973) (types de variables définissables par le programme et types de variables évoluées)

1978: *The C reference manual* (B. Kernighan, D.M. Ritchie)

Le C est né en 1973 de la main de Dennis Ritchie sous le système d'exploitation Unix (dont il est un des concepteurs). Le rôle initial de ce langage était de faciliter le développement d'Unix mais il a rapidement conquis d'autres systèmes d'exploitation tant il était portable et performant. Par la suite, il fut normalisé par l'ANSI afin de permettre son portage sur différents systèmes.

Aujourd'hui encore, ce langage reste l'un des plus performants et plus utilisés, notamment dans le cadre du développement des systèmes d'exploitation (Windows, Unix, OS2 ...).

1982: C++ (Bjarne Stroustrup)

1989: C (ANSI) - American National Standard Institue.

2. Structure d'un programme en langage C

2.1. Introduction au langage C

Le C fait partie de la famille des langages procéduraux. De tels langages permettent de fonder l'écriture d'un programme sur l'algorithmique.

Dans ce type de langage, la donnée passe au second plan. L'idée générale c'est le traitement des données initiales par appels successifs de procédures (= fonctions) jusqu'à obtenir le résultat souhaité.

Le langage C est à la fois :

Très puissant et très permissif ce qui comporte des avantages et des inconvénients; il offre un champ d'action très large qui permet de faire tout type de programmes presque aussi rapides qu'en assembleur mais il ouvre en même temps la porte à de nombreuses erreurs.

Il est structuré mais proche de la machine, c'est un langage de haut niveau. Il peut parfois paraître compliqué et illisible.

2.2. Exemple

```
/* Programme SPHERE.C */                          /*  et  =  commentaires*/
#include <stdio.h>                                /* directive de compilation */
#define PI 3.14                                   /*directive de compilation */

main ()                                           /* fonction principale */
{
      float volume, r;                            /* déclaration des variables */
      printf("entrez le rayon:");                       /* appel à printf */
      scanf("%f", & r);                           /* appel à scanf */
      volume = r * r * r * 4 * PI / 3;
      printf("le volume est : %f \n", volume);
}
```

main :

- fonction principale de tout programme C.
- marque le début et la fin de l'exécution du programme.
- le corps de ***main*** est entre 2 accolades {}. La partie entre les accolades est appelée un bloc. Il est conseillé d'indenter le code d'un programme pour le rendre plus lisible et ainsi éviter de nombreuses erreurs. Pour indenter, il suffit d'ajouter une tabulation après le retour à la ligne suivant une accolade ouvrante et de retirer une tabulation au placement d'une accolade fermante.
- d'abord les déclarations des variables, - ensuite l'appel aux fonctions (***printf***, ***scanf***).
- le résultat du calcul est affecté à la variable **volume** et affiché.

Les Instructions :

- Une instruction se termine toujours par un point virgule.
- Un groupe d'instructions (un bloc) se trouve entre accolades.
- Règles qui améliorent la lisibilité :
  - Une instruction par ligne.
  - Décaler les instructions par rapport aux accolades.
  - Aligner les accolades verticalement.

Directives de compilation :

- commandes au compilateur précédées par le signe # (dièse).
- **#include** demande au compilateur de lire et de compiler un fichier.
- STDIO.H (standard input output. header) = fichier « en-tête » = bibliothèque C avec des fonctions utiles (ex: ***printf***)
- **#define** définit une constante symbolique (ex #define PI 3.14). Le compilateur remplace chaque occurrence de PI par 3.14.
- ***printf*** (fonction standard de bibliothèque (STDIO.H)) affiche à l'écran un texte et des variables.
- ***scanf*** (fonction standard de bibliothèque (STDIO.H)) entre dans le programme des données saisies à l'écran (ex: la valeur de r).

Le programme est *compilé* (traduit en langage machine), *lié* aux fichiers « header » et *exécuté*.

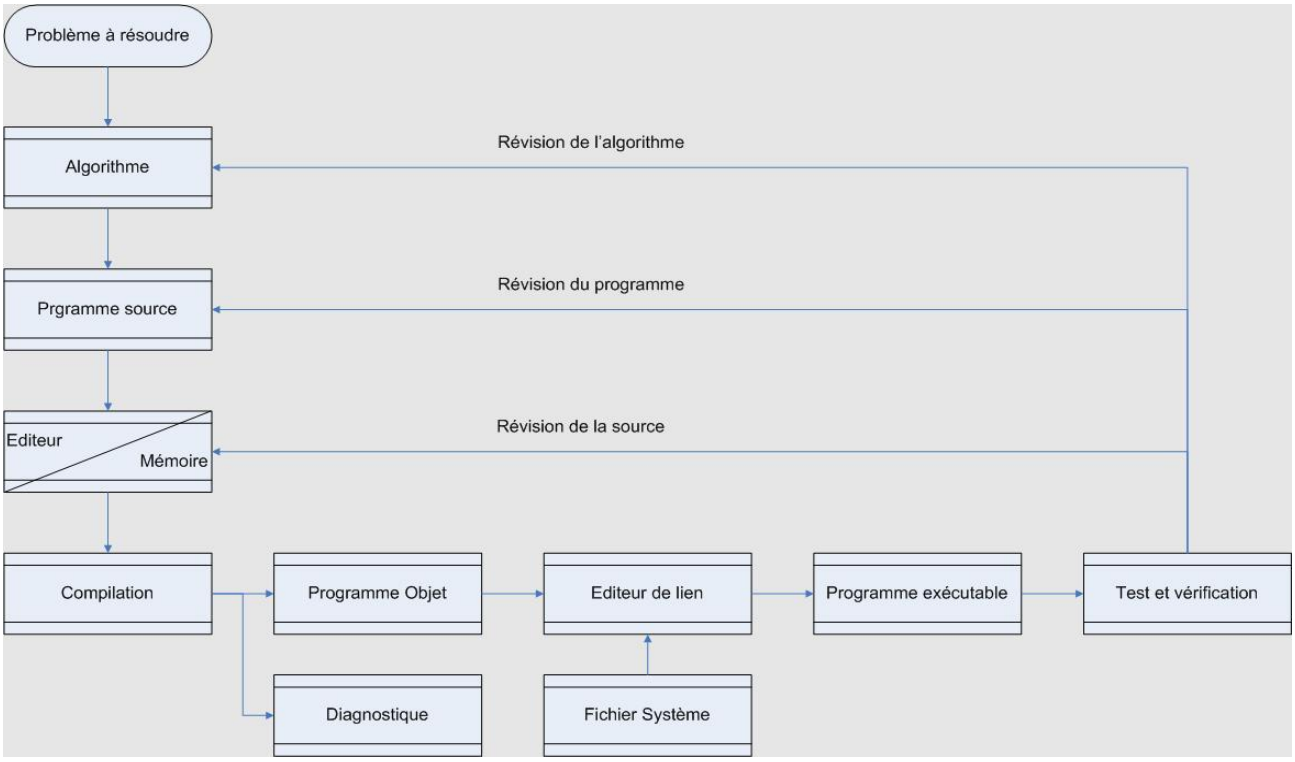
2.3. Etapes concernant la solution d'un problème à l'aide d'un ordinateur

1. Le projet une fois défini, l'analyste établit :
   - les données nécessaires.
   - le traitement à effectuer (algorithme, ordinogramme).
   - la forme requise pour les résultats.
2. Elaboration du programme : traduction de l'algorithme dans un langage de haut niveau.
3. Entrée du programme dans l'ordinateur :
   - saisie (clavier) (visualisation dans l'éditeur et mise en mémoire vive).
   - sauvegarde (fichier disque).
4. Compilation:
   - traduction et allocation de mémoire. Le programme source devient un programme objet.
   - détection et indication des erreurs :
     i. type de l'erreur.
     ii. instruction incorrecte.
5. Edition des liens:
   - relie le fichier « objet » aux fichiers système (bibliothèque).

- création du programme exécutable (portable) sauvegardé sur le disque.

6. Test du programme avec des données concrètes.
7. Révision si nécessaire.

2.4. Les étapes concernant la solution d'un problème



3. La compilation

3.1. Etapes

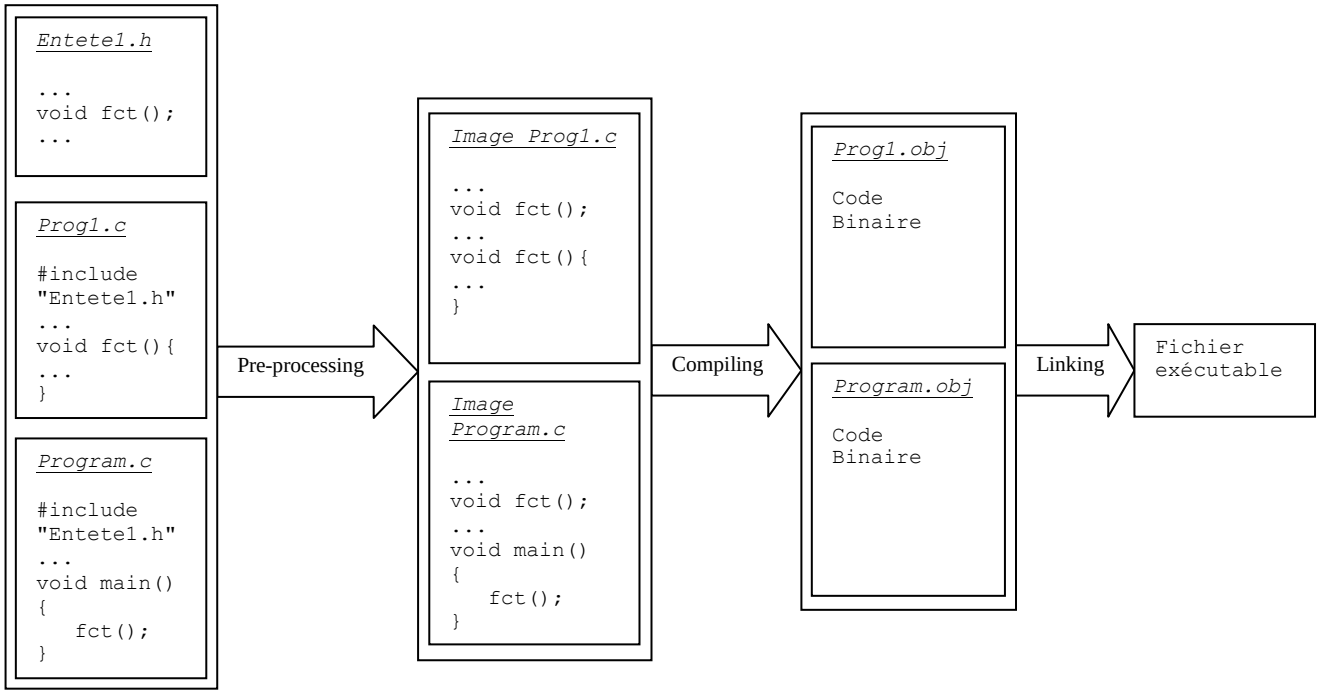
Le C est un langage compilé. On ne dispose, au départ, que de fichiers sources.

Dans un premier temps, les fichiers doivent être traités avant la compilation. An appelle cette étape *pre-processing* en anglais et elle est effectuée, par le préprocesseur. C'est dans cette étape que les directives du préprocesseur (#include, #define, …) sont interprétées.

Dans un deuxième temps, chaque fichier source généré par le préprocesseur est compilé indépendamment. Cette étape, réalisée par le compilateur, fabrique des fichiers objet qui contiennent la traduction en langage machine du fichier source correspondant.

Enfin, dans la dernière étape, appelée édition de liens (*linking* en anglais), on regroupe les différents fichiers objet en résolvant les références entre ceux-ci. Le fichier obtenu est exécutable par la machine.

Le schéma ci-dessous résume le procédé décrit :



3.2. Contraintes

Afin de compiler les fichiers intermédiaires créés par le préprocesseur, il est impératif que les entités suivantes soient être déclarées avant leur utilisation :

- les structures
- les variables
- les fonctions

C'est l'éditeur de liens, dans la dernière étape de la compilation, qui se charge de faire correspondre les définitions de ces entités, aux endroits ou elles sont utilisées.

Pour déclarer les fonctions sans les définir, on utilise les prototypes. Les structures peuvent être déclarées de la manière suivante :

```
class CMaClasse;
```

Par défaut, une variable globale n'est accessible que dans le fichier source dans lequel elle est déclarée, car chaque fichier source va être compilé dans un fichier objet indépendant.

Cependant, il est possible de rendre une variable globale accessible dans tous les fichiers source d'un programme grâce au mot clé **extern**. Cette pratique, est toutefois à éviter.

```
extern UneVariableGlobale;
```

4. Les types de données

Toute variable doit avoir un type. En C il existe 4 types de base : **char**, **int**, **float**, **double** auxquels on ajoute des qualificatifs et avec lesquels on forme des types dérivés (ou agrégats). Les qualificatifs de type qualifient les types de données de base: **short**, **long**, **signed**, **unsigned**.

4.1. Le type *int*

Il représente des nombres entiers (1, 5, -327).

Déclaration : **int** <nom_de_variable>

Nombre d'octets alloués en mémoire (rappel : 1 octet = 8 bits) :

2 octets pour **int** et **short_int** (16 bits) (dépend du compilateur).

4 octets pour **long_int** (32 bits).

signed int est la valeur signée (négative ou positive).

unsigned int représente des valeurs positives ou nulles.

Par défaut une valeur **int** est signée.

Nom de type	***Autre Nom***	***Intervalle de valeur***	***Octets***
int	signed, unsigned int	-32 768 à 32 767	2
short	short_int, signed short signed short_int	-32 768 à 32 767	2
long	long_int, signed long, signed long_int	-2 147 493 648 à 2 147 483 647	4
unsigned	unsigned int	0 à 65 535	2
unsigned short	unsigned short_int	0 à 65 535	2
unsigned long	unsigned long_int	0 à 4 294 967 295	4

4.2. Le type *float*

Il désigne les nombres décimaux réels (positifs ou négatifs) (à virgule flottante).

Déclaration :

float <nom_de_variable>

Nom de Type	***Autre Nom***	***Intervalle***
Float	-	$-3.4^{E}38$ à $+3.4^{E}38$
double		$-1.7^{E}308$ à $+1.7^{E}308$
long double		$-1.7^{E}308$ à $+1.7^{E}308$

Exemple

```
main ()
{
	int a, b;
	float quotient;			/* on aurait aussi pu mettre float a, b */
	printf("entrez deux nombres: ");
	scanf("%d %d", &a, &b); /* on aurait aussi pu mettre scanf("%f %f", &a, &b)*/
	quotient = a / b;
	printf("Le quotient est %f\n",quotient);
}
```

Il faut aussi remarquer que chaque instruction est suivie d'un point virgule (;). Si l'on divise 10 par 3 le résultat est 3.000000 (et respectivement 3.333333 dans le cas des variables de type **float** - la conversion en **float** se fait après la division).

4.3. Le type *char*

Il désigne les caractères.

Déclaration :

char <nom_var>

Nom de Type	***Autre Nom***	***Intervalle***	***Octets***
Char	signed char	-128 à 127	1
unsigned char		0 à 255	1

- Les types de données de base ne contiennent qu'une seule valeur.
- Les types dérivés ou agrégats peuvent contenir plus d'une valeur.

  Exemple: chaînes, tableaux, structures, énumérations, unions, pointeurs.

4.4. Conversion implicite de type

C admet le mélange de types de variables.

Le compilateur convertit le rang inférieur vers le rang supérieur (= promotion)

Hiérarchie des types :

char < short < int < long < float < double.

Exemple:

```
/*CONV.C:Exemple des conversion de type*/
#include <stdio.h>
#include <conio.h>

void main()
{
   char val_c = 10;
   int val_i = 20;
   long val_l = 64000;
   float val_f = 3.1;
   int resultat ;
   resultat = val_c + val_i + val_l + val_f;
   printf("%f\n", resultat);
}
```

Affiche -1503 (au lieu de 10+20+64000+3.1= 64033.10)

Il y a trois « promotions » :

char → int(30), int → long(64030), long → float(64033.10) .

La variable 'resultat' de type **int** (2 octets) ne peut contenir une variable de type **float** (4 octets). Il s'effectue alors une « rétrogradation » de **float** vers **int** qui entraîne une perte de données car le type supérieur est tronqué (octets rejetés).

4.5. Les Constantes

En C il existe 4 types de base:

1. Les constantes entières.
2. Les constantes en virgule flottante (constante réelle).
3. Les constantes caractères.
4. Les constantes chaînes de caractère.

Les constantes n°1 et 2 sont des constantes numériques.

4.5.1. Valeur numérique entière

- système numérique décimal ex: 1, 289, 9999.
- système numérique octal ex: 01, 0273, 0777.
- système numérique hexadécimal ex: 0x7DF, 0X7DF

4.5.2. Valeur numérique en base dix

- un point décimal ex: 0.2 ou bien 927.308
- un exposant ex: $2^{E}-6 = 2*10^{-6}$ ou bien $0.06^{E}+3 = 0.06*10^{3}$

La précision dépend des compilateurs:

- min: 6 chiffres significatifs.
- max: 18 chiffres significatifs.

4.5.3. Caractère unique entre apostrophes

'A', 'x', 'Z', '? ', ' '

Equivalent numérique (code ASCII : American Standard Code for Information Interchange) sur 7 bits donc 2^7 = 128 caractères.

Exemple:

Caractère	: 0…9	A…Z	a…z
Code ASCII	: 48…57	65…90	97…122

Séquence d'échappement :

Caractères non affichables formés d'un *backslash* (\ ) et d'un ou plusieurs caractères.

Exemple :

\n = retour à la ligne.

\t = tabulation.

\a = alarme (un bip).

\0 = caractère nul (code ASCII 000);

Attention \0 est différent de 0 (ASCII 48)

\' = l'apostrophe.

\\ = le *backslash*.

\" = les guillemets.

\x41 ou \X41 = A en séquence d'échappement hexadécimale.

4.5.4. Suite de caractères entre guillemets (chaînes)

Exemple :

"bonjour", "125.00FF", "2*(I+3)/J",

"A \n B \n C" → comprend 2 sauts de ligne.

"" → la chaîne vide.

" " → espace

Le compilateur place automatiquement le caractère nul '\0' en fin de chaîne (invisible).

"bonjour" est en fait "bonjour\0" ce qui est utile dans les programmes pour marquer une fin de chaîne.

'A' est différent de "A" car 'A' est un caractère avec la valeur ASCII 48 alors que "A" est une chaîne ("A\0") sans valeur numérique. "A" occupe plus de place que 'A'.

4.5.5. Constante symbolique

Nom associé à une constante.

Syntaxe: **#define** nom_chaine

Exemple:

```
#define PI  3.141593      // (pas de ;)
#define AMIE " Julie "
```

4.6. Les variables

Une variable est un espace réservé ou l'on peut lire et écrire une donnée. Afin de savoir ce qu'elle peut contenir, on va lui donner un *type*. Il existe plusieurs types de base prédéfinis par la norme du C mais vous verrez plus tard que vous pouvez aussi créer les vôtres. Voici quelques exemples de déclarations de variables de différents types :

Identificateur pour une valeur numérique et pour une constante caractère.

Déclaration : association de la variable à un type.

Exemple :

```
int a; int b;
int a, b;
char d; float r1,r2; long r, t ;
```

Affectation de valeur à une variable

Valeur initiale (initialisation)

Exemple :

```
int a;
a = 5;
int a = 5;
```

En cours de programme leur valeur peut changer.

4.7. Les tableaux

Variable qui identifie un ensemble de données du même type (ex: ensemble de caractères).

```
char msg[ ] = "bonjour";
char msg[8] = "bonjour";   // le 8 correspond au nombre de lettres du mot
                           // bonjour plus le '\0'.
```

4.8. Les Instructions

Le code écrit de l'action à faire par l'ordinateur.

Il y a plusieurs types d'instructions :

- simple ;
- composée {}
- de contrôle {}

4.8.1. Simple

```
A = 3;  c = a + b; i++;  printf ("oui");
```

4.8.2. Composée

```
{
	pi = 3.141592;
	circonference = 2 * pi * rayon;
	surface = pi * rayon * rayon;
}
```

4.8.3. De contrôle

Toutes les boucles, branchements, tests logiques, boucles **while** sont considérées comme étant des instructions de contrôle.

```
while(compteur <= n)
{
	printf("compteur = %i", compteur);
	++compteur;
}
```

Pour afficher nos variables nous allons utiliser une nouvelle fonction, ***printf()***. Cette fonction est très utilisée mais peut cependant surprendre lors de ses premières utilisations.

