C Facile

Abdelkader BELAHCENE

abelahcene@gmail.com

13 septembre 2011

Table des matières

1 Concepts Généraux **3**
1.1 Programmer sous Linux . . . 3
1.2 Généralités . . . 4
1.3 Structure d'un Ordinateur . . . 5
1.4 Représentation de l'Information . . . 7
1.5 Exercices . . . 9

2 Le Langage C Simple **10**
2.1 Introduction . . . 10

3 Concepts fondamentaux **13**
3.1 Identificateur . . . 13
3.2 Expression et Instruction . . . 15
3.3 Entrées-Sorties Standards . . . 20
3.4 Fonctions . . . 21
3.5 Les Tableaux . . . 22
3.6 Exercices . . . 23

4 Concepts Avancés **25**
4.1 Pointeurs . . . 25
4.2 Tableaux et Pointeurs . . . 26
4.3 Les Structures . . . 30
4.4 Traitement de Fichiers . . . 33
4.5 Exercices . . . 36

5 Création de Bibliothèque **38**
5.1 Programme Multi-fichiers . . . 38
5.2 Archivage de Fonctions . . . 39
5.3 Exercices . . . 40

1 Concepts Généraux

1.1 Programmer sous Linux

Pourquoi utiliser Linux ? parce que Linux est un systeme libre gratuit, solide et pratiquement sans virus (c'est un open source). Il fait fonctionner des machines aussi bien toute petite , genre pc-386 que les grands ordinateurs « super-computer ». Nous donnons ci-après les avantages essentielles :

1. Liberté d'utilisation et de choix de logiciels (ou libre choix de l'outil)
2. Liberté de modification (compléter améliorer et redistribuer)
3. Liberté d'aider son prochain (travail coopératif) grace aux innombrables forums lies a chaque logiciel et sites d'aide.
4. Liberté de choisir son fournisseur
5. Abandonner le piratage des logiciels avec tous les inconvénients qui y sont liés (virus, espionnage, illégalité...).

Lorsque votre machine démarre, si vous avez plusieurs systèmes installés, choisissez la ligne correspondant à Linux.

vous obtenez alors une invite pour vous logez, donnez votre nom utilisateur puis le mot de passe.

Pour pouvoir programmer, vous avez besoin de 2 applicataions : un éditeur de texte et un terminal. Dans le menu des applications, vous choisissez terminal et emacs. plusieurs editeurs pourraient etre installés, mais emacs est plus puissants et plus pratique pour la programmation.

Vous pouvez aussi faire des raccourcis sur le bureau a vos applications.

Quelques Commandes Utiles

Bien que vous pouvez utiliser les outils graphiques et la souris pour faire ce qui suit, il est utile de connaitre quelques commandes de base. Pour s'en convaincre regardons ce qui suit :

Exemple 1. Supposons vouloir convertir une liste d'images (disons 120 images ! ! !) gif en image png. Avec un outil graphique d'images (imagemagick par exemple), nous sommes obligés d'ouvrir les images une a une et de les exporter dans le nouveau format. Avec la commande il suffit de faire une boucle :

```
for i in $(ls *.gif) ; do
    convert $i.gif $i.jpeg
done
```

mkdir monRep Crée le répertoire monRep, dans le répertoire courant. mkdir=make directory

pwd donne le repertoire courant. pwd=path working directory

cd monRep aller vers répertoire monRep. cd= change directory

ls Liste le répertoire courant, ls=list

ls -lrt monRep Donne la liste des fichiers de monRep avec les détails (option -l) trie par temps (-t) en ordre inverse (-r)

cp f1.txt monRep Copie le fichier f1.txt dans monRep. cp = copy

Exercice 2. Personnalisez votre bureau : ajouter des boutons de fermeture de la barre des menus

Insérer un bouton de clavier avec (fr, us, ar)

Ajouter un raccourci pour emacs et le terminal

Exercice 3. Ouvrir l'éditeur emacs, créer un nouveau fichier, avec le nom prog1.c, taper les lignes suivantes (sans fair d'erreur), sauvegarder le fichier [1]. Utilisez la touche de tabulation (touche avec double flèche à gauche) pour arranger automatiquement le texte.

```
#include <stdio.h>
int main(){
   printf("\n Salam alaykoum ya ahla el kheir \n");
}
```