5. Entrées et sorties

L'échange d'information entre l'ordinateur et les périphériques tels que le clavier et l'écran est une étape importante que tout programme utilise. En C, les bibliothèques de fonctions comprises dans les fichiers d'en-tête (header), sont incluses dans le fichier source avec la directive: **#include**.

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

stdio.h correspond au fichier d'en-tête Standard Input-Output et le ".h" le définit comme étant un header.

5.1. Fonctions usuelles comprises dans stdio.h

printf (…) : fonction **de sortie**. - écrit à l'écran les données fournies par l'ordinateur.

scanf (…) : fonction **d'entrée** (dans le programme) des données saisies à l'écran (lues).

getchar(…) : lit un caractère en entrée.

putchar (…) : affiche un caractère à l'écran.

gets (…) : lit une chaîne en entrée.

puts (…) : affiche une chaîne à l'écran.

5.2. Sortie sur écran avec *printf*

5.2.1. Syntaxe

```
printf (<chaine_de_format>, <arg1>, <arg2>, ..., <argn>)
```

La chaîne_de_format comprend les paramètres d'affichage.

Paramètres d'affichage : symbole % + spécificateur de format (caractère indiquant le type de la donnée) arg1, arg2,…, argn sont les données à afficher.

5.2.2. Exemples de spécificateurs de format

%c : affiche un caractère unique

%d ou %i : un entier signé sous forme décimale

%f : affiche une valeur réelle avec un point décimal.

%e ou %E : affiche une valeur réelle avec un exposant.

%x ou %X : affiche un entier hexadécimal.

%u : affiche un entier en notation décimale non signée.

%s : affiche une chaîne de caractères (string).

%g ou %G : affiche une valeur réelle avec affichage de type e ou f selon la valeur.

5.2.3. Largeur minimale de champs

%4d : 4 digits "au moins" réservés pour l'entier.

%.2f : précision de 2 rangs décimaux.

Les arguments de ***printf*** sont :

Des constantes.

Des variables.

Des expressions.

Des appels de fonctions.

1°)

```
#include <stdio.h>
#include <math.h>          //pour sqrt ()
main ()
{
       float  i = 2.0, j = 3.0
       printf ("%f %f %f %f", i, j, i+j, sqrt(i+j));
}
```

Ici i+j est une expression et ***sqrt*** (i+j) est un appel de fonction.

Le programme affiche :

2.000000 3.000000 5.000000 2.236068

%f affiche donc par défaut avec une précision de 6 chiffres significatifs.

2°)

```
#include <stdio.h>
#include <math.h>          //pour sqrt ()
main ()
{
       printf ("Le produit de %d par %d est %d.",6 ,7, 42);
}
```

Affiche :

Le produit de 6 par 7 est 42.

3°)

```
#include <stdio.h>
main()
{
       int i = 12345;
       float x = 345.678;
       printf("%3d %5d %8d \n", i, i, i);
       printf("%3f %10f %10.1f ", x, x, x);
}
```

Affiche :

12345 12345 _ _ _12345

345.678 _ _ _345.678 _ _ _ 345.6

5.3. Autres Fonctions de sortie

puts : écrit une chaîne de caractères suivie d'un saut de ligne.

puts ("bonjour") est équivalent à ***printf*** ("bonjour \n")

putchar : écrit un seul caractère (sans \n).

putchar (a) est équivalent à ***printf*** ("%c",a)

5.4. Entrée sur clavier avec *scanf*

scanf est une fonction définie dans stdio.h

5.4.1. Syntaxe

```
scanf (<chaîne de format>, <adresse 1>,<adresse 2>,…);
```

Même chaîne de format que ***printf*** : %<lettre>

Après la chaîne de format ***scanf*** n'accepte que des **adresses**.

Exemple :

```
main()
{
	int a, b;
	float quotient;
	printf("Entrez 2 nombres:");
	scanf("%d  %d", &a, &b);
	quotient = a / b;
	printf("Le quotient est %f \n", quotient);
}
```

L'opérateur d'adresse "&" passe les adresses de a et de b à ***scanf***. On écrit à l'écran deux nombres séparés par un espace blanc.

Si virgule entre les spécificateurs : ***scanf*** ("%d ,%d", &a, &b); => on écrit deux nombres séparés par une virgule.

Entre le pourcentage (%) et la lettre (d, f, s, x, e) on peut inclure d'autres spécifications:

Exemple :

la largeur du champ décimal

%4d → 3496|21 (seuls les 4 premiers digits sont lus)

5.4.2. Entrée d'une chaîne avec *scanf*

Exemple :

```
main()
{
	char nom[30];
	printf ("Quel est votre nom ? ");
	scanf ("%s", nom);
	printf ("Hello %s\n", nom);
}
```

"nom" est un tableau de caractères.

La valeur de "nom" est l'adresse même du tableau. On n'utilise donc pas l'opérateur "&" devant "nom".

Le problème est que si l'on tape deux noms (Exemple : Anne Dupont) seul "Anne" s'affiche car "scanf" s'arrête au 1er espace blanc.

5.4.3. Solution avec *gets*

La fonction ***gets*** est l'abréviation de GET String, elle permet de récupérer une chaîne de caractères saisie.

Exemple :

```
#include <stdio.h>
main()
{
	char nom[10];
	printf ("Quel votre nom ? ");
	gets (nom);
	printf ("Hello %s\n", nom);
}
```

La fonction ***gets*** lit tout jusqu'à validation avec la touche Entrée.

La fonction ***getch***() lit un caractère unique .

Elle renvoie le caractère lu.

Exemple :

```
char c;
c = getch ();   // affectation à la variable c du caractère lu
```

6. Opérateurs

Ils servent à manipuler les données entrées avec le programme.

Il y a plusieurs types d'opérateurs:

- arithmétiques
- relationnels
- d'affectation (assignement)
- d'incrémentation / décrémentation
- logiques
- conditionnels
- de calcul et d'adresse d'indirection
- évaluation séquentielle

La plupart sont « binaires », c'est-à-dire qu'ils agissent sur 2 opérandes.

6.1. Opérateurs arithmétiques

Il existe plusieurs opérateurs arithmétiques que voici :

Opérateur	Rôle
+(unaire)	Identité
-(unaire)	Opposé
+(binaire)	Addition
-(binaire)	Soustraction
*(binaire)	Multiplication
/(binaire)	Division
%(binaire)	Modulo

L'instruction :

reste_de_la_division = 20%3 affecte la valeur 2 à la variable "reste_de_la_division".

Le type du résultat dépend du type des opérandes, par exemple, si on fait une division entre deux variables de type **int** (entier), le résultat sera un entier. En revanche, si l'un des deux opérandes est un réel, le résultat sera un réel.

```
int a = 10, b = 11;
float c = 11, res1, res2;
res1 = b / a;
res2 = c / a;
printf("res1 = %f \nres2 = %f\n", res1, res2);
/* affichera :
res1 = 1.000000
res2 = 1.100000 */
```

6.2. Opérateurs relationnels

Comme leur nom l'indique, ces opérateurs nous donnent la possibilité de comparer les valeurs de deux expressions. Le résultat de ces opérateurs est 1 pour vrai et 0 pour faux.

Opérateur	Rôle
<(binaire)	Strictement inférieur à
<=(binaire)	Inférieur à
>(binaire)	Strictement supérieur à
>=(binaire)	Supérieur à
==(binaire)	Égal à
!=(binaire)	Différent de

Toute valeur non nulle est considérée comme vraie.

```
/* Programme Vrai.c*/
#include <stdio.h>

main ()
{
	printf("C évalue %d comme vrai\n", 2==2);		/* 2==2 renvoie 1*/
	printf("C évalue %d comme faux\n", 2==4);		/* 2==4 renvoie 0 */
	if (-33)
	printf("C évalue %d comme vrai\n", -33);
}
```

Résultat à l'écran:

C évalue 1 comme vrai.

C évalue 0 comme faux

C évalue -33 comme vrai

6.3. Opérateur d'affectation et opérateurs d'affectation composés

6.3.1. Opérateur d'affectation (=)

Nous avons déjà vu l'opérateur principal d'affectation : '='. Il est cependant possible de le préfixer avec la plupart des opérateurs binaires. Ceci à pour effet d'effectuer d'abord le calcul entre une expression et la valeur actuelle d'une variable puis de modifier cette variable avec le résultat obtenu.

Exemple :

```
int val = 5 ;
```

= est associé aux opérateurs arithmétiques et au niveau du bit

6.3.2. Opérateurs d'affectation composés

Expression	***Equivalence***	***Fonction***

x + = y	x = x + y	Addition
x - = y	x = x - y	Soustraction
x * = y	x = x * y	Multiplication
x / = y	x = x / y	Division
x % = y	x = x % y	Reste de la division ou Modulo

Dans une affectation composée, le signe égal (=) est toujours précédé des autres signes.

```
val ^= exemple              /* correct */
val =^ exemple              /* incorrect */
```

6.4. Opérateurs incrémentaux

Opérateurs unaires qui ajoutent (++) ou qui retranchent (--) 1 à une valeur.

Opérateur	Rôle
++ (unaire)	Incrémenter
−− (unaire)	Décrémenter

Ils se placent avant ou après la variable associée:

val++ et ++val signifient que val = val+1

val-- et --val signifient que val = val -1

Si on place ces opérateurs avant la variable, celle ci est incrémentée (ou décrémentée) **avant** l'utilisation de sa valeur. Si on place ces opérateurs après la variable, celle ci est incrémentée (ou décrémentée) **après** utilisation de sa valeur.

```c
#include <stdio.h>

main ()
{
	int res, a = 3, b = 3;
	res = a++;
	printf("res=%d a=%d \n",  res, a);
	res = --b;
	printf("res=%d b=%d\n", res, b);
}
```

6.4.1. A l'écran

res =3 a=4

res =2 b=2

6.4.2. Explications

res = a++ → affecte la valeur de a (3) à 'res' puis incrémente 'a'

res = --b → décrémente d'abord b, puis affecte la valeur à 'res'.

```c
main()
{
	int a, b, res;
	char *format;
	format = "a=%d  b=%d  res=%d\n";	/*le pointeur "format"
						 pointe vers une chaîne de caractères*/
	a = b = 5;
	res = a + b;	printf (format, a, b, res);
	res = a++  + b;	printf (format, a, b, res);
	res = ++a  + b;	printf (format, a, b, res);
	res = --a +  b;	printf (format, a, b, res);
	res = a-- +  b;	printf (format, a, b, res);
	res = a +  b;	printf (format, a, b, res);
}
```

Avant d'exécuter le programme, calculez les résultats.

6.5. Opérateurs logiques &&, || et !

Les opérateurs logiques comparent des expressions qui sont considérées fausses si leur valeur est 0 et vraies dans tous les autres cas. Le résultat de ces opérateurs est comme précédemment 1 pour vrai et 0 pour faux.

Opérateur	Rôle
&&(binaire)	Et logique
||(binaire)	Ou logique
!(unaire)	Négation logique

ET et OU sont utilisés dans les instructions conditionnelles.

```
if (a>10 &&  b<5)          /* 2 conditions à satisfaire */
      printf ("c'est correct!\n");
```

Les opérateurs relationnels (> et <) ont la priorité sur le ET logique (ils sont évalués en premier) - voir tableau des priorités.

NON - inverse la valeur logique d'une expression (1 -> 0 et 0 -> 1)

```
main()
{
      int val = 0;
      if (!val)                                  /* (!val) équivaut à 1 */
            printf ("val est zéro");
}
```

6.6. Opérateur d'adresse mémoire

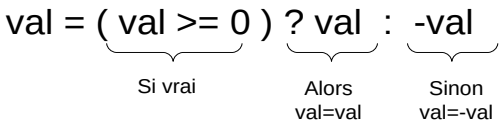
- L'opérateur de calcul d'adresse (&) retourne une constante qui est égale à l'adresse machine de l'opérande.
- L'opérateur d'indirection (*) retourne la valeur contenue dans l'adresse de l'opérande.

(Les deux sont utilisés avec les pointeurs).

4.6 Opérateur conditionnel (? :)

Expression conditionnelle à évaluer: si vraie <action1> si non<action2>

équivaut à l'instruction conditionnelle IF - ELSE :




```
if (val >= 0)
       val = val;
else
       val = -val;
```

6.7. Opérateur 'sizeof'

Retourne le nombre d'octets contenus dans son opérande.

```
/* Programme TAILLEDE.C */

#include <stdio.h>
char chaine[ ] = "Bonjour !";
main()
{
       printf ("Un entier contient %d octets \n", sizeof (int));
       printf ("La chaine contient %d octets \n",sizeof (chaine));
}
```

Affiche :

Un entier contient 2 octets

La chaîne contient 10 octets (9 caractères + le caractère de fin de chaîne '\0')

6.8. Opérateur d'évaluation séquentielle

La virgule (,) peut être :

a) signe de ponctuation (comme ';')

b) opérateur d'évaluation séquentielle.

Dans a) la virgule sépare les arguments d'une fonction ou les déclarations d'une variable.

Dans b) la virgule sépare des expressions faisant partie d'une opération unique (ex: expressions de la boucle **for**).

6.8.1. Exemple 1

Programme VIRGULE.C (permutation de valeurs)

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

```
main ()
{
	int val = 5, val1 = 666, temp;
	temp = val, val = val1, val1 = temp;
	printf ( "val= %d val1= %d \n", val, val1);
}
```

Résultat : val= 666 val1= 5

6.8.2. Exemple 2

Programme EXPMULT.C (expressions multiples dans une boucle FOR)

```
#include <stdio.h>

main ()
{
	int a, b;
	for (a = 256, b = 1; b < 512; a /= 2, b *= 2)
		printf ("a= %d \t b= %d \n", a, b);
}
```

6.9. Ordre de priorité des opérateurs

Le plus haut niveau de priorité est situé en haut du tableau. L'associativité régit l'ordre de priorité entre les opérateurs d'un même groupe de priorité.

Symbole :	***Nom ou fonction :***	***Associativité :***
( ) [ ] . ->	fonction, tableau, select.mbre struct	de gauche à droite
! ++ -- - ~ * & sizeof(type)	opérateurs unaires	de droite à gauche
* / %	multiplication, division, modulo	de gauche à droite
+ -	addition, soustraction	de gauche à droite
<< <= > >=	opérateurs relationnels	de gauche à droite
== !=	opérateurs de comparaison	de gauche à droite
&&	*ET* logique	de gauche à droite
||	*OU* logique	de gauche à droite
? :	opérateur conditionnel	de droite à gauche
= += -= *= /= %= <<= >>=	opérateurs d'affectation	de droite à gauche
,	opérateur virgule	de gauche à droite

```c
#include <stdio.h>

main()
{
	int a,b,c;
	printf ("entrez les valeurs de a, b, c");
	scanf ("%d %d %d", &a, &b, &c );
	c += (a>0 && a<=10) ? ++a : a/b;
	printf ("%d",c);
}
```

Evaluation de l'expression :

```c
(a > 0 &&  a <= 10)
/*pas besoin de parenthèses internes car les opérateurs relationnels*/
/*sont prioritaires sur l'opérateur logique */
```

Résultat du test :

si vrai on exécute ++a

si faux on exécute a/b

Somme et affectation :

on ajoute à c le résultat de l'action exécutée suite au test

Affichage du résultat :

ex1 : a=1, b=2, c=3 => c=5

ex2 : a=50, b=10, c=20 => c=25

7. Les fonctions

Fonction = groupe d'instructions qui exécute une tâche et retourne (souvent) une valeur à l'instruction appelante.

- Facilite l'écriture « modulaire » (modules logiques distincts).
- Crée des variables locales invisibles aux autres fonctions.
- Elimine les répétitions de code.
- On peut « appeler » une fonction ou l'exécuter à partir de n'importe quel point du programme.

7.1. Déclaration et définition des fonctions

La déclaration spécifie le nom de la fonction et le type de la valeur retournée, ainsi que le nombre et le type des arguments passés à la fonction.

7.1.1. Le prototypage

Prototype (ou déclaration) de fonction = nom de la fonction + type retourné + nombre et type des arguments (formels)

(selon la norme ANSI -> American National Standard Institute)

Exemple de prototype:

```
float sphere(int ray);
```

Le prototype permet au compilateur de vérifier le type des arguments.

7.1.2. La définition

La définition de la fonction comprend :

- L'en-tête (identique au prototype sans le (;)).
- Le corps (les instructions).

Une fonction peut en appeler une autre.

Une fonction ne peut être définie à l'intérieur d'une autre.

7.2. Structure d'un programme avec des fonctions

```
/*Volume.C:Calcule le volume d'une sphère*/

#include <stdio.h>            //directive de compilation
#define PI 3.14               //directive de compilation

float sphere (int ray);       //prototype (déclaration)
                              //le nom de l'argument formel « ray » peut être omis
void main ()
{
 float volume;
 int rayon = 3;
 volume = sphere(rayon);
 printf ("Volume : %f \n", volume);
}

float sphere (int ray)        //définition de la fonction sphere
{
       float resultat;
       resultat = ray * ray *ray;
       resultat = 4 * PI *resultat;
       resultat = resultat /3;
       return resultat;
 }
```

main :

- Fonction principale de tout programme C.
- Marque le début et la fin de l'exécution du programme.
- Le corps de ***main*** est encadré par deux accolades.
- Déclare les variables locales **volume** et **rayon**.
- Appelle les fonctions ***sphere*** et ***printf***.

sphere :

- Fonction qui calcule le volume de la sphère.
- Argument =ray (valeur que l'on passe à la fonction).
- Déclare la variable locale **resultat**.
- **return** : mot réservé qui retourne le résultat d'une fonction.

La fonction ***sphere*** est appelée dans ***main*** par l'instruction **volume = sphere (rayon)**; qui réalise deux opérations:

- Appelle la fonction ***sphere*** et lui transmet la valeur de la variable ***rayon*** (paramètre effectif).
- Affecte la valeur retournée par ***sphere*** à la variable ***volume***.
- ***printf*** :
  - fonction de bibliothèque fournie avec le compilateur.
  - Comporte une chaîne de format et une variable.

Le programme Volume.c comporte deux appels de fonction :

- à la fonction ***sphere***.
- à la fonction ***printf***.

Le prototype peut manquer si la définition de ***sphere*** précède le ***main*** (pas recommandé).

7.3. Arguments et paramètres

A l'appel d'une fonction on lui passe un « argument ».

Exemple :

```
volume = sphere (rayon) ;   // rayon est l'argument réel ou effectif de la fonction
```

L'instruction passe un argument appelé rayon à la fonction ***sphere***.

L'en-tête de la fonction ***sphere*** déclare un paramètre **ray** : ***float sphere (int ray)*** (ou **ray** est un paramètre formel) et lui affecte la valeur passée par l'appel de fonction.

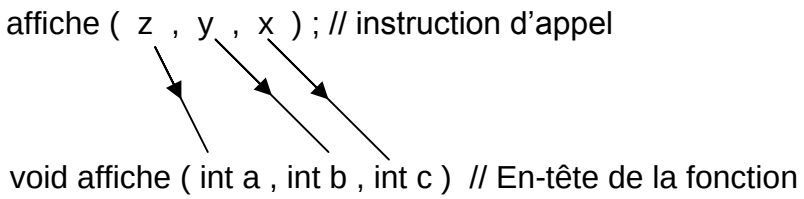
L'argument et le paramètre désignent la même valeur.

7.3.1. Passage par valeur

Correspondance ordonnée des arguments et des paramètres (si plusieurs) ;

```
/* Programme ORDPASS.C */

#include <stdio.h>

void affiche (int a, int b, int c) ;

void main ()
{
      int x = 10, y = 20, z = 30 ;
      affiche (z, y, x) ;                  //appel de la fonction affiche
}

void affiche (int a, int b, int c)
{
      printf ("a= %d b= %d c= %d \n", a, b, c) ;
}
```

main définit 3 variables de type **int** (x, y, z) et passe leur valeur aux trois arguments de l'affiche. Ces valeurs sont affectées dans le même ordre aux paramètres énumérés dans l'en-tête de la fonction affiche.



Le programme affiche a=30 b=20 c=10. Les arguments de la fonction sont passés **par valeur**. La fonction ***affiche*** crée 3 variables (a, b, c) et copie les valeurs reçues dans ces variables.

a, b, c = variables locales à afficher (ou temporaires) :

- n'existent qu'à l'activation de ***affiche***.
- sont détruites à la fin de l'exécution de ***affiche***.

x, y, z = variables locales à ***main*** (définies dans ***main***).

La fonction ***affiche*** peut changer les valeurs de ses variables locales (a, b, c), sans affecter les valeurs des variables originales (x, y, z) de main.

```
/*PARVAL.C (exemple de passage des paramètres par valeur)*/
#include <stdio.h>
#include <conio.h>

void affiche(int a, int b, int c);

void main()
{
	int x = 10, y = 20, z = 30;
	affiche(z, y, x);
	printf ("z= %d y=%d x=%d\n", z, y, x);		//(3°)
}

void affiche (int a, int b, int c)
{
	printf ("a=%d b=%d c=%d \n", a, b, c);		//(1°)
	a = 55;
	b = 66;
	c = 77;
	printf ("a=%d b=%d c=%d \n", a, b, c);		//(2°)
}
```

Affiche à l'écran :

1°) a=30 b=20 c=10

2°) a=55 b=66 c=77

3°) z=30 y=20 x=10

7.3.2. Passage par adresse

Le passage de paramètres par adresse est opposé au passage de paramètres par valeur; en effet on utilise l'adresse mémoire qui contient la valeur même de la variable et non pas une copie de cette valeur. Ce qui signifie que lorsqu'une modification de la valeur est effectuée dans la fonction, cela modifie également la valeur d'origine.

7.4. Commentaires sur les paramètres d'une fonction

Une déclaration de fonction définit :

- le nom de la fonction.
- Le type de valeur qu'elle retourne.
- Le nombre et le type des paramètres.

Déclaration = Prototype de fonction.

Chaque fonction du programme a un prototype, à l'exception de ***main***. Le prototype permet au compilateur de vérifier la concordance des types de données utilisées et est placé en début de programme (avant le ***main***).

7.5. Retour des valeurs de fonctions

L'instruction **return** exécute 2 opérations :

- provoque la fin de la fonction et rend le contrôle à l'instruction appelante ;

- retourne une valeur.

(voir " return resultat " dans la fonction " sphere ").

Une fonction peut avoir plus d'une instruction **return**.

```
if (erreur == 0)
      return 0 ;
else
      return 1 ;
```

Le premier **return** qui s'exécute termine la fonction.

Une fonction sans **return** se termine à l'accolade fermante.

```
/* la valeur de return de sphere est affectée à la variable « volume » */
volume = sphere (rayon) ;
printf ("Volume : %f \n", volume) ;
```

ou bien, si on n'a pas besoin de sauvegarder cette valeur :

```
printf (" Volume %f \n", sphere (rayon)) ; // (code plus concis).
```

Le type de valeur de retour est spécifié dans la déclaration et la définition de la fonction.

void = (le type vide) spécificateur du type (pas de valeur de retour).

7.6. Les fonctions récursives

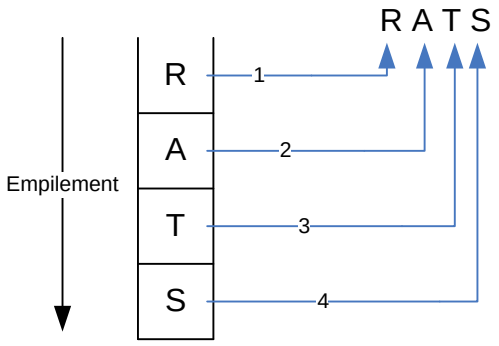
Fonctions qui s'appellent elles-mêmes. Le résultat de chaque appel est stocké dans la pile.

PILE = zone mémoire accessible seulement en entrée (fonctionnement de type LIFO).Voir cours d'algorithmique;

LIFO = Last In First Out (dernier entré, 1er sorti) (exemple : pile d'assiettes, de dossiers, etc.).

Les appels s'arrêtent avec une « condition d'arrêt ». Les résultats empilés sont dépilés en ordre inverse.

Affichage renversé d'un texte :




```c
#include <stdio.h>
#define EOL '\n'            // EOL = constante symbolique (End Of Line).

void renverse (void) ;      // prototype

main ()
{
       printf ("Saisissez le texte \n") ;      //ex « star »
       renverse () ;                           //appel
}

void renverse (void)
{
       char c ;
       if ((c =  getchar()) != EOL)      //arrêt avec EOL
              renverse () ;              //appel récursif
       putchar ( c) ;                    //affiche
       return ;
}
```

8. Attributs et qualificatifs des variables

8.1. Attributs

- Visibilité (portée) (classe de mémorisation)
- Durée de vie (en mémoire) (classe de mémorisation)

8.2. Qualificatifs

- Comportement durant l'exécution.

8.3. Classe de mémorisation

- automatique (auto)
- externe (extern)
- statique (static)
- registre (register)

8.4. Qualificatifs

- constante (la variable ne peut être modifiée)
- volatile (peut être modifiée par des évènements extérieurs au programme (exemple : interruptions))

8.4.1. Variables de classe automatique (variables locales)

- Déclarée à l'intérieur d'une fonction.
- Les arguments formels d'une fonction sont automatiques.
- Portée limitée à la fonction (portée locale).
- Durée de vie limitée à la fonction (leurs valeurs sont empilées/dépilées automatiquement lors des entrées/sorties de la fonction).
- Mot-clé : auto (pas obligatoire).

8.4.2. Variables de classe externe (variables globales)

- Déclarée en dehors de toute fonction.
- Portée jusqu'à la fin du programme (utilisée par toutes les fonctions).
- Durée de vie égale à celle du programme.
- Stockage dans la mémoire permanente.
- Mot-clé : extern (pas obligatoire si la variable est définie avant toute fonction).

- Initialisée à zéro par défaut.

8.4.3. Variables de classe statique

- Déclarée dans une fonction (comme les locales).
- Portée locale.
- Durée de vie : tout le programme (comme les globales).
- Stockées dans la mémoire permanente.
- Mot-clé : static.
- Initialisées à zéro par défaut.

« static » appliqué aux variables locales prolonge leur vie au delà de la fonction.

Exemple de variables locales déclarées « static » :

```
/* Programme STATIQUE.C*/
#include <stdio.h>

void ajoute_dix (int valeur) ;

main ()
{
	int val = 10 ;
	ajoute_dix (val++) ;
	ajoute_dix (val) ;
}

void ajoute_dix (int valeur)
{
	static int score ;
	if (valeur == 10)
		score = 0;
	score = score + valeur;
	printf ("score = %d \n", score) ;
}
```

Affiche :

score = 10

score = 21

1er appel -> passage de val = 10 à ajoute_dix (l'incrémentation se fera *après* le passage)

score = 0 + 10 = **10**

2ème appel -> passage de val = 11 à ajoute_dix

score = 10 + 11 = **21**

9. Les structures de contrôle

Elles contrôlent l'exécution du programme.

Il y a deux sortes d'instructions :

- Les instructions de branchement: **if…else**, **switch**;
- Les instructions d'itération (répétitives): **while**, **do**, **for**.

Les structures de contrôle se séparent en deux familles, les structures de choix et les structures de boucles. Elles permettent de modifier le comportement de l'application en fonction des valeurs de ses variables en modifiant l'ordre d'exécution des instructions.

9.1. Structures de choix

Les structures de choix offrent la possibilité au programmeur de contrôler le comportement de son application en évaluant des expressions.

9.1.1. La structure *if*

L'utilisation de **if** est des plus simples. Cela teste une condition et exécute une instruction ou un bloc d'instructions si cette condition est remplie. La condition peut être n'importe quelle expression. De manière facultative on peut exécuter d'autres instructions dans le cas ou la condition est fausse avec la structure **else**.

```
if (condition) {
/* la condition est remplie */
      instruction;
      autre instruction;
}
/* fin des instructions conditionnelles */
```

Exemple en pratique:

```
int a = 10;
if (a <= 20){
      printf("la valeur de a est inférieure à 20 \n");
      scanf("%d",&a);
}
```

Ceci va afficher la ligne 'la valeur de a est inférieure à 20' à l'écran car la condition est remplie.

Exemple d'utilisation de la structure **else**:

```
int a = 10;
if (a > 10)
      printf("la valeur de a est strictement supérieure à 10\n");
/* pas d'accolades car il n'y a qu'une instruction */
else
      printf("la valeur de a est inférieure ou égale à 10\n");
```

On peut aussi imbriquer les structures conditionnelles avec **elseif**:

```
int a = 10, b = 17;

if (a > 10){
	printf("la valeur de a est supérieure à 10\n");
	a = 10; /* une instruction quelconque */
}
elseif (b < 20){
printf("la valeur de a est inférieure à 10 et la valeur de b est inférieure à 20");
	a = 10;
}
else
	printf("a est inférieure ou égale à 10 et b est supérieure ou égale à 20\n");
```

if simple :

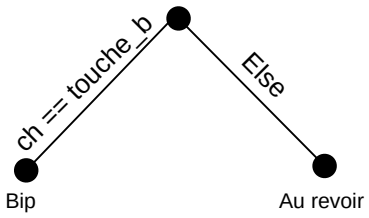
```
#include <stdio.h>
#include <conio.h>

main ()
{
	char ch;
	printf ("Appuyer sur la touche b\n");
	ch = getch ();
	if (ch == 'b')			//expression conditionnelle
	  printf ("Bip \a \n");		// instruction simple le '\a' émet un bip sonore
}
```

if emboîtés :

```
...
if (ch == 'b')
{
	printf ("bip \a \n");
	compteur_bips ++;
	if (compteur_bips > 10)
	{
		printf ("Plus de 10 bips...\n");
		if (compteur_bips > 100)
			printf ("Ne détruisez pas la touche 'b'! \n");
	}
}
```

Clause **else** : (utilisée avec **if**)

```
#include <stdio.h>
#include <conio.h>
#define TOUCHE_B 'b'

main ()
{
	char ch;
	printf ("Appuyer sur b pour entendre bip. \n");
	ch = getch ();
	if (ch == TOUCHE_B)
		printf ("Bip! \a \n");
	else
		printf ("Au revoir. \n");
}
```

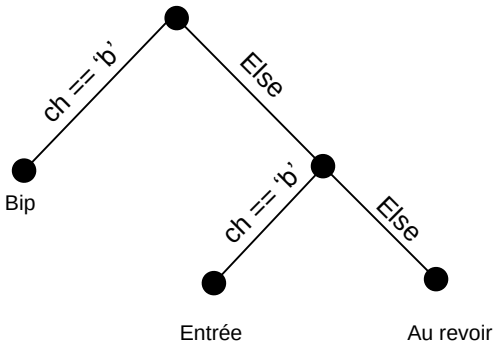


La structure **if … else** peut être remplacée par un opérateur conditionnel (mais ce n'est pas très lisible:):

```
int a;
printf("Entrez un nombre supérieur à 10 :");
scanf("%d", &a);
(a<10) ? ({
printf("a est inférieure à 10\n entrez à nouveau a : ");
scanf("%d", &a);
})
:({
printf("C bon, a est bien supérieure à 10\n");
});
```

Structure **if … else … if** :

```
/* Programme if - else - if */
#include <stdio.h>
#include <conio.h>

void main ()
{
	char ch;
	printf ("Appuyer sur la touche b, sur Entrée ou sur une autre touche. \n");
	ch = getch ();
	if (ch == 'b')
		printf ("Bip! \a \n");
	else	{
		if (ch == '\r')
			printf ("Entrée \n");
		else
			printf ("Au revoir. \n");
		}
}
```



9.1.2. La structure *switch*

La structure **switch** permet de fonder le choix des instructions à exécuter sur les valeurs possibles d'une expression et non plus sur la véracité de celle-ci. Chaque choix correspondant à une valeur est identifié par le mot clé **case**. Contrairement à **if**, le groupe d'instructions n'est plus délimité par des accolades mais par un autre mot clé : **break**. Le mot clé **default** correspond quant à lui au choix par défaut dans le cas ou aucun autre choix ne correspond à l'expression.

Le **switch** remplace avantageusement les **if … else** imbriqués. L'expression de sélection à évaluer donne comme résultat une valeur constante. Le corps de l'instruction est compris entre les deux accolades, et contient plusieurs choix (**case**).

```
...
int val;

if (val == 1)
       instructions 1;
else
       if (val == 2)
              instruction 2;
       else
              if (val == 3)
                     instruction 3;
```

Sera équivalent à:

```
switch (expression) {
    case val1:
        instruction 1;
        break ;
    case val2:
        instruction 2;
        break ;
    ...
    case valN:
        instruction N;
        break ;
    default:
        instruction D ;
        break ;
}
```

Un choix comprend :

- une étiquette.
- une instruction (tâche à exécuter).
- une instruction de sortie (**break**).

L'étiquette comprend :

- un mot réservé : **case**.
- une constante (comparée à l'expression de sélection).
- le signe **:**

default est un mot réservé (étiquette) qui contient l'instruction D et qui s'exécute si toutes les "val i" sont différentes de l'expression (i allant de 1 à N).

```
/*SWITCH.C: Principe de l'instruction switch */
```

```c
#include <stdio.h>
#include <conio.h>
#define TOUCHE_B 'b'
#define TOUCHE_ENTREE '\r'

void main()
{
       char ch;
       printf("Appuyer sur la touche b pour entendre un bip. \n");
       ch = getch ();

   switch(ch)
   {
       case TOUCHE_B:
              printf ("Bip! \a \n");
              break;
       case TOUCHE_ENTREE:
              printf("Entrée \n");
              break;
       default:
              printf("Au revoir. \n");
              break;
   }
}
```

Remarque :

- L'ordre des **case** est sans importance.
- Le nombre et la place des **break** sont importants.

break provoque des ruptures de séquences :

- elle sort d'une instruction **switch**.
- elle arrête une boucle.

*Utilisation de **break** dans **switch** :*

- termine un **case** et sort de **switch**.
- sans **break**, l'instruction **switch** passe au **case** suivant (exécution de plusieurs **case** de suite).

*Utilisation de **break** dans une boucle :*

- Provoque l'arrêt immédiat de la boucle et termine le programme.
- Cas des boucles emboîtées : l'instruction **break** ne met fin qu'à la boucle dans laquelle elle apparaît.

Attention **break** ne peut pas faire sortir d'une instruction **if…else**.

Exemple :

```c
/*BREAK.C: utilisation de l'instruction BREAK dans une boucle*/

#include <stdio.h>
#include <conio.h>

void main()
{
	char ch;
	printf("Appuyer sur une touche. TAB pour quitter. \n");
	while (1)	/*expression conditionnelle toujours vraie*/
	{
		ch = getch ();
		if(ch == '\t')
		{
			printf ("\a \n Vous avez appuyé sur TAB. \n");
			break;
		}
	}
}
```

Exemple :

```c
/*BREAK1.C: une instruction BREAK ne fait sortir que d'une seule boucle */

#include <stdio.h>
#include <conio.h>

void main()
{
	char ch;
	printf ("Appuyer sur une touche. ENTREE pour quitter. \n");

	do
	{
		while ((ch = getch ()) != '\r')
		{
			if (ch == '\t')
			{
				printf ("\a \n Vous avez appuyé, sur TAB. \n");
				break;
			}
		}
	} while (ch != '\r');

	printf("\n Au revoir.");
}
```

9.2. Les instructions d'itération

Création de boucles à exécution répétitive. Les structures de boucles nous donnent la possibilité de faire répéter une instruction (ou un bloc) plusieurs fois.

9.2.1. Structure *while*

La structure **while** permet de tester une condition (expression) et d'exécuter une instruction (ou un bloc) tant que cette condition est vraie. C'est l'équivalence au « tant que » algorithmique.

Syntaxe:

```c
while (condition)
```

```
	instruction
/* ou */
while (condition) {
	/* bloc d'instructions */
}
```

Exemple :

```
int compteur = 0;
while (compteur != 10) {
	printf("Compteur vaut %d\n", compteur);
	compteur ++;
}
```

Exemple :

```
/*WHILE.C: Principe de la boucle while */

#include <stdio.h>
#include <conio.h>

void main()
{
	int test = 10;
	while (test > 0)
	{
		printf("test = %d \n ", test);
		test -= 2;
	}
}
```

Affichage :

```
test = 10
test = 8
test = 6
test = 4
test = 2
```

9.2.2. Structure *do … while*

Une variante de **while** existe : **do ... while**. Elle permet d'entrer une première fois dans la boucle avant de tester la condition pour les autres fois.

Exemple:

```
int a = 0;
do
{
	printf("Voici la valeur de a : %d\n", a);
	a--;
} while (a > 0);
```

Dans ce cas, le programme affiche une fois la valeur de `a`.

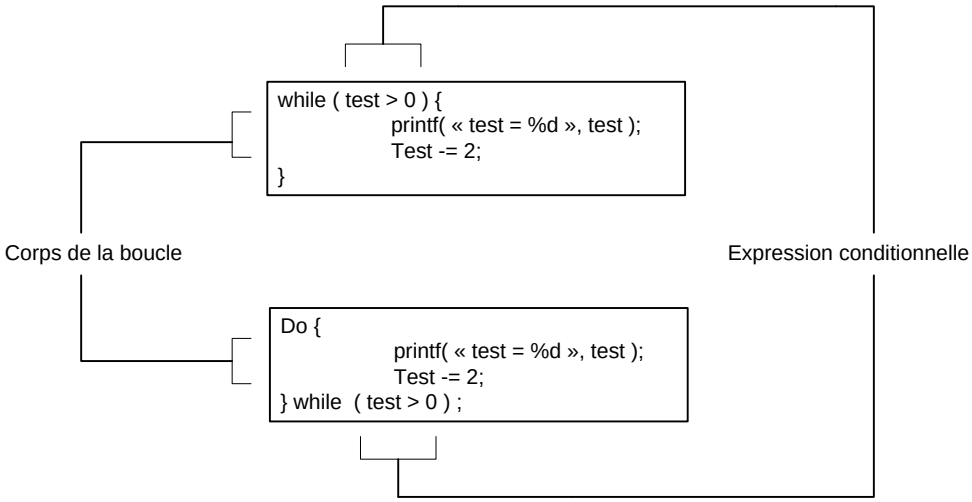
1. Evaluation de l'expression conditionnelle.
2. Si elle est vraie (non-nulle) => exécution et l'instruction et reprise de (1°)) (s'il y a plusieurs instructions on met des accolades).
3. Si fausse, pas d'exécution. On sort de **while**.

```
#include <stdio.h>

main ()
{
	int test = 10;
	do
	{
		printf ("test= %d \n", test);
		test -= 2;
	} while (test > 0);
}
```

Remarques :

- L'expression conditionnelle est évaluée *après* l'exécution des instructions.
- Les instructions sont évaluées au moins une fois.

*Comparaison entre les boucles **do** ... **while** et **while***



9.2.3. Structure *for*

L'instruction **for** permet aussi de faire des boucles avec un test de condition mais elle inclut en plus la possibilité de lui ajouter deux expressions supplémentaires qui le plus souvent vont servir à initialiser et à modifier les variables d'une autre expression. L'utilisation la plus courante est l'initialisation et la modification d'un compteur jusqu'à ce que celui ci atteigne la valeur souhaitée. C'est une méthode simple de faire exécuter un nombre précis de fois un bloc d'instructions mais on peut aussi bien utiliser une structure **while** pour faire ceci. En fait, l'instruction **for** n'est présente que pour rendre les programmes plus lisibles et pour sauver quelques lignes dans les fichiers car elle est assez

facilement remplaçable et dans certains cas, peu optimisée. D'une manière générale, il est plus efficace en termes de rapidité d'exécution d'utiliser une boucle **while.** Mais si vous cherchez à rendre votre code plus lisible, une structure **for** sera bienvenue.

Syntaxe :

```
for (expression(s) 1; expression(s) 2; expression(s) 3)
{
       instruction(s);
       instruction(s);
       instruction(s);
}
```

- Expression 1 => Initialisation d'une ou plusieurs variables (index).
- Expression 2 => Expression conditionnelle (condition d'arrêt).
- Expression 3 => Expression modifiant l'index.

*Fonctionnement équivalent à **while**:*

```
expression 1;
while (expression 2)
{
       instructions;
       expression 3;
}
```

Utilisation du ***for*** :

Remarque : On doit utiliser la structure **for** lorsque l'on connaît le nombre d'itérations, si on ne connaît pas ce nombre, alors vous devez utiliser la structure **while**

```
#include <stdio.h>
main ()
{
       int test;
       for (test = 10; test > 0; test -= 2)
              printf ("test= %d \n", test);
}
```

Même affichage que pour les boucles **while** et **do…while**.

Boucle For à expression s multiples

Exemple :

```
#include <stdio.h>
main ()
{
       int a, b;
       for (a = 256, b = 1; b < 512; a /= 2, b *= 2)
              printf ("a= %d b= %d \n", a, b);
}
```

(,) = opérateur d'évaluation séquentielle.