Exercice 4. Ouvrir un terminal, compiler le programme avec la commande suivante : `gcc prog1.c -o prog1`, maintenant executer votre programme(noubliez pas le « ./ ») `./prog1` et vous avez le résultat suivants :

```
Salam alaykoum ya ahla el kheir
```

Exercice 5. Toujours dans le terminal, creez un repertoire nomme mesProg. Faites pwd pour voir votre repertoire de travail. listez le contenu de mesProg, ensuite copier votre programme prog1 et prog1.c dans mesProg. Listez de nouveau mesProg. Allez dans ensuite dans mesProg, refaites pwd puis listez avec details.

Ouvrez maintenant le gestionnaire graphique des fichiers et retrouvez vos fichiers.

1.2 Généralités

Dans ce chapitre nous allons donné les éléments de base pour la compréhension de la circulation de l'information lors de l'exécution d'un programme informatique.

Nous verrons aussi les différents circuits qui interviennent pour mieux comprendre le principe de réalisation et d'exécution d'une application scientifique de gestion.

Ces concepts de base sont utiles pour apprendre à bien programmer et réaliser des logiciels précis, concis et performant, surtout avec un langage aussi puissant que le langage C.

Une application informatique peut être décomposée en :

1. Acquisition des données en entrée.
2. Traitement de l'information.
3. Stockage de l'information.

Pour réaliser ces fonctions, il est nécessaire de disposer des trois éléments suivants :

1. Ordinateur : Partie matérielle (ou hardware, en anglais).
2. Logiciel : Programmes fournis par le constructeur, ou écrits par l'utilisateur, permettant de doter le matériel d'une intelligence.
3. Utilisateur : Fournit les données en entrées et reçoit les résultats.

Pour faire la paie des employés d'une entreprise il faut :
- Un ordinateur possédant les organes d'entrée (clavier, disque ou disquette...) et de sortie (imprimante, ou écran,...)
- Un logiciel (programme de calcul et d'édition) permettant de faire le calcul du salaire de chaque employé et de faire la sortie de la fiche de paie.
- L'information concernant les employés (le grade, la catégorie, les absences, poste, ancienneté, situation familiale, etc.).

Dans ces trois fonctionnalités on retrouve : l'acquisition de la donnée en entrée (information concernant les employés), le traitement de l'information réalisé par le programme de calcul de la paie, le stockage des résultats sur papier à remettre aux employés ou sur disquette pour des calculs ultérieurs.

1. ATTENTION a la casse : majuscule et minuscule sont différentes, Prog1 et prog1 ne sont pas identiques

1.3 Structure d'un Ordinateur

Les trois principales parties de tout ordinateur sont :

1. L'unité principale
2. Les unités secondaires
3. Les accessoires

1.3.1 Unité Principale

Elle se compose de la mémoire centrale (M.C) et de l'unité de traitement et de calcul (U.T.C).

Mémoire centrale L'information doit transiter au moins temporairement par la mémoire centrale (avant et après traitement). La mémoire centrale est l'endroit ou doit être stockée l'information avant qu'elle ne soit traitée.

L'unité de traitement et de calcul est l'ensemble des circuits qui sont spécialisés dans les calculs. Tous les traitements se ramènent aux opérations élémentaires arithmétique, logique et de transfert (chargement ou rangement de la donnée en mémoire).

Pour réaliser une opération quelconque sur une donnée, il faut envoyer la donnée, stockée au préalable en mémoire, vers l'unité de calcul qui fait alors les traitements nécessaires puis renvoie le résultat vers la mémoire centrale. Le schéma ci-dessous en donne une illustration.

Faire la moyenne de deux notes n1 et n2 et récupérer le résultat dans une zone notée moy.

1. les notes n1 et n2 (supposées stockées en mémoire centrale) sont d'abord envoyées en Unité de Traitement et de Calcul (U.T.C).
2. L'U.T.C fait les opérations nécessaires (addition et division) puis renvoie le résultat vers la mémoire centrale.
3. Le résultat est stocké dans la zone appelée moy. Toutes ces étapes de traitement de l'information sont contrôlées par l'U.T.C.