Affichage :

```
a = 256       b = 1
a = 128       b = 2
a = 64        b = 4
a = 32        b = 8
a = 16        b = 16
a = 8         b = 32
a = 4         b = 64
a = 2         b = 128
a = 1         b = 256
```

9.2.4. Instruction *continue*

Suspend l'itération en cours (saute les instructions restantes) et reprend l'itération suivante dans la même boucle (**break** sort de la boucle).

```
#include <stdio.h>
main ()
{
	int n, compteur, nbpositifs;
	float x, moyenne, somme = 0;
	printf ("Nombre de valeurs?");
	scanf ("%d", &n);
	for (compteur = 1; compteur <= n; ++compteur)
	{
		printf ("x = ");
		scanf ("%f", &x);
		if (x < 0)
			continue;	// si le test réussit saute les 2 instructions
					// restantes
		somme += x;		// et reprend à partir du FOR
		++nbpositifs;
	}
	moyenne = somme / nbpositifs;
	printf ("\n La moyenne est %f \n", moyenne);
}
```

Sans **continue** on aurait fait la moyenne des nombres positifs ET négatifs.

```
//Programme reproduisant l'algorithme du pgcd d'Euclide.

#include <stdio.h>
int pgcd (int u, int v)
{
	int t;
	while (u > 0)
	{
		if (u < v)
			{ t = u; u = v; v = t; }
		u = u-v;		// ou bien u=u%v;
	}
	return v;
}

void main ()
{
	int x, y;
	while (scanf ("%d  %d", &x, &y) != EOF)
		if (x>0 && y>0)
			printf ("%d %d %d \n", x, y, pgcd(x, y));
}
```

10. Les tableaux

10.1. Définition

Les variables que nous avons rencontrées jusqu'ici sont simples; elles ne permettent de stocker qu'une seule valeur à la fois. Mais, imaginez un programme dans lequel il faudrait stocker 100 entiers différents au même instant. Il serait très fastidieux de déclarer 100 variables de type **int** et de leur assigner une valeur à chacune. Pour cela, le C à définit les types complexes. Un tableau est un type complexe; il permet de stocker plusieurs variables d'un même type sous une seule variable de ce type. En fait les différentes variables sont indexées par rapport au début du tableau et on y accède en spécifiant le numéro d'index (indice) de chacune.

Pour résumer, un tableau est un ensemble de données du même type.

10.2. Tableau unidimensionnel

Type = le type des composantes du tableau (**int**, **char**, **double**…).

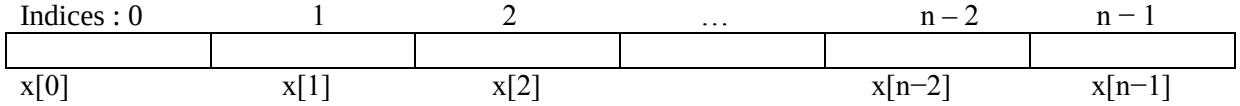
Nom = le nom du tableau.

Taille = le nombre *n* de composantes du tableau.

```
type nom [ taille ] ;
```

L'indice de la première composante est égal à 0.

L'indice de la dernière composante est égal à $n - 1$.



Le tableau x [*n*] à *n* éléments. Un tableau (*array*) = variable dimensionnée.

Nom = le nom de la variable (tab)

Type = le type de la variable (int)

Taille = le nombre de composantes de la variable (20)

10.2.1. Déclaration et initialisation d'un tableau d'entiers

```
int tab [3] = {-3, 84, 0} ;
tab[0] = -3;
tab[1] = 84;
tab[2] = 0;
```

10.2.2. Déclaration et initialisation d'un tableau de constantes

```
const type nom [ taille ] = { liste des valeurs } ;
```

Exemple :

```
const int minmax[2] = { -32768, 32767 }; // ou -32768 et 32767 sont des valeurs
                                         // constantes.
```

- *Si la taille déclarée dépasse le nombre d'éléments :*

```
float tab [ 7 ] = { 0.5, 0, -2.90, 0.85, 2.90 } ;
```

les éléments non-initialisés prennent automatiquement la valeur 0.

tab [0] = 0.5 tab [1] = 0 tab [2] = -2.90 tab [3] = 0.85

tab [4] = 2.30 tab [5] = 0 tab [6] = 0

- *Tableau de **caractères** :*

```
char couleur [ 4 ] = { 'B', 'L', 'E', 'U' } ;
```

on a

couleur [0] = 'B' couleur [1] = 'L' couleur [2] = 'E' couleur [3] = 'U'

- *Tableau de **chaînes** :*

```
char coul [ 4 ] = "BLEU";
char col [ ] = "BLEU";
```

Les tableaux sont différents :

coul [0] = 'B' coul [1] = 'L' coul [2] = 'E' coul [3] = 'U'

col[0] = 'B' col[1] ='L' col[2] ='E' col[3] = 'U' col [4] = '\0'

Dans le tableau "col" le compilateur ajoute automatiquement le caractère de fin de chaine '\0' (correct). Dans le tableau "coul" il n'y a pas de place (donc incorrect). Déclaration correcte : char coul [5] = "BLEU".

Programme de calcul d'écarts à la moyenne :

(nécessite un tableau pour mémoriser les nombres et les gérer dans une boucle).

```
#include <stdio.h>
main ()
{
       int n, x;
       float moy, ec, somme = 0;
       float tab [ 20 ];
       printf (" \n Nombre de valeurs ?");
       scanf ("%d ", &n);
       for (x = 0; x < n; ++ x)
       {
              printf (" i= %d t = ", x+1);
              scanf (" %f", &tab [ x ]);       //valeurs saisies
       somme += tab [ x ];
       }
       moy = somme / n;
       printf (" \n La moyenne est %5.2f \n ", moy);
       for (x = 0; x < n; ++ x)
       {
              ec = tab [ x ] - moy;
              printf (" i= %d t= %5.2f ec= %5.2f \n", x+1, tab [ x ], ec);
       }
}
```

Résultats affichés : (les valeurs soulignées sont saisies)

```
Nombre de valeurs ? 5            La moyenne est : 4.18
i= 1   t= 3                 i= 1   t= 3.00      ec= -1.18
i= 2   t= -2                i= 2   t= -2.00     ec= -6.18
i= 3   t= 12                i= 3   t= 12.00     ec= 7.82
i= 4   t= 4.4               i= 4   t= 4.40      ec= 0.22
i= 5   t= 3.5               i= 5   t= 3.50      ec= -0.68
```

Même programme avec des valeurs entrées.

```
#include <stdio.h>

int n = 5;                          //Valeurs entrées
float tab [ ] = {3, -2, 12, 4.4, 3.5}; //valeurs entrées

main ()
{
       int n, x;
       float moy, ec, somme = 0;
       for (x = 0; x < n; ++x)
              somme += tab [ x ];
       moy = somme / n;
       printf (" \n La moyenne est %5.2f \n ", moy);

       for (x = 0; x < n; ++ x)
       {
              ec = tab [ x ] - moy;
              printf (" i= %d t= %5.2f ec= %5.2f \n", x+1, tab [ x ], ec);
       }
}
```

10.3. Tableau multidimensionnel

Un tableau multidimensionnel est un tableau composé d'autres tableaux. On peut se représenter facilement un tableau à deux dimensions; c'est un tableau qui regroupe des tableaux. Mais un tableau tri- voir quadridimensionnel est beaucoup plus difficile à concevoir dans notre esprit. Ce serait un tableau de tableaux eux mêmes composés de tableaux à leur tour composés de tableaux… Mais un tableau multidimensionnel en mémoire est représenté comme un vecteur : de façon linéaire.

Voici comment on déclare un tableau bidimensionnel de 100 éléments:

```
int tableau[2][50];
```

Ceci équivaut à un tableau unidimensionnel de 100 éléments, seul l'indexage change. Le tableau possède 2 lignes et 50 colonnes.

Les éléments du tableau sont eux-mêmes des tableaux.

Déclaration :

```
TYPE NOM [ TAILLE1 ] [ TAILLE2 ] …. [ TAILLEN ];
```

TAILLEi = nombre de composantes associées à la dimension i.

Exemple :

Tableau à 2 dimensions (matrice).

Déclaration :

```
int tab[3][4]
```

3 lignes et 4 colonnes.

Initialisation dans le programme :

```
int mat [ 3 ][ 4 ] = { {5, 3, -2, 42}, {44, 15, 0, 6}, {97, 6, 81, -21} }
```

L'élément de valeur –21 est désigné par : mat [2][3] (3ème ligne, 4ème colonne)

Col0	Col1	Col2	Col3	
5	3	-2	42	Ligne0
44	15	0	6	Ligne1
97	6	81	-21	Ligne2

On peut initialiser un tableau multidimensionnel de cette façon:

```
int tableau[20][40],i,j;
for (i=0; i < 20; i++){
      for (j=0; j<40; j++) tableau[i][j] = i*40+j;
}
```

Mais on peut aussi procéder ainsi :

```
int tableau[20][40],i=20*40;
while(i.!=0)
      tableau[i/40][i%40] = i;
```

10.4. Manipulation des tableaux et des chaînes

Une chaîne peut être représentée avec un tableau ou avec un pointeur.

10.4.1. Méthodes pour entrer une chaîne dans un tableau

a) Déclaration et initialisation du tableau avec la chaîne :

Exemple :

```
char texte [ ] = "ceci est un exemple ";
```

b) Utilisation de la fonction **strcpy** (de STRING COPY) :

Exemple :

```
char tab [ 30 ];    //déclaration d'un tableau de 30 caractères
strcpy ( tab, "ceci est un exemple");
```

c) Utilisation de la fonction **gets** (de GET STRING):

Exemple :

```
char tab [ 30 ];
printf ("Entrez la chaîne ");
gets (tab);                          //chaîne saisie à l'écran
```

d) **scanf** peut lire des chaînes mais s'arrête au premier blanc :

Exemple :

```
scanf ("%s ", tab);          // ne lit que la chaîne "ceci"
```

e) **scanf** peut lire la chaîne caractère par caractère

```
#include <stdio.h>

main ()
{
       int i;
       char tab [ 30 ];
       printf ("Entrez la chaîne "); // on entre "ceci est un exemple"
       for (i = 0; i < 10; ++ i)      // lecture de la chaîne caractère par caractère
         scanf ("%c", &tab[ i ]);    // &tab[ i ] désigne l'adresse du caractère 'i'
}
```

10.4.2. Affichage des chaînes avec *puts* (de PUT STRING)

Exemples :

```
#include <stdio.h>
main ()
{
       char tab [ ] = "ceci est un exemple";
       puts ( tab );       //affiche :  ceci est un exemple
}
```

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

main ()
{
       char tab [ 30 ];
       strcpy ( tab, "ceci est un exemple");
       puts ( tab );       //affiche la même chose.
}
```

10.4.3. Copie de chaînes

```
#include <stdio.h>

void main ()
{
       char src [ ] = "bonjour !";
       char dest [ ] = "salut les copains";
       int i = 0;
       while ( src [ i ] != '\0')
       {
              dest [ i ] = src [ i ];
              i ++;
       }
       puts ( dest );            //affichage ?
}
```

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

```
void main()
{
	char src [ ] = "bonjour !";
	char dest [ ] = "salut les copains";
	int i = 0;
	while ( (dest [ i ] = src [ i ]) != '\0')
		i ++;
	puts ( dest );		//affichage ?
```

Exemple 1 :

```
/*PROGRAMME TABLEAU.C (principe des tableaux uni et bi-dimensionnel)*/

#include <sdtio.h>

void main ()
{
   int j, k;
   int tableau_n [ 2 ][ 3 ] = { {12, 2, 444}, {6, 55, 777} };
   char tableau_c [ ] = "Salut";

   printf (" --- Valeurs ---     --- Adresses---\n \n");

   for (j = 0; j < 2; j = j + 1)
   {
      for (k = 0; k < 3; k = k + 1)
      {
       printf ("tableau_n [ %d ][ %d ] = %d ", j, k, tableau_n [ j ][ k ]);
       printf ("\t &tableau_n  [ %d ][ %d ] = %u \n", j, k, &tableau_n [ j ][ k ]);
      }
      printf ("\n");
   }

   for (j = 0; j < 6; j = j + 1)
   {
       printf ("tableau_c [ %d ] = %x %c", j, tableau_c [j ], tableau_c [ j ] );
       printf ("\t &tableau_c [ %d ] = %u \n", j, tableau_c [ j ] );
   }
}
```

Affiche :

```
---Valeurs---              ---Adresses---

tableau_n [ 0 ][ 0 ] = 12              &tableau_n [ 0 ][ 0 ] = 3792
tableau_n [ 0 ][ 1 ] = 2               &tableau_n [ 0 ][ 1 ] = 3794
tableau_n [ 0 ][ 2 ] = 444             &tableau_n [ 0 ][ 2 ] = 3796

tableau_n [ 1 ][ 0 ] = 6               &tableau_n [ 1 ][ 0 ] = 3798
tableau_n [ 1 ][ 1 ] = 55              &tableau_n [ 1 ][ 1 ] = 3800
tableau_n [ 1 ][ 2 ] = 777             &tableau_n [ 1 ][  2 ] = 3802

tableau_c [ 0 ] = 53 S                 &tableau_c [ 0 ] = 3803
tableau_c [ 1 ] = 61 a                 &tableau_c [ 1 ] = 3804
tableau_c [ 2 ] = 6c l                 &tableau_c [ 2 ] = 3805
tableau_c [ 3 ] = 75 u                 &tableau_c [ 3 ] = 3806
tableau_c [ 4 ] = 74 t                 &tableau_c [ 4 ] = 3807
```

L'opérateur "calcul d'adresse" (&) fournit l'adresse mémoire de l'expression qu'il précède.

10.4.4. Passage de tableaux à une fonction

Les tableaux peuvent être utilisés comme arguments de fonctions.

```
float calcule (int, float [ ]);  //prototype sans noms d'arguments formels;
                                 //attention aux crochets vides
```

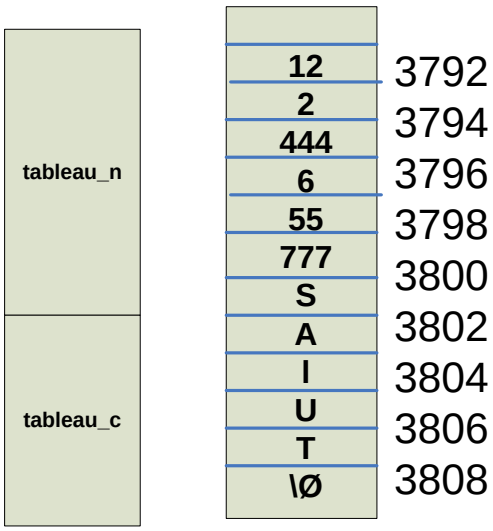
```
// float calcule (int a, float x [ ]); //prototype avec noms de variables

main ()
{
	int n; float res; float tab [100];
	...
	res = calcule (n, tab);	//appel de la fonction "calcule";
				//tab est transmis par adresse
	...
}

float calcule (int a, float x [ ])		//définition de « calcule »
{
	...
}
```

Le passage des arguments se fait par *adresse*. L'adresse de "tab" (adresse du 1er élément) est transmise à "x" (pointeur sur le 1er élément). On peut accéder à tout élément du tableau en variant les indices. Les modifications se font sur l'original donc elles sont valables au-delà de la fonction.

10.5. Comportement en mémoire des tableaux

```
char tableau_c [ 6 ] = { 'S', 'a', 'l', 'u', 't', '\0'};
```

'\0' est le caractère final ajouté par le compilateur.



10.5.1. Mise en mémoire des tableaux à plusieurs dimensions

Le compilateur C ne reconnaît que des tableaux à une dimension. Ses éléments peuvent être aussi des tableaux.

Donc : tableau de tableau de …… (à *n* dimensions).

Exemple :

Soit un tableau à 2 dimensions (matrice) à N lignes et M colonnes:

float tab [N] [M]

tab peut être vu comme un tableau à N éléments, dont chaque élément est un tableau à M éléments.

Chaque élément tab [i][j] de tab [i] est un **float** . Attention à la double indexation (tab [i, j] est faux).

Exemple de mise en mémoire pour N = 2 et M = 3

tab [0][0], tab [0][1], tab [0][2], tab [1][0], tab [1][1], tab [1][2]

10.5.2. Calcul de l'adresse d'un élément du tableau

```
adresse_de_tab [ i ][ j ] = adresse_de_tab [ 0 ][ 0 ] + i * taille_de_ligne_de_tab
+ j * taille_de_élément_de_tab

taille_de_ligne_de_tab = M * taille_de_élément_de_tab
```

Note : le nombre total de lignes N ne compte pas dans le calcul.

Exemple :

```
int tab[3][4] = {{1,2,3,4},
                 {5,6,7,8},
                 {9,10,11,12}};

tab [ 1 ][ 2 ] = 7
```

&tab [1][2] = &tab [0][0] + 1 * 8 octets + 2 * 2 octets = &tab [0][0] + 12 octets

taille_ligne = 4 * 2 octets = 8 octets

Mémorisation linéaire du tableau :
{1,2,3,4,5,6,7,8,9,10,11,12}

&tab[0][0]

Calcul de l'indice linéaire d'un élément de la matrice :
tab [i][j] => M * i+ j

Exemple :
7 = tab [1][2] =4 * 1+ 2 (7 est le 6ème élément)

11. Les structures

Un tableau est pratique mais on ne peut y stocker qu'un type de données. Il est souvent utile de pouvoir regrouper des éléments de types différents. Pour cela nous avons les structures. Les structures vont permettre, tout comme un tableau, de rassembler des variables dans une même boite définie par un seul nom. Les variables contenues dans cette boite sont appelées *champs*. Avant de définir des variables de type structure, il faut déclarer la structure elle-même; en fait on crée le modèle qui servira à créer les variables de ce type.

11.1. Création d'un type structure

On procède comme ceci :

```
struct NomdelaStructure {
	type champ1;
	type champ2;
	type champn;
	...
}; /* ne pas oublier le ; */
```

Ce qui donne en pratique:

```
struct Client {
	char *Nom; /* Vous verrons bientôt ce que représente le type 'char *' */
	char *Prenom;
	int age;
	char *adresse;
};
```

Et ensuite on peut définir un variable de type Client dont on pourra modifier les champs :

```
struct Client Monclient;
/* Client est le nom de la structure et Monclient est
le nom de la variable de type Client qui sera
utilisée. */
```

Puis on accède aux champs de la structure en utilisant cette syntaxe:

```
Monclient.age = 42; /* on affecte 42 au champ âge de la variable Monclient de
type client */
```

Exemple :

La fiche de paie d'un employé comprend :

- son nom.
- nombres d'heures travaillées.
- le taux horaires.

```
struct nom_type
{
	déclaration champ 1
	…
	déclaration champ N
};
```

```
struct employe
```

```
{
	char nom [ 10 ];
	int heures;
	float salaire;
};
```

- **struct** = mot réservé;
- employe = nouveau type de données (étiquette de **struct**);
- nom, heures, salaire = membres ou champs de la structure employe.

Création des variables de type "structure"

```
struct < nom_type> <nom_var>;
```

```
struct employe dupont; //(instruction qui crée la variable dupont de type employe)
```

Ou bien dans la même instruction :

- déclaration du type (employe)
- création de la variable dupont.

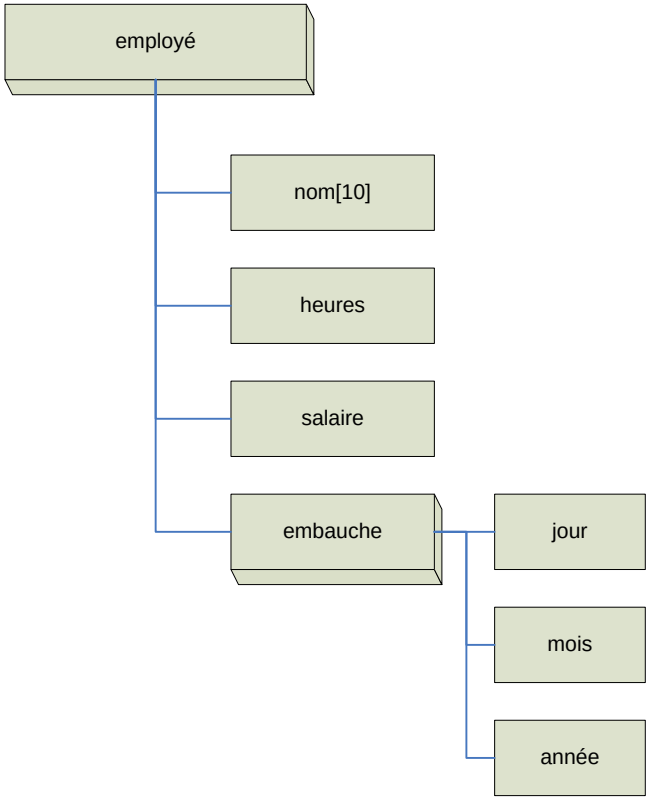
```
struct nom_type
{
	déclaration des champs
} nom_variaibles;
```

```
struct employe
{
	char nom [ 10 ];
	int heure;
	float salaire;
} dupont;
```

Ou bien :

```
struct                    //l'identificateur "employé" peut manquer
{
	char nom;
	int heures;
	float salaire;
} dupont, durand, mathieu;       //plusieures variables
```

Une structure membre d'une autre structure :




```
struct date         //déclarée avant employe
{
       int jour;
       int mois;
       int annee;
};

struct employe
{
       char nom [ 10 ];
       int heures;
       float salaire;
       struct date embauche;
} empl1, empl2;

//embauche est une variable de "date"
//empl1 et empl2 sont des variables de type employe
```

11.1.1. Appel des champs par les variables de structure

Syntaxe de l'appel : <nom_variable>**.** <nom_champ>

Exemple :

struct employe :

dupont.nom

dupont.heures

dupont.salaire

‘.’ = opérateur sélecteur membre de structure.

Initialisation

Initialisation de la variable de type employe :

```
strcpy (dupont.nom, "Dupont,D");
dupont.heures = 180;
dupont.salaire = 9320.50;
```

Déclaration et initialisation simultanée

```
struct employe dupont = { "Dupont,D", 180, 9320.50};
```

Cas des structures imbriquées

```
struct employe empl1 =
{
      "Jean,Gros", 180, 6800.80, 15, 11, 1990
};           //15, 11, 1990 correspond à  la structure embauche.
```

```
empl1.embauche.mois;
/* la variable empl1 (de employe) appelle la variable embauche (de date) et celle-
ci appelle le champs mois */
```

Changement de valeur d’un membre

```
dupont.heures = 147;
```

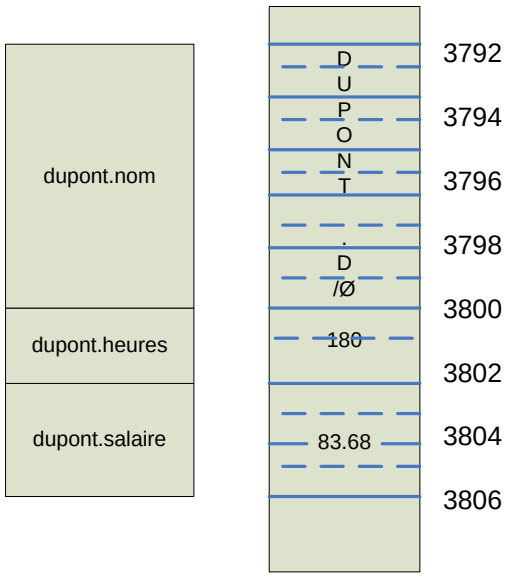
Affectation

Affectation d’une structure à une autre :

```
struct employe durand = dupont;
```

(les valeurs des champs de "dupont" sont affectées à la variable "durand")

11.1.2. Rangement en mémoire de la variable de structure



11.2. La directive *typedef*

Permet d'associer un type de donnée à un nom.

```
typedef int entier ;
```

(« entier » devient synonyme de **int**)

Syntaxe :

```
typedef struct
{
      déclaration des composantes
} nom_type ;
```

Ou nom_type est le nom donné au type structuré.

Syntaxe : **typedef** type identificateur

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

"personne" est un type structuré avec 3 composants.

Avantage de « typedef » :

- améliore la clarté du programme ainsi que sa portabilité.
- permet de déclarer des variables de structure

Ex : personne jean

(Jean est une variable de type **struct** avec 3 composantes)

Initialisation : jean.age=20 ;

11.3. Tableau de structures

Les N employés d'une entreprise ont tous la même structure :

```
struct employe
{
       char nom[10] ;
       int heures ;
       float salaire ;
} ;
```

Déclaration du tableau dont les éléments ont la structure employe :

```
struct employe tab [N] ;
```

Avec **#define N 50** au début du programme.

Ou bien déclaration de la structure et du tableau en une seule instruction :

```
struct employe
{
       char nom[10] ;
       int heures ;
       float salaire ;
} tab[N] ;
```

Il faut initialiser les éléments du tableau (données entrées avec le programme ou saisies avec des questions).

```
tab[26].heures = 100; // initialise les «heures » de l'employé 27.
```

Boucle **do**, **for** ou autre pour saisir ou afficher.

```
scanf ("%s ", &tab[0]. nom)            // saisit le nom du 1er employé.
printf ("%f ", tab[1]. salaire)        // affiche le salaire du 2ème employé.
```

Exemple de données entrées avec le programme :

```
struct employe
{
       char nom[10] ;
       int heures ;
       float salaires ;
} tab[ ]=
     {
       {"Dupont. D", 80, 90.50} ,
       {"Bertrand. J", 75, 82.60},
       ...
       {"Poirot. V", 32, 45.20}
     } ;
```

Exemple de tableau de structures imbriquées :

```
struct date {int jour ; int mois ; int année ;} ;
```

```
struct employe {
       char nom[10] ;
       struct date embauche ;  /*déclare la variable « embauche » de la structure
                               imbriquée « date » */
} empl[25] ;
```

Appel du nom de l'employé n°15 :

empl[14].nom

Appel du 8ème caractère du nom :

empl[14].nom[7]

Appel du mois d'embauche de l'employé n°15 :

empl[14].embauche.mois

Incrémentation du mois d'embauche :

++ empl[14].embauche.mois //l'opérateur '**.**' est prioritaire.

11.4. Les structures comme arguments des fonctions

Le passage des informations s'effectue :

1°) avec certains membres de structure.

2°) avec la structure tout entière

Les passages se font par valeur ou par adresse.

1°) Passage de membre(s) de structure à une fonction :

```
struct employe
{
       char nom[10] ;
       int heures ;
       float salaire ;
} ;
//on veut afficher seulement les noms et les heures travaillées :

void affiche (char nom[ ], int heures) ;      //prototype
// void affiche (char [ ],int) ;              // variante

main ()
{
       struct employe empl1 ;
       strcpy (empl1.nom, "pascal") ;
       empl1.heures = 200 ;
       affiche (empl1.nom, empl1.heures) ;    //appel de la fonction affiche
}

void affiche (char nom[ ], int heures)
{
       printf ("Nom : %s \n", nom) ;
       printf ("Durée de service : %d \n", heures) ;
}
```

2°) Passage de la structure (entière) :

Par valeur :

```c
/* Programme Employe.c*/
#include <stdio.h>
#include <string.h>

struct employe
{
	char nom [10];
	int heures;
	float salaire;
};

void affiche (struct employe personne);		/* prototype */

main ()
{
	struct employe dupont;			/* crée la var "dupont" */
	strcpy (dupont.nom, "Dupont. D") ;
	dupont.heures = 180 ;
	dupont.salaire = 83.68 ;
	affiche (dupont) ;			/* appel de affiche */
}

void affiche (struct employe personne)
//une copie de "dupont" est passée à "personne"
{
	printf ("Nom : %s \n", personne.nom) ;
	printf ("Durée de service : %d \n", personne.heures) ;
	printf ("Salaire horaire : %6.2f \n", personne.salaire) ;
}
```

Affichage :

Nom : Dupont.D

Durée de service : 180

Salaire horaire : 83.68

Visibilité d'une variable de structure.

-locale – si définie à l'intérieur de la fonction.

-globale – si définie à l'extérieur de la fonction.

Exemple : On ajoute à la fin de la fonction « affiche » (programme employe.c)

strcpy (personne.nom, "Leblanc.B") ;

printf ("%s \n", personne.nom) ;

Affichage :

Leblanc.B => structure locale à la69fonction « affiche »

=> pas d'effet sur celle de « main » car le changement s'est effectué sur la « copie »

Preuve :

Si l'on affiche à la fin de « main » le nom de l'employé dupont

```
printf ("%s \n", dupont.nom) ;
```

Affichage :

Dupont.D //on voit que le nom original n'a pas changé

11.5. Pointeurs de structure

On peut affecter à un pointeur l'adresse d'une variable de structure en créant ainsi un « pointeur de structure ». Avec ce pointeur on peut accéder à n'importe quel membre de cette structure.

Rappel de la notation utilisée avec une variable de structure

dupont.nom dupont : *variable de structure*

(.) *opérateur sélecteur de membre de structure.*

nom : *nom du membre de structure*

Notation utilisée avec un pointeur de structure :

Remplacer le nom de la variable de structure par le nom du pointeur

Remplacer l'opérateur (.) par l'opérateur (->)

Les instructions à écrire sont :

```
struct employe *ptr_e ;        // création du pointeur de structure

struct employe dupont ;        //création de la variable de structure

ptr_e = &dupont ;              // orientation du pointeur vers la variable

ptr_e -> nom ;             // appel du membre de nom « nom » ou bien (variante):
(*ptr_e).nom ;             //(les parenthèses sont nécessaires car '.' est
                           // prioritaire sur ' * '.
```

Passage d'une structure par adresse (programme Employe1.c)

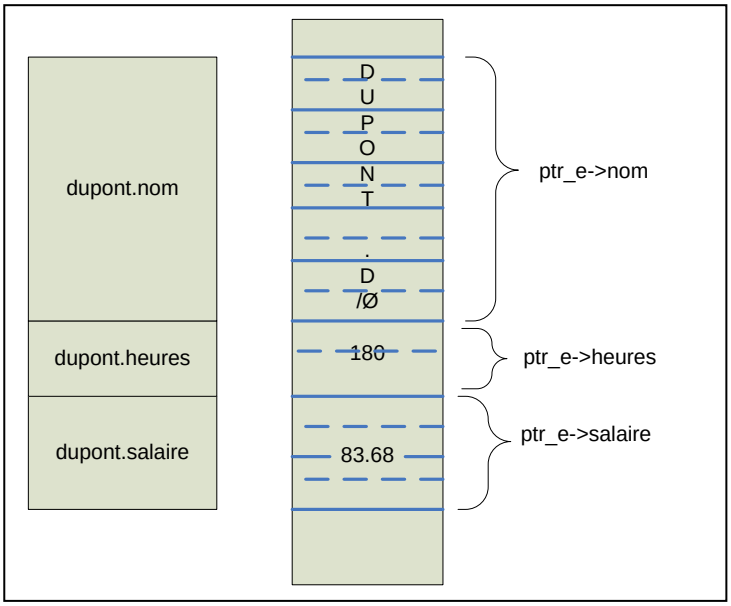
```
#include <stdio.h>

struct employe              //déclaration de la structure « employe »
{
      char nom[10] ;
      int mois ;
      float salaire ;
} ;


void affiche(struct employe *ptr_e) ;
      /* prototype de affiche, ptr_e est un pointeur de structure*/

int main(void)
```

```
{
	// déclaration et initialisation simultanées de la variable "dupont"
	struct employe dupont = {
		"Dupont. D", 180, 83.68
	} ;

	affiche(&dupont) ;  // appel de la fonction "affiche"
}

void affiche (struct employe *ptr_e)    // définition de la fonction « affiche »
{
	printf("Nom : %s \n", ptr_e->nom) ;
	ptrintf("Durée de service : %d\n", ptr_e->heures) ;
	printf("Salaire horaire : %6.2f\n", ptr_e->salaire) ;
}
```

Rangement en mémoire des pointeurs de membre de structure dans « Employe1.c »

Le pointeur « ptr_e » créé par la fonction « affiche » reçoit l'adresse passée dans l'appel d'affiche : affiche (&dupont) ;

Le pointeur ptr_e peut se référer à n'importe quel membre de la structure (nom, heure, salaire)



L'instruction :

printf(″Nom : %s \n″, ptr_e → nom) ; /ou :

printf(″Nom : %s \n″, (*ptr_e).nom) ; donne le même résultat que :

printf(″Nom : %s \n″, dupont.nom) ; mais accède à l'original et non pas à la copie.

11.6. Illustration de l'accès à la structure originale

Modifions le nom de l'employé avec l'instruction :

```
strcpy ( ptr_e->nom, "Durand, G" ) ;
```

ajoutée à la fin de la fonction « affiche ». Puis ajoutons à la fin de « main », l'instruction :

```
printf(" %s \n", dupont.nom) ;
```

Le programme Employe1 affiche : Durand, G, preuve que grâce au PMS, la fonction « affiche » a pu modifier la structure.

Même opération dans le programme Employe :

On ajoute dans « affiche » :

```
printf( personne.nom, "Durand, G" ) ;
printf(" %s\n", personne.nom ) ;
```

Cela affiche : Durand, G car une nouvelle chaîne est copiée dans le membre personne.nom.

Ce changement n'a pas d'effet sur la structure d'origine. La preuve :

L'instruction :

```
printf("%s \n", dupont.nom) ;
```

Ajoutée dans « main », affiche toujours : Dupont.D

Comparaison des programmes :

Employe.c utilise une variable de structure

Employe1.c utilise un pointeur de structure

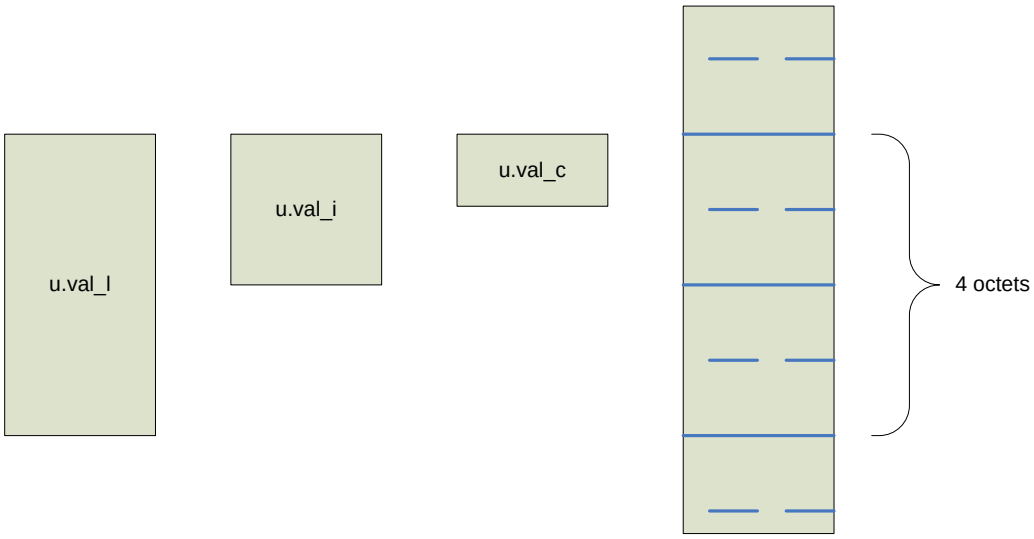
Employe.c :	Employe1.c :
Passe la structure entière à la fonction affiche : Void affiche(struct employe personne) ; (personne est une variable de type employe)	Passe à affiche un pointeur: void affiche(struct employe *ptr_e) ; (*ptr_e est un pointeur de type employe)
Appelle ″affiche″ avec la variable de structure : Affiche(dupont) ;	Appelle ″affiche″ avec l'adresse de la variable: affiche(&dupont) ;
″affiche″ utilise une copie locale de la structure	″affiche″ utilise l'original à l'aide de l'adresse →économie de mémoire →l'accès à l'original permet de modifier une structure définie ailleurs dans le programme (visibilité accrue)

11.7. Les Unions

Déclaration : (pareille à la structure) :

```
union exemple       // exemple = étiquette de l'union
{
       char val_c ;
       int val_i ;  // champs ou membres
       long val_l ;
} u ;               // u = variable de type union
```

La variable u ne peut contenir (à un moment donné) qu'un seul des trois types (char, int, long). Une variable de structure contient les trois types simultanément. Le compilateur alloue la mémoire nécessaire à la plus grande des données (ici long : 4 octets).



Accès aux membres de l'union :

nom_union.membre // . ou → = sélecteurs de membre de structure

pointeurs_union → membre

Un seul des trois membres est actif à la fois

Avantage : économie de mémoire

```c
union monunion
{
       int i ;
       double d ;
       char ch ;
} mu, *mu_ptr = &mu ;
             // Le pointeur *mu_ptr pointe vers la variable mu de type union

/*La variable mu de type union monunion peut contenir :
1 int sur 2 octets  //
1 double sur 8 octets      //     un seul des trois champs à la fois
1 char sur 1 octet  //

L'appel: sizeof(union monunion)  // renvoie 8 octets
       ou     sizeof(mu)    // (espace réservé pour la plus grande des données ) */

int main(void)
{
 mu.d = 4.016;                                  // initialisation du membre 'd'
 // ou (*muptr).d = 4.016;                      // autre forme d'appel de 'd'
 //ou muptr->d = 4.016;                         // appel par le selecteur '->'

 printf( "mu.d = %f\n", mu.d);          // affichage correct : mu.d = 4.016
 printf( "mu.i = %i\n", mu.i);          // résultat imprévu car la mémoire
                                        // est occupée par le double 'd'
 muptr->i = 3;                          // initialisation du membre 'i'
 printf( "mu.i = %d\n", mu.i);          // affichage de la valeur : mu.i = 3
 mu.ch = 'A';                           // initialisation du caract. 'ch'
 printf( "mu.ch = %c\n", mu.ch);        // affichage: A
 printf( "mu.i = %d\n", mu.i) ;         // résultat imprévu
 printf( "mu.d = %f\n", muptr->d);      // résultat imprévu
}
```

Conclusion : une UNION ne peut contenir à la fois qu'un seul de ses membres

12. Les pointeurs

On dit souvent qu'on perd toute la puissance et la flexibilité du C si on n'utilise pas ou si on ne maîtrise pas les pointeurs. Le cas des pointeurs est souvent considéré comme délicat lors de l'apprentissage du C. En fait, ce n'est pas si compliqué que ça.

Un pointeur est une variable qui contient l'adresse en mémoire d'une autre variable. On peut avoir un pointeur sur n'importe quel type de variable. Pour récupérer l'adresse d'une variable, on la précède de l'opérateur `&`. Pour obtenir la valeur contenue par une zone mémoire pointée par un pointeur, on préfixe celui-ci par *. Lorsqu'on déclare un pointeur, on doit obligatoirement préciser le type de la donnée pointée pour permettre au pointeur de connaître le nombre de blocs mémoire que prend la variable qu'il pointe. Un bloc représente un octet.

Quel est l'intérêt des pointeurs ? Principalement leur grande efficacité et flexibilité par rapport aux variables standard. Grâce aux pointeurs, on va pouvoir économiser des déplacements et copies de mémoire inutiles en précisant uniquement où se trouve la donnée.

12.1. Déclaration d'un pointeur

Pareille à celle d'une variable :

```
type *ptr;
```

```
int *ptr;
```

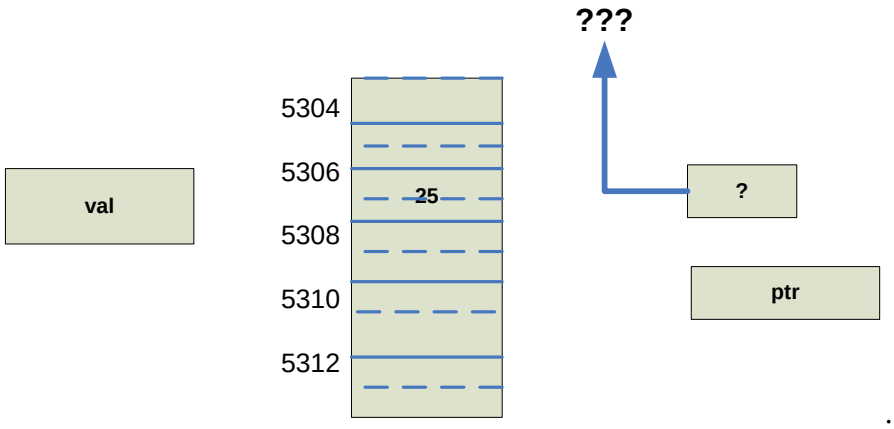
(ou **int** désigne le type (pointé) et ptr le nom du pointeur.

(*) = opérateur d'indirection (indique que la variable "ptr" est un pointeur).

int = le type d'objet vers lequel il pointe.

```
char *ptr, *car; // déclaration de deux pointeurs de type char.
int val = 25; // la variable "val" contient une valeur de type int.
int *ptr; // la valeur "ptr" pointe vers une donnée de type int.
```

La relation entre "val" et "ptr" dans le programme POINTEURS.C après la déclaration de ptr (non initialisé).



La variable "val" est rangée à l'adresse mémoire 5308. A cette adresse on trouve le contenu de "val" (c'est à dire l'entier 25).

La variable "ptr" est rangée à une adresse non spécifiée (le compilateur lui alloue la mémoire nécessaire). Le contenu de "ptr" n'a pas encore été défini (?) → variable non initialisée (à éviter).

12.2. Initialisation d'un pointeur :

Initialiser un pointeur c'est lui donner une adresse mémoire:

```
ptr = &val;  /*affecte l'adresse de "val" à "ptr" */
```

& = opérateur de calcul d'adresse.