1.3.2 Unités Secondaires

Définition 6. Les mémoires de masse de masse sont des mémoires pouvant stocker de grande quantité d'information de façon permanente (l'information n'est pas perdue à la fin des programmes ou à l'arrêt de l'ordinateur).

Les supports les plus répandus actuellement sont les disques et disquettes magnétiques. Il y a aussi les bandes magnétiques qui tendent à disparaître et l'arrivée des disques laser, ou encore plus recents les clés usb.

Pour avoir une idée de la capacité de stockage, il suffit de savoir qu'une petite disquette de 1.44 Mega-octets peut contenir un gros dictionnaire de 1000 pages (en caracteres standards, sans image). Les disques durs (incorporés dans la machine) ont une capacité beaucoup plus grande, leur capacité dépassent les 800 Mega-octets sur les micro-ordinateurs. Leur capacité devient de plus en plus grande. Ces mémoires de masse peuvent servir d'entrée ou de sortie de grande quantité d'information.

1.3.3 Autres périphériques

On peut citer :

1. Le clavier, qui est l'organe essentiel d'entrée manuelle des informations.
2. L'écran, imprimante, table traçante, scanner etc... qui peuvent recevoir l'information en sortie.
3. Les mémoires de masse décrites précédemment (bande, disque dur, clé usb, disquette...) sont des organes de lecture et d'écriture (Entrées/Sorties).