```
&val = 5308; // adresse de val.
```

Le contenu de "ptr" est alors défini : ptr = 5308.

Variantes du programme POINTEUR.C :

Initialisation de "val" par *ptr	Déclaration et initialisation simultanée du pointeur
…….. main () { int val, ptr; ptr = &val; *ptr = 25; printf .... }	…….. main () { Int val = 25; int *ptr = &val; printf ……. …… }

Remarquez la différence de syntaxe entre :

```
ptr = &val ;    // initialisation après déclaration
```

et

```
int *ptr = &val ;   // déclaration et initialisation simultanée
```

12.3. Utilisation simple des pointeurs

Pointeur = variable qui contient l'adresse d'une variable. Il ***pointe*** cette variable.

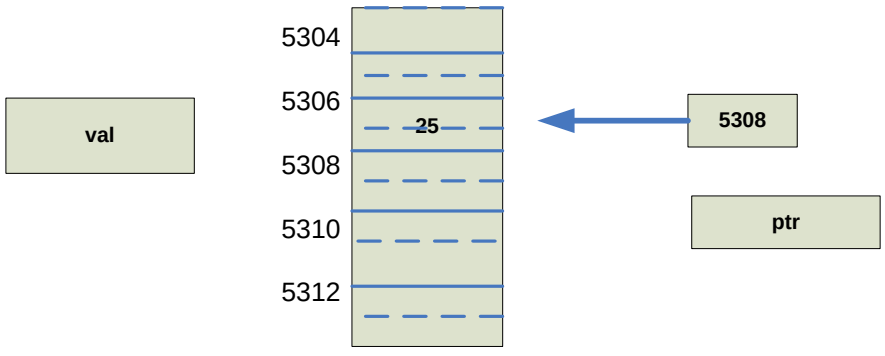
```
/* POINTEUR.C : Principe des pointeurs */
#include <stdio.h>
main ()
{
      int val;                  //déclaration de la variable "val"
      val = 25;                 //initialisation de la variable
      int *ptr;                 //déclaration du pointeur
      ptr = &val;               //initialisation du pointeur
      printf ("val = %d \n", val);
      printf ("*ptr = %d \n \n", *ptr);
      printf ("&val = %d \n ", &val);
      printf ("ptr = %d \n", ptr );
}
```

Résultat à l'écran :

val = 25	même valeur :
*ptr = 25	la valeur "val"
&val = 5308	même valeur :
ptr = 5308	l'adresse de "val"

Le programme crée une variable de type pointeur nommée "ptr". Ptr pointe vers une variable de type "int" nommée "val".

12.3.1. Relation entre "val" et "ptr" après initialisation



Tout pointeur pointe vers un objet de données présent en mémoire. L'espace mémoire peut être alloué:

- implicitement (en déclarant la variable. *Exemple :* int val → alloue 2 octets à val).
- explicitement (avec la fonction de bibliothèque ***malloc*** - allocation dynamique).

Utilisation des pointeurs de variables

Une fois déclaré et initialisé un pointeur peut :

a) Accéder à la valeur d'une variable.

b) Modifier le contenu d'une variable.

a) Soit le pointeur "ptr" qui pointe vers la variable "val".

Pour obtenir la valeur de "val", il y a deux méthodes :

- méthode directe qui utilise le nom de "val":

  printf ("val = %d \n", val);

- méthode indirecte qui utilise le pointeur ptr avec *:

  printf (" *ptr = %d \n", *ptr);

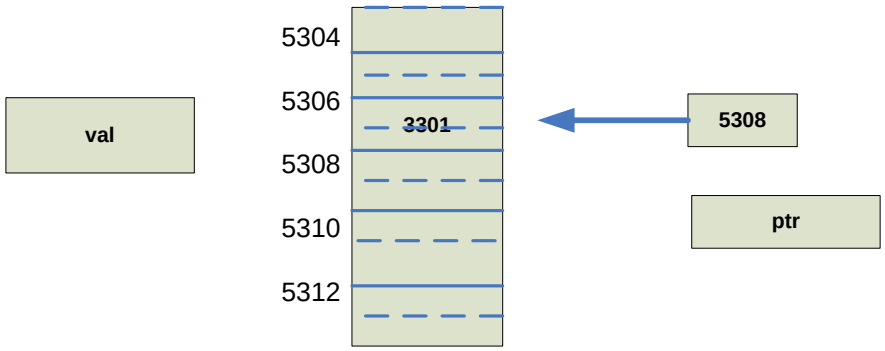
L'opérateur d'indirection (*) sert à accéder au contenu de val (il sert aussi à déclarer un pointeur).

Le pointeur *ptr peut être employé à la place de "val".

b) Ajoutons les instructions suivantes à la fin de POINTEUR.C :

*ptr = 3301	modification de la valeur pointée
printf ("%d \n", val);	le programme affiche 3301

12.3.2. Relation entre "ptr" et "val" après exécution des 2 instructions



La valeur contenue dans "val" a été changée indirectement avec le pointeur.

12.4. Passage d'un pointeur à une fonction

Le passage des variables à une fonction se fait :

a) par valeur.

b) par adresse.

a) La fonction reçoit et travaille sur une copie des variables; elle n'a pas accès aux originaux qui restent inchangés.

b) La fonction a accès aux variables originales qu'elle peut lire et modifier, même si elles sont locales à une autre fonction.

```
/* PFONC.C : passage de pointeurs à une fonction */

#include <stdio.h>

void echange (int *ptr1, int *ptr2);    //prototype.

main ()
{
       int premier = 1, second = 3;
       int *ptr = &second;
       printf (" premier : %d second : %d \n", premier, *ptr);
       echange (&premier, ptr);
       printf (" premier : %d second : %d \n ", premier, *ptr);
}

void echange (int *ptr1, int *ptr2)
{
       int temp = *ptr1;          //affecte la valeur 1 à "temp"
       *ptr1 = *ptr2 ;            //affecte 3 à premier (1 est écrasé)
       *ptr2 = temp;              //affecte 1 à second (3 est écrasé).
}
```

Affiche :

Premier : 1 Second : 3

Premier : 3 Second : 1

On remarque que les valeurs originales de val1 et val2 ne se retrouvent plus dans main.

Explications :

Prototype de "echange".

```
void echange (int *ptr1, int *ptr2);
```

Le prototype montre que la fonction "echange" ne renvoie rien et qu'elle attend 2 pointeurs donc deux adresses. Il s'agit des adresses des variables "premier" et "second" définies dans "main" (donc invisibles à la fonction "echange"). L'appel de la fonction échange se fait donc par ***adresse***:

```
echange (&premier, ptr);
```

On observe qu'il existe deux méthodes pour passer une adresse à une fonction :

- "&premier" (passer l'adresse de la variable avec l'opérateur &).
- "ptr" (passer le nom du pointeur qui contient l'adresse de la variable).

Les 2 arguments sont équivalents: ***ils passent une adresse.***

La fonction "echange" crée 2 pointeurs (*ptr1 et *ptr2) et leur affecte ces deux adresses dans l'ordre :

- ptr1 reçoit l'adresse de premier.
- ptr2 reçoit l'adresse de second (par le biais de ptr).

La fonction "echange" intervertit leurs valeurs à l'aide de la variable "temp";

Conclusion:

Le passage par adresse modifie les valeurs originales.

Voyons le même programme avec "passage par valeur".

```
#include <stdio.h>

void echange (int val1, int val2);            //le prototype

void echange (int val1, int val2)             //l'en-tête de la définition
{
      printf (" Avant l'echange: val1 = %d, val2 = %d \n", val1, val2);
      int temp = val1;
      val1 = val2;
      val2 = temp;
      printf ("Après l'echange : val1 =%d, val2 = %d", val1, val2);
}

void main ()
{
      int premier = 1, second = 3;
      echange (premier, second);
      printf ("Valeurs en fin de programme :");
      printf (" val1 = %d, val2 = %d", val1, val2);
}
```

Affiche :

Avant l'échange : val1 = 1, val2 = 3

Après l'échange : val1 = 3, val2 = 1

Valeurs en fin de programme : val1 = 1, val2 = 3

On remarque que les valeurs originales de val1 et de val2 sont retrouvées dans ***main*** n'étant pas affectées par les changements opérés dans "echange". On a modifié uniquement les *copies* de ces valeurs.

Principe de l'allocation dynamique de mémoire :

Variable statique :

- déclaration du nom (identificateur).
- déclaration du type (**int**, **float** etc.)
- elle peut être « globale » ou bien « locale » avec « static »
- L'espace mémoire (fixe) lui est alloué à la compilation.

Variable dynamique :

- Pas de déclaration explicite.
- Espace mémoire alloué sur demande explicite.
- Repérée par un pointeur qui contient son adresse.
- La valeur est générée à l'exécution du programme.

Utilité : pour les données de taille variable (inconnue à la compilation).

ALLOC.H = fichier avec des prototypes de fonctions pour :

- allouer de l'espace mémoire.

- libérer l'espace mémoire.

Allocation de place en mémoire avec les fonctions :

- ***malloc*** (**m**emory **alloc**ation):

  malloc(n); n = nombre d'octets à réserver dans le tas (*heap*).

- ***calloc*** (**c**onsecutive **alloc**ation):

  calloc (k, n) (k blocs consécutifs à n octets chacun).

Libération de la place en mémoire avec la fonction :

- ***free***(ptr) libère la zone mémoire si l'adresse contenue dans le pointeur "ptr" a été allouée avec ***malloc*** ou ***calloc***.

12.5. Création dynamique d'une variable

Variante du programme POINTEUR.C :

```
#include <stdio.h>
main ()
{
       int *ptr;
       ptr = (int *) malloc (sizeof(int));
       *ptr = 25;
       printf (" *ptr = %d \n", *ptr);
       printf (" ptr = %d \n", ptr);
}
```

La variable ptr n'est pas déclarée. Elle est crée par "***allocation dynamique***" de mémoire.

Explications :

- ***sizeof***(int) : retourne 2 (ce qui correspond au nombre d'octets d'un int).
- ***malloc***(2) : choisit 2 octets consécutifs de mémoire (libre) et retourne l'adresse du premier.
- (int *) : donne à cette adresse la signification d'un pointeur vers un type **int** (type casting).
- ptr = : l'adresse est stockée dans "ptr".

La valeur de cette variable est donnée par *ptr = 25.

Attention :

Affecter d'abord l'adresse à "ptr" et ensuite la valeur à " *ptr ".

Conversion de type

1. implicite :

   conversion exécutée automatiquement par le compilateur, en utilisant des règles de conversion précises (en général des "promotions" - voir le programme CONV.C).

2. explicite (avec *cast*) :

   conversion demandée explicitement par le programmeur avec l'opérateur unaire *cast* ("opérateur de conversion de type").

Syntaxe de Cast :

```
(nom_de_type) expression
```

L'expression est convertie dans le type précisé.

```
double sqrt( double );     // prototype de la fonction " sqrt "
int n ;
res = sqrt((double)n);     // appel de sqrt avec un cast qui convertit 'n' en double
.
```

Cette notation est utilisée dans l'allocation dynamique de mémoire (avec ***malloc***).

Prototype de malloc :

```
void *malloc (taille de n)
```

void * retourne un pointeur non initialisé (de type **void**) sur n octets de mémoire.
Si l'allocation échoue le ***malloc*** retourne NULL.
On fait ensuite un *cast* pour forcer explicitement la conversion **void** vers le type correct.

Exemple :

Fonction qui copie une chaîne s dans p :

```
char *copie (char *s)
{
       char *p;      // déclaration du pointeur ' p ' non - initialisé
       p = (char *) malloc(strlen( s ) + 1);   // strlen retourne le nombre de
                                               // caractères sans le ' \ 0' final
       if (p != NULL)       // si malloc n'a pas échoué
              strcpy ( p, s );    // s est copié  dans p
       return p;
}
```

La chaîne de caractères donnée en argument est copiée dans un espace mémoire alloué par ***malloc***. L'espace mémoire n'est pas alloué à la compilation (comme pour les tableaux de taille fixée) mais demandé en cours d'exécution et libéré après utilisation. L'allocation de mémoire est dynamique. C'est le cas des tableaux définis par des pointeurs (pointeurs de tableaux).

12.6. Utilisation avancée des pointeurs

12.6.1. Pointeurs de tableaux

Un tableau peut être déclaré de trois façons :

```
int x [ 10 ];
```

```
#define N 10
int x [ N ];
```

```
int *x;
x = (int *) malloc( 10 * sizeof(int));
```

Dans les cas 1et 2 il y a réservation automatique de mémoire.

Dans le 3ème cas la mémoire est allouée avec ***malloc*** pour 10 entiers. ***malloc*** retourne un pointeur (**void**) sur le début de la zone.

Le *cast* change le type **void** en **int**. 'x' est un pointeur sur le 1er élément. Il contient l'adresse du 1er élément.

Déclaration et initialisation simultanée :

```
int x [ 10 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

ou bien :

```
int x [ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Notation pointeur (tri d'un tableau unidimensionnel).

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

void trier (int n, int *x);

main ()
{
       int i, n, *x;
       printf ("\n Nombre de valeurs à trier ?");
       scanf ("%d", &n);
       x = (int *) malloc (n * sizeof (int));

       for ( i = 0; i < n; ++ i)
       {
              printf (" i = %d x = ", i + 1);
              scanf ("%d", x + i );
       }
       trier (n, x);
       printf ("i = %d x = %d \n", i +1,*(x + i ));
}




void trier ( int n, int *x)
{
```

```
	int i, el, temp;
	for (el = 0; el < n - 1; ++ el)
		for ( i = el + 1; i < n; ++ i )
			if ( *( x + i ) < * (x + el ))
			{
				temp = * (x + el);
				* (x + el) = * ( x + i);
				* (x + i) = temp;
			}
	return;
}
```

Notation tableau et pointeur

Un pointeur de tableau peut accéder (ou modifier) tout élément individuel d'un tableau.

```
/* PTBLEAU.C pointeur de tableau */

#include <stdio.h>

int tableau_i [ ] = {25, 300, 2, 12}; // declaration et initialisation simultanée.

main ()
{
	int *ptr;
	int compteur;
	ptr = &tableau_i [ 0 ];
// ou bien: int *ptr =&tableau_i [ 0 ]
	for (compteur = 0; compteur < 4; compteur ++)
	{
		printf ("tableau_i [ %d ] = %d", compteur; *ptr++);
	}
}
```

Affichage :

tableau_i [0] = 25

tableau_i [1] = 300

tableau_i [2] = 2

tableau_i [3] = 12

L'instruction ptr = &tableau_i [0]; affecte l'adresse du premier élément de tableau_i .

&tableau_i [0] est équivalent à tableau_i (l nom du tableau = pointeur sur le premier élément) donc ptr = &tableau_i [0]; équivaut à ptr = tableau_i;

12.6.2. Arithmétique des pointeurs

Incrémenter un pointeur de tableau c'est le faire pointer sur l'élément suivant du tableau.

L'adresse contenue dans " ptr " augmente de 2 unités (car chaque élément d'un tableau de **int** occupe 2 octets). Pour un tableau de **char** le compilateur ajoute 1 octet. Pour un tableau de **float** le compilateur ajoute 4 octets etc. Le compilateur tient compte du type pointé et ajuste l'adresse du pointeur incrémenté.

Décrémenter un pointeur de tableau le fait revenir à l'élément précédent (ptr--).

L'instruction ptr += 3; fait avancer le pointeur de 3 éléments mais d'un nombre variable d'octets en fonction du type des éléments.

Attention : il ne faut pas dépasser les limites du tableau (C ne vérifie pas les références des pointeurs de tableau).

12.6.3. Opérateurs arithmétiques appliqués aux pointeurs

1°) Pour modifier l'adresse : (arithmétique des pointeurs)

*ptr ++ (ou *ptr --) est équivalent à * (ptr +1); (respectivement à * (ptr-1);).

Note: * ptr ++ est équivalent à *(ptr++) car les opérateurs unaires ' ++' et ' * ' sont évalués de droite à gauche (voir tableau sur l'associativité).

En général : *(ptr + n); (nécessite les parenthèses car * est prioritaire sur +) incrémente (ou décrémente) l'adresse de ' n ' objets.

2°) Pour modifier la valeur de l'objet pointé :

* ptr + n; incrémente l'objet pointé de ' n ' unités (parenthèses inutiles car * est prioritaire sur +).

En particulier : * ptr + (ou -) 1 est équivalent à *ptr + (ou -) = 1

Notes: (*ptr)++ parenthèses nécessaires vu l'associativité D->G de ++ et * Sinon confusion avec *ptr++ équivaut à *(ptr + 1).

12.6.4. Relations entre les pointeurs et les tableaux

Le nom d'un tableau = pointeur sur le 1er élément.

```
int tab [ 5 ];   // déclaration d'un tableau de 5éléments.
&tab [ 0 ];      // = adresse du 1er élément.
tab = &tab[ 0 ]; // => expressions équivalentes. Les deux expressions contiennent
                 // l'adresse de tab [ 0 ].*
```

tab [i] est évaluée comme *(tab + i). Soit int *p = &tab [0]; // déclaration et initialisation d'un pointeur p.

La variable p est un pointeur vers le 1er élément du tableau.

```
p = &tab [ 0 ];     // expressions
p = tab;            // équivalentes.
```

<p + n> a comme valeur l'adresse de l'élément tab [n].

Expressions équivalentes pour désigner le 4ème élément du tableau "tab":

tab [3] p [3] *(p + 3) *(tab + 3)

Pointeurs de chaînes / tableaux de chaînes

Une chaîne peut être définie et affichée de 2 façons:

Tableau de caractères :
(Terminé par \0)

```
main(){
       char msg[30] ;
       strcpy(msg, « Bonjour »);
       puts(msg);
}
```

1
2
3

msg :

b	o	n	j	o	u	r	\Ø

1) Le compilateur réserve un espace mémoire pour 30 caractères.

2) Création de la chaine « Bonjour ».

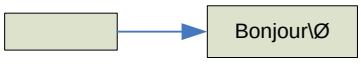
Strcpy copie la chaine dans l'espace mémoire vers lequel point msg (le nom d'un tableau est un pointeur vers l'adresse du 1er élément -> B)

3) L'adresse de B est passée à ***puts*** qui affiche la valeur stockée (B) et continue jusqu'au « \0 ».

Affiche : « Bonjour »

Pointeur de caractères :

```
main(){
       char *msg;
       msg = « Bonjour »;
       puts(msg);
}
```

msg : 

1°) création d'un pointeur non-initialisé et sans espace mémoire alloué

2°) création de la chaine « Bonjour\0 »

l'adresse de B est affectée à msg (initialisation du ptr)

puts affiche un à un les caractères de « B » à « \0 ».

Affiche : « Bonjour »

L'avantage de la méthode pointeur est qu'elle n'oblige pas de préciser l'espace mémoire, le compilateur adapte la longueur automatiquement selon la taille de la chaîne.

Remarque :

Il y a 2 façons (équivalentes) pour accéder aux caractères d'une chaîne:

1°)Incrémenter l'adresse du pointeur :

a°) *ptr ++ → voir programme Pchaine.c

b°)*(nom + compte) → voir programme Pchaine3.c

ou nom = nom du pointeur et compte = indice (1, 2, 3…)

a et b sont équivalents car :

*ptr ++ = (ptr + 1)

(ptr + compte) = (ptr + 1) .....*(ptr + 2)…

2°)Incrémenter un indice dans une notation tableau :

Exemple :

Nom [compte] → voir programme Pchaine2.c

2 est équivalent a 1 (a et b) car le nom d'un tableau est un pointeur sur l'élément indexé.

Attention : *(nom + compte) est différent de * nom + compte car le premier incrémente l'adresse alors que le second incrémente l'objet pointé

```
/*PROGRAMME PCHAINE1.C : Pointeur et chaîne*/
#include <stdio.h>

main ()
{
	char nom [ ] = "paul";
	char *ptr = nom;
	while (*ptr)
		printf ("*ptr = %c \n", *ptr ++ );
}
```

Affiche :

*ptr = p

*ptr = a

*ptr = u

*ptr = l

L'expression conditionnelle de la boucle **while** (*ptr) est évaluée comme vraie jusqu'à ce que "ptr" pointe vers le caractère nul '\0' qui marque la fin de la chaîne. L'instruction conditionnelle peut s'écrire: **while** (*ptr != 0)

```
/* PROGRAMME PCHAINE2.C : Tableau et chaîne  */

#include <stdio.h>
#include <string.h>

main()
{
	int compte;
	char nom [ ] = "paul";
	for (compte = 0; compte < strlen (nom); compte ++)
		printf ("nom [ %d ] = %c \n", compte, nom[ compte ]);
}
```

Affiche :

Même affichage que celui du programme PCHAINE1.C

```c
/* PROGRAMME PCHAINE3.C : Equivalence entre les pointeurs et les tableaux*/
#include <stdio.h>
#include <string.h>

main()
{
	int compte;
	char nom [ ] = "paul";
	for (compte = 0 ; compte < strlen (nom); compte ++)
		printf ("*(nom+compte)= %c \n", *(nom + compte));
}
```

Affiche :

*(nom+compte) = p

*(nom+compte) = a

*(nom+compte) = u

*(nom+compte) = l

Programme pour copier une chaîne dans une autre.

A°) Version tableau :

```c
void main ()
{
	int i = 0;
	char src [ ] = "bonjour";		//la chaîne source
	char dest [ ] = "au revoir les copains";	//la chaîne de destination
	while ( (dest [ i ] = src [ i ]) != '\0')
		i++;
	puts (dest);
}
```

B°)Version Pointeurs :

```c
void main()
{
	char *src, *dest;
	int i = 0;
	src = "bonjour";
	dest = "au revoir les copains";
	while ((*dest = *src) != '\0')
	{
		src ++;
		dest ++;
		i ++;
	}
	puts (dest-i); // dest – i pour ramener le pointeur au début de la chaîne.
}
```

Variante de la boucle while :

```c
while (*dest ++ = *src++) i ++;
```

12.6.5. Tableau de pointeurs

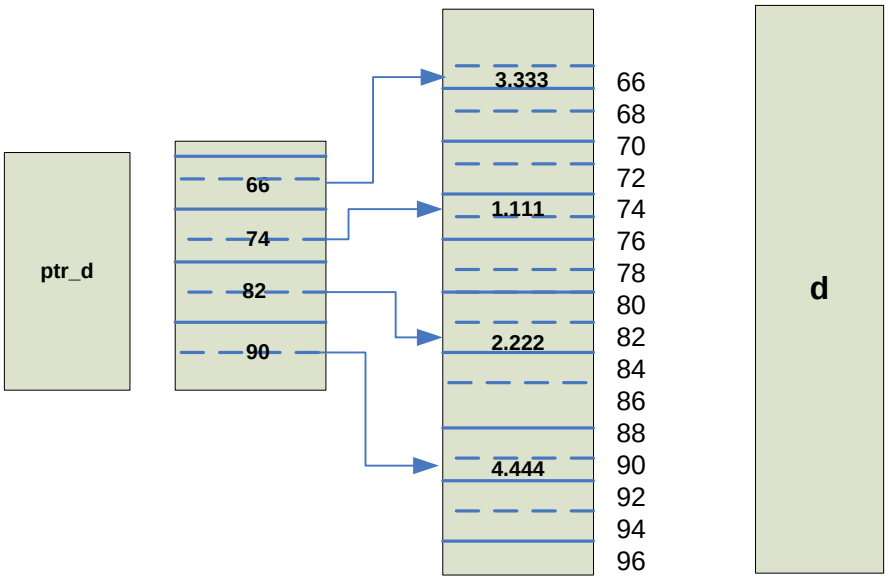
Un tableau de pointeur est une liste de pointeurs (variables du même type) utilisée pour :

- Accélérer certaines opérations (exemple : le tri).
- Pour stocker les chaînes de caractères de différentes longueurs.

```
#include <stdio.h>
#define TAILLE 4

void tri (int taille, double *p[ ]);    //*p représente un tableau de pointeurs

void affiche (int taille, double *p[ ], double d[ ]);

main ()
{
       int x;
       double d [ ] = {3.333, 1.111, 2.222, 4.444};
       double *p[ TAILLE ];
       for (x = 0; x < TAILLE; x ++)
              p[ x ] = &d [ x ];
       affiche (TAILLE, p, d);
       tri (TAILLE, p);
       affiche (TAILLE, p, d);
}

void affiche (int taille, double *p[ ], double d [ ])
{
       int x;
       printf ("__________");
       for (x = 0; x < taille; x ++)
       {
              printf ("*p[ %d ] = %1.3f", x,*p[ x ]);
              printf ("p [ %d ] = %u", x, p[  x ]);
              printf ("d [ %d ] = %1.3f \n", x, d[ x ]);
       }
}

void tri (int taille, double p[ ])      // p[ ] est un tableau de pointeurs et la
{                                       // variable p est un pointeur de pointeur.
       int x, x1;
       double *temp;
       for (x = 0; x < taille-1; x ++)
              for (x1 = x+1; x1 < taille; x1 ++)
              {
                     if (*p[ x ] > *p[ x1 ])
                     {
                            temp= p[ x1 ];
                            p[ x1 ] = p [ x ];
                            p[ x ] = temp;
                     }
              }
}
```

12.6.6. Le tri de tableau de pointeurs

Aspect des pointeurs avant le tri du tableau de pointeurs



Création du tableau (d) de 4 éléments de type double (8 octets chacun).

Création du tableau (ptr_d) de 4 pointeurs vers les doubles (un pointeur occupe 2 octets de mémoire pour leurs adresses).

Initialisation des pointeurs du tableau ptr-d: ptr_d [x] = &d [x]; pour pointer vers les éléments du tableau d.

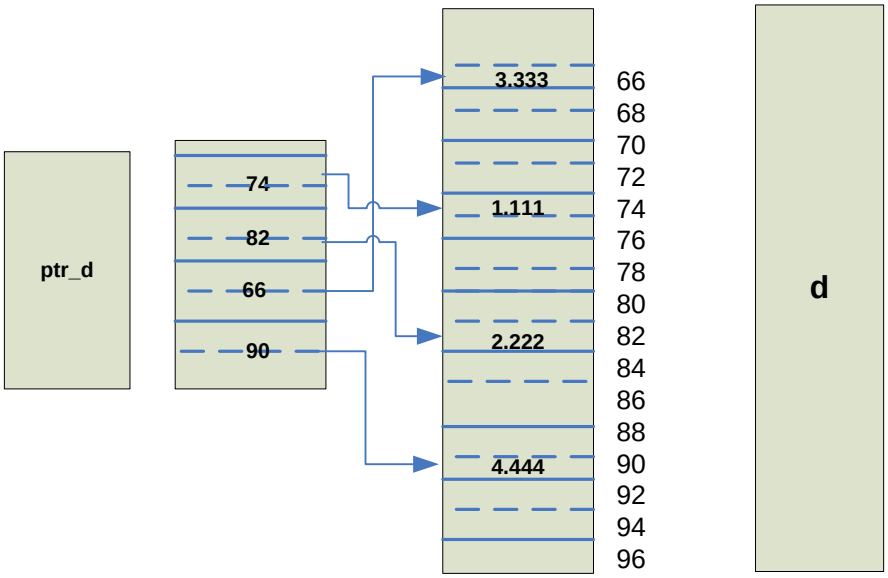
La fonction "affiche" affiche :

La valeur référencée par chaque pointeurs.

L'adresse affectée à chaque pointeur;

La valeur de chaque élément du tableau d.

Aspect des pointeurs après le tri du tableau de pointeurs



Il est préférable de trier les pointeurs (2 octets chacun) que les doubles du tableau (8 octets chacun) : moins de mémoire manipulée.

Déclaration de "tri" : **void** tri (**int** taille, **double** *p [ ]); ou *p [ ] = pointeurs vers un tableau de pointeurs.

Quand le programme appelle " tri " il passe la taille de tableau (TAILLE) et un pointeur de tableau de pointeur (ptr_d) : ptr_d => nom de tableau sans indice (pointeur vers le 1er élément.

Tri effectue le tri des pointeurs (selon leur valeur référencée).

Appel à "affiche" pour afficher le résultat de "tri".

Attention : Ne pas confondre pointeurs de tableau et tableau de pointeurs.

Autre utilisation des tableaux de pointeurs:

2°)Stockage des chaînes de caractères dans un tableau à 2 dimensions ou dans un tableau de pointeurs :

A°) Tableau à 2 dimensions :

déclaration et initialisation : char tab [ ] [12] = {"Jours payés", "Lun", "Mar", "Mer"};

tab

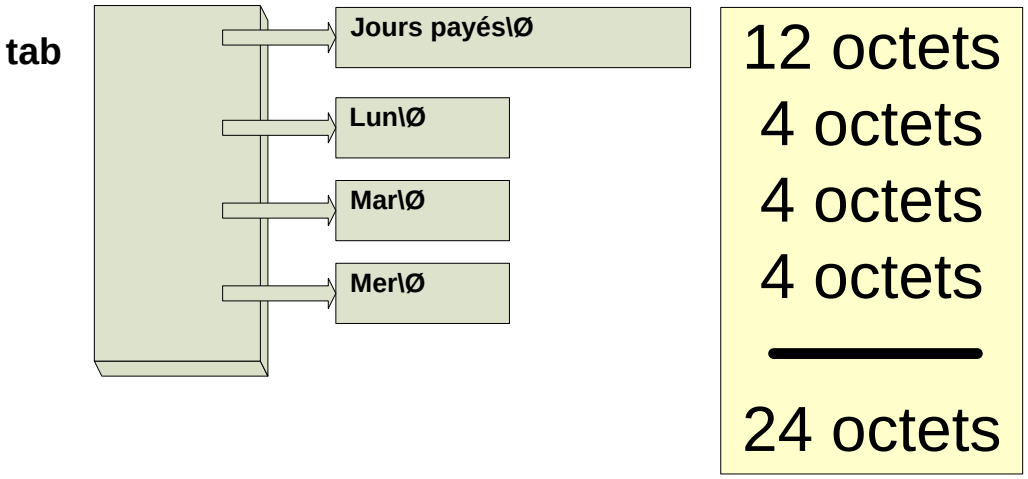
Jours payés\Ø	Lun\Ø	Mar\Ø	Mer\Ø
0	12	24	36

Le compilateur réserve 12 octets pour chaque ligne (de 0 à 11) (le nombre de lignes résulte de l'initialisation – pas besoin de le préciser dans la déclaration)

Donc : 4*12= 48 octets réservés (gaspillage de mémoire).

B°) Tableaux de pointeurs:

Déclaration et initialisation : char *tab [ ] = {"Jours payés", "Lun", "Mar", "Mer"};



Avantage : les pointeurs pointent sur des vecteurs de longueurs différentes d'ou une économie de mémoire.

12.7. Pointeur de pointeur

Un pointeur peut pointer vers n'importe quel type de variable, donc vers un autre pointeur aussi (pointeur = variable)

```
/* PROGRAMME PTR_PTR.C: Principe d'un pointeur de pointeur */
#include <stdio.h>

main ()
{
       int val = 501;
       int *ptr = &val;          //ptr contient l'adresse de val
       int **ptr_ptr = &ptr;     //ptr_ptr contient l'adresse de 'ptr'.
       printf ("val = %d \n", **ptr_ptr);
       printf (" ptr= %u \n", *ptr_ptr);
}
```

Affiche :

Val = 501

Ptr = 2000

** = double indirection.

Le pointeur "ptr_ptr" se réfère à "ptr". Les pointeurs "ptr" se réfèrent à "val". La double indirection permet à "ptr_ptr"d'accèder à "val".

*ptr_ptr accède au contenu de 'ptr' (adresse de val).

**ptr_ptr accède au contenu de cette adresse (val).

```
/*PROGRAMME TRI.C: Principe du tri avec des notation de pointeurs et utilisation
des pointeurs de pointeurs*/
#include <stdio.h>
#define TAILLE 4
```

```
void tri (int taille, double **p);
void affiche (int taille, double **p, double dd[ ]);

main ()
{
	int x;
	double d[ ] = {3.333, 1.111, 2.222, 4.444};
	double *ptr_d [ TAILLE ];
	for (x =0; x < TAILLE; x ++)
		ptr_d [ x ] =& d [ x ];
	affiche ( TAILLE, ptr_d,d);
	tri (TAILLE, ptr_d);
	affiche (TAILLE, ptr_d,d);
}

void tri (int taille, double **p)
{
	int x; x1;
	double *temp;
	for (x = 0; x < TAILLE-1; x ++)
		for (x1 = x+1; x1 < taille; x1 ++)
		{
			if (**(p+x) > **(p+x1))
			{
				temp = *(p+x1);
				*(p+x1) = * (p+x);
				*(p+x) = temp;
			}
		}
}

void affiche (int taille, double **p, double dd [ ])
{
	int x;
	printf ("_________________");
	printf ("_________________");
	for (x = 0; x <taille; x++)
	{
		printf (" *ptr_d [ %d ] = %1.3f", x, **(p+x));
		printf (" ptr_d [ %d ] = %u", x,*(p+x));
		printf ("d[ %d ] = %1.3f \n", x, dd [ x ]);
	}
}
```

TRI1.C utilise la notation pointeurs

TRI.C utilise la notation tableau et on obtient le même résultat.

On peut donc choisir entre les deux. La notation pointeurs est préférable pour manipuler les tableaux.

Remarque :

Dans le programme TRI.C on a:

void tri (**int** taille, **double** *p[ ]);

*p[ ] représente un tableau de pointeurs car p est le nom du tableau qui lui est un pointeur vers le 1er élément (qui est un pointeur) donc p est un pointeur de pointeur : (**p) => TRI1.C

12.8. Pointeur de fonction

Le pointeur contient l'adresse de la fonction. Il pointe vers la fonction.

Application usuelle : Passer une fonction comme argument à une autre fonction.

```c
/*PROGRAMME FONCTIONPTR.C : Création d'un pointeur de fonction*/
#include <stdio.h>
main ()
{
      int (*ptr_fonct) ();       // déclaration du pointeur de fonction "ptr_fonct"
      ptr_fonct = &puts;         // initialisation avec l'adresse de "puts" le &
                                 // peut être omis.
      (*ptr_fonct)("Salut \n"); // appel indirect de "puts" identique à puts
                                 //("Salut \n");
}
```

Affiche :

Salut

Explications :

ptr_fonct est de type "int " car "puts" retourne un "int" (le '\0' de fin de chaîne).

Règle générale : un pointeur de fonction a le même type que celui retourné par la fonction.

int (*ptr_fonct) () ; les parenthèses finales montrent que le pointeur pointe vers une fonction.

De plus les parenthèses entourant *ptr_fonct sont nécessaires car sinon on aurait int *ptr_fonct (); qui correspondrait à la déclaration d'une fonction qui retourne un pointeur d'entiers.

Exemple de passage de pointeurs de fonction comme arguments

```c
/* PROGRAMME FONCPTR1.C*/
#include <stdio.h>

void donne_fonct (void (*ptr_fonct)());              //déclaration de donne_fonct
main ()
{
      donne_fonct(&puts);   // & peut être omis car le compilateur s'attend
                            // à une adresse.
}

void donne_fonct (void (*ptr_fonct)())        //définition de donne_fonct
{
      (*ptr_fonct)("A votre santé!");
}
```

Affiche :

A votre santé!

Explications :

donne_fonct s'attend à recevoir comme arguments un pointeur de fonction (déclaration).

main appelle "donne_fonct" avec l'adresse de la fonction de bibliothèque "puts".

13. Manipulation des fichiers sur disque

Un fichier stocke indéfiniment (sur disque ou disquette) les données d'un programme.

Les 2 noms d'un fichier :

- externe (physique) - reconnu par le DOS (chaîne de caractères)

Ex: FICHIER.C

- interne au programme (logique) - variable de type fichier

*Ex.: FILE *toto* (la variable "toto" est un pointeur sur un objet de type **FILE** - en majuscules)

13.1. Les principes de base

Il faut d'abord ouvrir le fichier (créer un pointeur de **FILE**). Mêmes fonctions E/S que pour le clavier et l'écran, mais :

Précédées de ' f ' (ex. : ***fprintf***). Ajout du pointeur de **FILE** dans les arguments. Un fichier doit être fermé à la fin pour vider la mémoire tampon (le buffer).

Le système d'exploitation (MS-DOS) ne peut ouvrir que N fichiers à la fois. Le nombre N est fixé dans la commande :

FILE = xxx du fichier config.sys

N_{max} = 20 (selon les versions du MS-DOS)

```c
/* OUVERT.C - illustre l'ouverture d'un fichier */

#include <stdio.h>

main()
{
       char Nom[20];
       FILE *toto;                                    // Nom interne : toto

       printf ("\n\t\ Nom du fichier : ");            // Nom externe saisi
       scanf (" %20s", Nom);
       toto = fopen (Nom, "wb");                      // Ouverture du fichier
       if (!toto) printf ("\a\n\t Ouverture de %s ratée!", Nom);
       fclose (toto);
}
```

Détails :

- stdio.h: contient tous les prototypes des fonctions sur les fichiers
- ***fopen*** : fait la liaison entre le nom externe et le nom interne du fichier.
  - 1er arg. : nom externe (avec extension) +chemin d'accès ;
  - 2e arg. : les actions autorisées (lecture (r), écriture (w), etc.) le mode dans lequel est écrit le fichier (binaire (b) , texte (t))

- ***fopen*** retourne l'adresse d'un fichier FILE ; cette adresse est affectée à "toto" ; si échec d'ouverture, fopen retourne NULL.
- ***fclose*** : ferme le fichier et vide le tampon de la mémoire

Remarque : on ne peut ouvrir en lecture (r) un fichier nouveau.

13.2. Actions autorisées sur les fichiers

r (read)	Ouvre un fichier existant pour la lecture
w (write)	Crée et ouvre un fichier pour l'écriture ; tout fichier existant est écrasé
a (append)	Ouvre un fichier pour l'ajout d'écritures (extension d'un fichier existant). Si le fichier n'existe pas, il crée un nouveau.
r+	Ouvre un fichier existant pour la lecture et l'écriture le fichier doit exister obligatoirement
w+	Lecture et écriture sur un nouveau fichier ; curseur en début de fichier
a+	lecture et ajout de données à la fin d'un fichier existant

L'option 'b' ou 't' placée en deuxième position, ex. : "*rb*" ou "*wb*" ou "*a+ t*" ; indique si le fichier est en format texte (t) ou binaire (b). Par défaut le format est "texte" :

"w" → "wt"

"r" → "rt"

```c
/* ECRIFICH.C : Crée et écrit dans un fichier disque. */
#include <stdio.h>

int main(void)
{
	FILE *fp;

	if ((fp = fopen( "c:\\tc\\fichtest.asc", "w" )) != NULL ) // si fopen réussit
	{
		fputs( "Exemple de chaîne", fp );
		fputc( '\n', fp );
		fclose( fp );
	}
	else
		printf( "message d'erreur\n" );
	return 0 ;
}
```

Explications :

- La variable fp est un "pointeur de FILE"
- ***fopen*** : ouvre en écriture et en mode texte le fichier : fichtest.asc. Remarque. : Turbo.C convertit les deux barres \\ en une seule \ ;si l'on pose une seule: "c:\fichtest.asc" il y a confusion avec le caractère d'échappement ' \ f' (saut de page)
- ***fopen*** : retourne l'adresse du fichier →affectée à fp (si empêchement d'ouvrir →fopen retourne NULL) ;
- ***fputs*** écrit la chaîne dans le fichier pointé par fp sans le caractère nul de fin de chaîne et sans saut de ligne ;
- ***fputc*** renvoie le caractère « saut de ligne » à la même adresse ;
- ***fclose*** ferme le fichier

```c
/* LITFICH.C : Lit un fichier et affiche les caractères à l'écran. */
#include <stdio.h>

int main(void)
{
	int c;
	FILE *fp;

	if( fp = fopen( "c:\\tc\\fichtest.asc", "rb" ) )
	{
		while( (c = fgetc( fp )) != EOF )
			printf( " %c\t%d\n", c, c );
		printf("Repère de fin de fichier: %x", c ); // %x -> ascii héxadécimal
		fclose( fp );
	}
	else
		printf( "Erreur d'ouverture du fichier\n" );

	return 0 ;
}
```

Explications :

- ouverture pour lecture en mode binaire ;
- si la valeur affectée à fp est ≠ 0 → test évalué à VRAI
- "tant que" fgetc retourne un car ≠ EOF → la boucle while tourne
- EOF (End Of File) = constante symbolique (=-1 sous Dos) ;
- La boucle while agit sur une seule ligne (puisque sans accolades) ;
- affiche le caractère lu :
- comme caractère (%c)
- comme nombre décimal (%d) →son code ascii
- Quand c = EOF →sortie de while →second printf →affichage de EOF en %d → -1

Affichage du programme LITFICH.C

En mode binaire		En mode texte	
Caractères (%c)	Ascii décimal (%d)		
E	069	E	069
X	120	x	120
E	101	e	101
M	109	m	109
P	112	p	112
L	108	l	108
E	101	e	101
	032		32
d	100	d	100
e	101	e	101
	032		32
C	099	c	099
h	104	h	104
a	097	a	097
î	148	i	148
n	110	n	110
e	101	e	101
	13		10
	10	Repère de fin de fichier : -1	
Repère de fin de fichier : -1			

En mode binaire il reste 2 caractères après la chaîne : 13 et 10

En mode texte il n'en reste qu'un seul: 10

EOL sous DOS :

$10_{10} = 0A_{16}$ = saut de ligne (LF – Line Forward)

$13_{10} = 0D_{16}$ = retour chariot (CR – Carriage Return)

EOL :

End Of Line (Caractère de fin de ligne)

Fichiers texte et fichiers binaires

Les formats (binaire et texte) sont deux façons de représenter des valeurs numériques. Les modes (binaire / texte) sont les paramètres passés à la fonction "fopen".

Un entier en format binaire occupe (généralement) 2 octets sur disque. Un entier en format texte peut occuper un ou plusieurs octets.

Exemple 1 : le chiffre 5 (pris comme caractère) → 1 octets // texte

Le nombre –25439 (6 caractères) → 6 octets // texte

→ 2 octets // binaire

Exemple 2 : 12345.678 → occupe 8 octets pour les nombres

1 octet pour le point décimal

1 octet pour séparer les variables

10 octets en format texte

12345.678 → float → 4 octets en format binaire

Conclusion 1 : le format binaire économise de l'espace sur le disque.

Pour lire et écrire des valeurs numériques en format texte il faut traduire les valeurs ASCII en format binaire interne. Ces traductions sont évitées en mode binaire.

Conclusion 2 : le format binaire économise du temps machine.

La traduction binaire → ASCII →binaire altère la précision.

Conclusion 3 : Le format binaire assure la précision en virgule flottante.

Conclusion 4 : Sauvegarde plus rapide d'un tableau ou structure avec "fwrite"

Conclusion générale : le format binaire est avantageux dans le cas des fichiers numériques

14. Fonctionnalités avancées

14.1. Arguments de la ligne de commande

La fonction " main " à deux arguments utilise les tableaux de pointeurs pour récupérer les arguments passés en ligne de commande lors du lancement du programme :

```
int main ( int argc, char *argv[ ]) ;
```

argc (argument count) = nombre d'arguments de la ligne de commande

argv (argument vector) = tableaux d'arguments = pointeur sur un tableau de pointeurs de chaînes

```
// Programme ARG.C - arguments de la ligne de commande
#include <stdio.h>

void affiche( char *argument );

int main(int argc, char *argv[ ])
{
   int i;
   for ( i = 1; i < argc; i++ )
      affiche( argv [ i ] );
   return 0;
}

void affiche( char *argument)
{
   printf( " %s \n ", argument );
}
```

Le programme est compilé (sans évaluer **argc** et **argv**), l'exécutable ARG.EXE est créé. C'est en lancement sous DOS en ligne de commande l'exécutable que vous pourrez donner la valeur aux arguments :

Exemple :

c:\ arg toto titi

arg représentant le nom de l'exécutable et toto et titi les arguments optionnels.

argc = 3 (évalué par le compilateur)

argv [0] = arg (nom de l'exécutable)

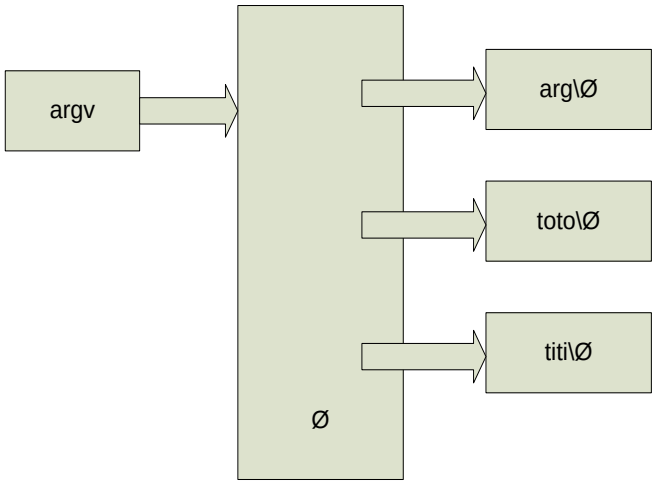
argv [1] = toto (1er argument optionnel)

argv [2] = titi (2^{éme} argument optionnel)

argv [3] = ∅ (ptr. null – fin des chaînes)

Affichage :

toto
titi



Variante 1 :

```
...
int main( int argc, char *argv [ ] )
{
   int i ;
   for( i = 1; i < argc; i++ )
      printf("%s %s ", argv [ i ], (i < argc - 1) ? " " : "" );
   printf("\n");
   return 0;
}
```

Affichage :

toto, titi

Variante 2 :

```
...
int main( int argc, char *argv [ ] )
{
       while( --argc  > 0 )
              printf( "%s %s ", *++ argv, (argc > 1) ? " " : "" );

       printf( "\n" );
       return 0;
}
```

Variante de " printf " :

```
printf(( argc > 1  ) ? "%s " : "%s", *++argv );
```

Exemple de commande : c:\ arg les vacances arrivent

Résultat à l'écran : les vacances arrivent

```
// Programme ARGFICH.C
// commande (sous DOS) pour ouvrir et afficher le contenu d'un fichier texte
// ---------------------------------------------------------------------

#include <stdio.h>

void main (int argc, char *argv [ ])
{
       FILE *f;
       int ch;
```

```
        f = fopen (argv[1], "rb");

        if (f == NULL)
        {
        printf ("Ne peut ouvrir le fichier %s\n", argv[1]);
        return (1);
        }
        while ( (ch = getc ( f ) ) != EOF )
        putch (ch);
        fclose ( f );
}
```

Exemple de commande: c:\ argfih toto.c

Si toto.c n'est pas dans le même répertoire que argfich.c, on indique le chemin : c:\ argfih c:\tc\tp5\toto.c

Exemple :

```
// Programme ATOIFICH.C - commande (sous DOS) pour ouvrir un fichier en
// lecture et afficher le n-ième caractère d'une chaîne qui s'y trouve
// ---------------------------------------------------------------------

#include <stdio.h>

void main (int argc, char *argv[ ])
{
        int n = (argc > 1) ? atoi (argv [ 1 ] ) : 1 ;          // par défaut n = 1;
        char c;
        int i;
        FILE *fp;

        if( fp = fopen ( "c:\\tc2\\sources\\fichtest.asc", "r" ))
        // on introduit le nom du programme qui
        {                                          // contient la chaîne
                for (i=1; i <= n; i++)
                        c = fgetc ( fp );
                putch (c);
                fclose( fp );
        }
        else
                printf( "Erreur d'ouverture du fichier\n" );
}
```

atoi : fonction de la bibliothèque stdio.h qui convertit une chaîne en un entier.

Prototype : **int** atoi(**const char** *chaine)

Retourne : un entier ou 0

Exemple de commande : c:\ atoifich 9

Cela retourne le 9éme caractère de la chaîne contenue dans fichtest.asc

14.2. La programmation de bas niveau

La programmation dite de bas niveau a cette appellation car le langage est proche du langage machine (assembleur), la vitesse des opérations est donc accrue.

14.2.1. Les variables de classe registre

Registre = zone de mémoire dans l'unité centrale (registres du processeur).

Var. registre ≈ var automatic (même portée locale). L'opérateur d'adressage '&' est inopérant. Ces variables sont déclarées avec le mot réservé : " register ".

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

Si pas de place dans les registres → automatic / par défaut. " Register " associé à des variables entières (et pointeurs) et aux arguments d'une fonction.

```
void fonc (register int u, register int *p) ;        // prototype
void main ()
{
      register int u, *p ;
      u = 5 ; *p = 12 ;                  // l'entier pointé par p a la valeur 12
      fonc (u, p) ;                      // l'appel de « fonc »
}

void fonc(register int u, register int *p)     // définition
{
...
}
```

Exemple d'emploi des variables " registres " :

La suite de Fibonacci – version itérative

```
#include <stdio.h>
#include <time.h>   // contient les fonctions time, difftime

void main() ;
{
      time_t debut, fin ; // les variables "debut" et "fin" sont de type time_t
      int compteur, n = 23 ;
      long int boucle, max = 10 000 000 ;
      register int f, f1, f2 ;
      time(&debut) ;

      for(boucle = 1 ; boucle <= max ; ++boucle)
      {
            f1 = 1 ; f2 = 2 ;
            for(compteur = 1 ; compteur <= n ; ++ compteur)
            {
                  f = (compteur < 3) ? 1 : f1 + f2 ;
                  f2 = f1 ; f1 = f ;
            }     // Explications : on répète 10.10^6 fois le calcul
      }           // des 23 termes pour avoir un temps de calcul
      --compteur ; // plus grand.
      time(&fin) ;
      printf("i = %d F = %d \n", compteur, f) ;
      printf("temps écoulé : %.01f secondes", difftime (fin, debut)) ;
}
```

Affiche :
i = 23 F = 28657
Temps écoulé : 37 secondes.

14.2.2. Opérateurs au niveau du bit (*bitwise*)

Cette section détaille l'utilisation des opérateurs aux niveaux binaire que vous avez déjà vus dans les chapitres précédents. Ils traitent les données de type **int** au niveau des bits. Ils ne peuvent être dissociés du niveau binaire.

&	ET au niveau du bit	(AND)
|	OU inclusif au niveau du bit	(OR)
^	OU exclusif au niveau du bit	(XOR)
<<	décalage à gauche	(Shift left)
>>	décalage à droite	(Shift right)
~	complément à un	

ET (bitwise) : (&) compare les bits de même rang de ses deux opérandes et retourne une valeur correspondant à la table de vérité du ET logique :

\	**0**	**1**
0	0	0
1	0	1

Exemple 1:

255 & 15 = 15

255	1 1 1 1 1 1 1 1	⇒
& 15	0 0 0 0 1 1 1 1	⇒
= 15	0 0 0 0 1 1 1 1	⇒

Exemple 2:

18 & 14 = 2

18	0 0 0 1 0 0 1 0	⇒
& 14	0 0 0 0 1 1 1 0	⇒
= 2	0 0 0 0 0 0 1 0	⇒

L'opérateur & est utilisé pour forcer certains bits à 0.

OU inclusif bit à bit : (|) →table de vérité du ou logique :

\	**0**	**1**
0	0	1
1	1	1

Exemple 1:

255 | 15 = 255

255	1	1	1	1	1	1	1	1	⇒
| 15	0	0	0	0	1	1	1	1	⇒
= 255	1	1	1	1	1	1	1	1	⇒

Exemple 2:

18 | 14 = 30

18	0	0	0	1	0	0	1	0	⇒
| 14	0	0	0	0	1	1	1	0	⇒
= 30	0	0	0	1	1	1	1	0	⇒

L'opérateur | est utilisé pour forcer certains bits à 1.

OU exclusif bit à bit : (^)

\	**0**	**1**
0	0	1
1	1	0

Exemple 1:

255 ^ 15 = 240

255	1	1	1	1	1	1	1	1	⇒
^ 15	0	0	0	0	1	1	1	1	⇒
= 240	1	1	1	1	0	0	0	0	⇒

Exemple 2:

18 ^ 14 = 28

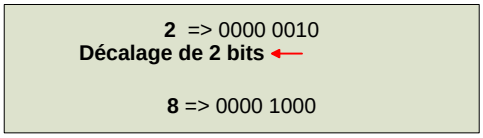
18	0	0	0	1	0	0	1	0	⇒
^ 14	0	0	0	0	1	1	1	0	⇒
= 28	0	0	0	1	1	1	0	0	⇒

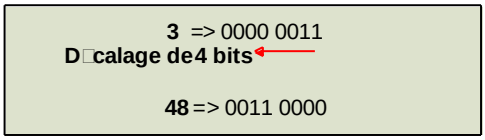
14.2.3. Décalage à gauche (<<) ou à droite (>>)

Décalage de l'opérande de gauche selon la valeur indiquée par l'opérande de droite (toujours en binaire).

Exemple 1: 2 << 2 = 8

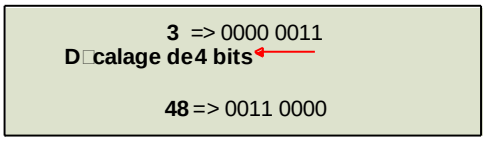
(2 * 2^2 = 8)





Exemple 2: 3 << 4 = 48

(3 * 2^4 = 48)

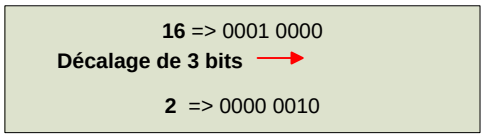


Utilisation de (<<) :

Multiplication rapide par une puissance de 2.

Exemple 3: 16 >> 3 = 2

(16 / 2^3 = 2)



Utilisation de (>>) :

Division rapide par une puissance de 2.

14.2.4. L'opérateur ~

Cet opérateur est également utilisé par le complément à 1.

Exemple 1: $(2)_{10} = (0010)_2$ ~2 = 1101 = $C_1(2)$

$(3)_{10} = (0011)_2$ $C_2(3) = 1101 = C_1(2) = -3$

complément à 2 de 3

Exemple 2: $(6)_{10} = (0110)_2$

$C_1(6) = \sim 6 = 1001$

$(7)_{10} = 0111$

$C_2(7) = 1001 = C_1(6) = -7$

En général :

$C_1(x) = C_2(x+1) = -(x+1)$

Car :

$x + C_1(x) = 2^n - 1 \quad \Rightarrow \quad C_1(x) = 2^n - 1 - x$

$x + C_2(x) = 2^n$

$x + 1 + C_2(x+1) = 2^n \quad \Rightarrow \quad C_2(x+1) = 2^n - 1 - x$

Quand on lance ~x la machine retourne –(x+1) car elle interprète le $C_1(x)$ comme le $C_2(x+1)$

14.3. Convention de représentation

14.3.1. Le complément à 1

Chaque bit est inversé.

Inconvénients :

- 0 a une double représentation : C1(00…0) = 11…1 .
- la somme $x + (\sim x) \neq 0$ (=11…1 n fois).

Exemple :

Sur 16 bits :

```
  113 =  0 0 0 0  0 0 0 0  1 1 1 0  0 0 0 1 +
 ~113 =  1 1 1 1  1 1 1 1  0 0 0 1  1 1 1 0
 ---------------------------------------------------
         1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1
```

En général : $x + C_1(x) = 2^n - 1 = (111\ldots1)_{n\ bits}$

14.3.2. Le complément à 2

Egalement appelé « unanimement accepté », il faut que $x + C_2(x) = 0$ (2^n sur n bits)

Solution : ajouter 1 à $C_1(x)$

$x + C_1(x) + 1 = 2^n - 1 + 1 = 2^n = x + C_2(x)$ (=00…0)

Règle pratique pour obtenir $C_2(x)$:

on laisse inchangé les bits à droite du 1er 1 inclus.

on inverse le reste.

Exemple :

```
x :       0 1 0 1 0 1 0 0 +
C2(x) :   1 0 1 0 1 1 0 0
          ------------------------------
```

1 0 0 0 0 0 0 0 0

14.4. Directives de compilation

Transmises directement au préprocesseur (sans traduction en langage machine).

Commandes traitées avant la compilation (pré-compil).

Commencent par # (dièse) et ne comportent pas de ; final.

Buts :

- Insérer le contenu d'un fichier (#include).
- Effectuer des substitutions de texte (#define)
  - constantes symboliques.
  - des macros paramétrées.
- Guider la compilation du programme avec des directives conditionnelles :
- #if
- #else
- #elif
- #endif

14.4.1. La directive #include

#include <nom_de_fichier.h>

fichier inséré = fichier « en-tête » (header → *.h)

14.4.2. La directive #define

1°) Constantes symboliques : #define PI 3.14

A chaque occurrence PI est remplacé par 3.14 et ceci est valable dans tous les fichiers.

undef PI (pour annuler)

2°) Les macros :

#define <nom> <texte de remplacement>

A chaque occurrence <nom> est remplacé par <texte>

Exemple :

#define max (x,y) ((x)>(y) ? (x) : (y))

```
void main()
{
	int a = 4 ; int b= 7 ;
	printf ("%d ", max (a, b)) ;	/* affiche : 7 */
}
```

printf appelle la macro ″max″ avec les paramètres a et b .

le compilateur remplace max(a, b) par sa définition.

Avantage : rapidité d'exécution par rapport à l'appel d'une fonction qui implique les instructions :

- Sauvegarde de l'état courant ;
- Recopie des valeurs des arguments ;
- Branchement avec conservation de l'adresse de retour ;
- Recopie de la valeur de retour ;
- Restauration de l'état courant ;
- Retour dans le programme.

Désavantage : perte d'espace mémoire car instructions générées (par le pré-processeur, en C) à chaque appel. Les instructions correspondent à une fonction → générée 1 seule fois par le compilateur.

« Effets de bord » indésirables et quelques fois imprévus.

14.4.3. Effet de bord d'une macro

A°) Omission des parenthèses des arguments de la macro

```
#include <stdio.h>
#define QUATREX (arg) (arg * 4)

void main()
{
	int val ;
	val = QUATREX (2+3) ;
	printf("val = %d\n", val) ;	// affiche val=14 (au lieu de 20)
}
```

La directive QUATREX (2+3) est développée par le pré-compilateur en val = 2+3*4=14 (car * est prioritaire sur +).

Solution : #define QUATREX (arg)((arg)*4) → val = (2+3)*4

B°) Utilisation d'opérateurs d'incrémentation ou de décrémentation dans les arguments d'une macro

```
#include <stdio.h>
#define max(x,y) ((x)>(y) ? (x) : (y))

void main()
{
	int a = 4 ; int b= 7 ;
	printf(" %d ", max(a++, b++)) ;  /* affiche : 8 */
}
```

Les macros évaluent 2 fois leurs arguments.

14.4.4. Directives conditionnelles

Conditionnent la compilation d'une partie du fichier.

#if et #endif → fonctionnent comme l'instruction " if "

#else et #elif → fonctionnent comme l'instruction " else"

Exemple 1:

```
#define VRAI 1
#define SOLUTION VRAI
…
#if (SOLUTION == VRAI)                  // évalué lors de la compilation
      affiche(informations) ;           // compilée si SOL == VRAI
#endif                                  // sinon, saut à endif
```

Exemple 2:

Différentes versions de programmes pour différents types d'IBM PC.

```
#define VRAI 1
#define FAUX 0
#define XT VRAI
#define AT FAUX
...
#if XT == VRAI
      #define DNIVEAU 0
      #include <XT.H>
#elif AT == VRAI
      #define DNIVEAU 1
      #include <AT.H>
#else
      #define DNIVEAU 2
#endif
```

14.5. Programmes multi-fichiers

Les variables « externes » peuvent être :

- définies dans un fichier (sans extern) ;
- déclarées (avec « extern ») et utilisées dans un autre fichier.
- si définies avec « static » dans une fonction, alors :
  - la durée de vie = au programme
  - la visibilité est locale (limitée à la fonction)

" Static " appliqué aux variables extern (globales) limite leur visibilité au fichier où elles sont déclarées.

Donc, les fonctions dans les programmes multi-fichiers peuvent être **extern** ou **static**.

« extern » → utilisable dans tout le programme.

« static » → utilisable dans le fichier uniquement.

Sans mention, une fonction est **extern** par défaut.

14.6. Visibilité d'une variable dans plusieurs fichiers sources (C)

Règle générale : une variable est visible uniquement dans le fichier source où elle est déclarée.

Le mot réservé **extern** élargit la visibilité à d'autres fichiers.
Le mot réservé **static** limite la visibilité d'une variable globale, au seul fichier où elle est déclarée.

Les fonctions sont externes (visibles) par défaut. Si l'on ajoute 'extern', c'est pour la lisibilité.

STATIC limite la visibilité au fichier où elle apparaît.

```c
/* EXTERN1.C – illustre la visibilité dans plusieurs fichiers */
int toto = 20, titi = 30 ;
extern void ailleurs(void) ;

void main()
{
	ailleurs() ;
}
```

```c
/*EXTERN2.C */
#include <stdio.h>
void ailleurs(void)
{
	extern int toto, titi ;
	printf("toto = %d, titi= % d\n", toto, titi) ;
}
```