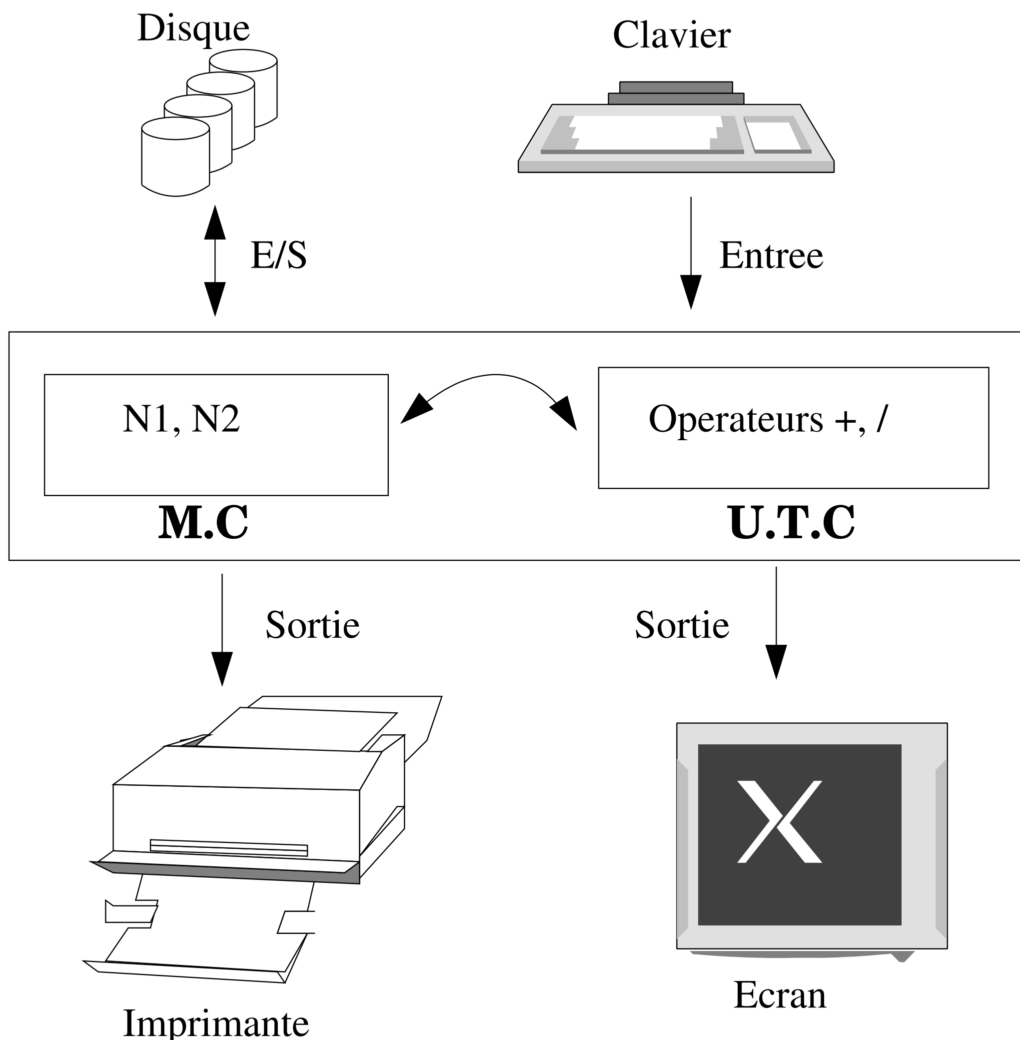


Figure 1.3.1: Schema d'un ordinateur

4. Accessoires avec l'évolution rapide de la technologie on peut relier à l'ordinateur une multitude d'instruments en commençant par la souris jusqu'à un complexe industriel en passant par les capteurs optiques, les robots, etc.

Le transfert d'information d'un périphérique vers la mémoire centrale s'appelle lecture Le transfert inverse s'appelle une écriture voir figure 1.3.1

1.3.4 Adressage

Une instruction élémentaire est formée :
– D'une partie code d'opération,
– D'une partie adresse,
– Et éventuellement d'une partie opérande (sur laquelle se déroule l'opération).
Chaque opération élémentaire est codifiée (par exemple l'addition, le chargement, etc.). L'adresse donne l'endroit en mémoire ou il faut mettre ou récupérer l'opérande.

Faire l'addition C = A + B. Soient ad1, ad2, ad3 les adresses mémoires des variables A, B, C. Soient R1, R2 les registres de l'unité de traitement. Un registre est une zone mémoire spéciale se trouvant dans l'unité de traitement et non pas en M.C.
L'addition peut se ramener, globalement, aux instructions élémentaires suivantes :

```
Charge (ad1) R1
Charge (ad2) R2
Ajout R1 R2
SAUVE R2 (ad3)
```

Tout traitement doit transiter par l'unité de traitement, c'est elle qui doit coordonner toutes les actions. En quelque sorte c'est le cerveau de l'ordinateur.

Une adresse permet d'accéder à une zone mémoire. La taille de la zone adresse dépend de la mémoire que l'on veut adresser. Par exemple si la mémoire est faite de 1 MB (lire un MegaByte = 220 octets = environ 1 million d'octets) il faut une zone adresse de 20 bits

1.4 Représentation de l'Information

1.4.1 Représentation de Caractères

Pour pouvoir traiter l'information il faut d'abord la codifier, la codification la plus répandue pour les caractères est le code ASCII (Américain Standard Code for Information Interchange). Les 128 caractères standards (Alphabétiques, numériques et de ponctuation) sont codifiés, de plus dans l'extension IBM on retrouve les caractères semi-graphiques et étranger (par rapport à l'anglais) à partir du code 128. Par exemple

- Les chiffres 0 à 9 ont pour code ASCII de 48 à 57.
- Les lettres Majuscules A à Z ont pour code ASCII de 65 à 90.
- Les lettres minuscules a à z ont pour code ASCII de 97 à 122.

Il y a une trentaine d'années la mémoire centrale était formée d'anneaux magnétiques (tors de ferrite) se magnétisant dans un sens pour indiquer 1 et dans l'autre sens pour indiquer 0 selon le sens du courant qui le traverse. Actuellement les mémoires sont formées de semi-conducteurs, mais le principe reste le même, une polarisation indique 1 et une autre indique 0.

Le regroupement de plusieurs anneaux (8 anneaux forment un octet ou Byte) permet la représentation d'un caractère.

Si les tors (L1, C2) et (L1, C8) sont positionnés sur 1, alors ils codifient le caractère A de code 65 c'est à dire en binaire (0100 0001). De même le positionnement des tors (ou Bits) (L2, C3) et (L2, C4) à 1 correspond au caractère 0 (et non pas la valeur 0) de code ASCII 48 ou en binaire (0011 0000). Ainsi on peut représenter n'importe quel caractère en mémoire.

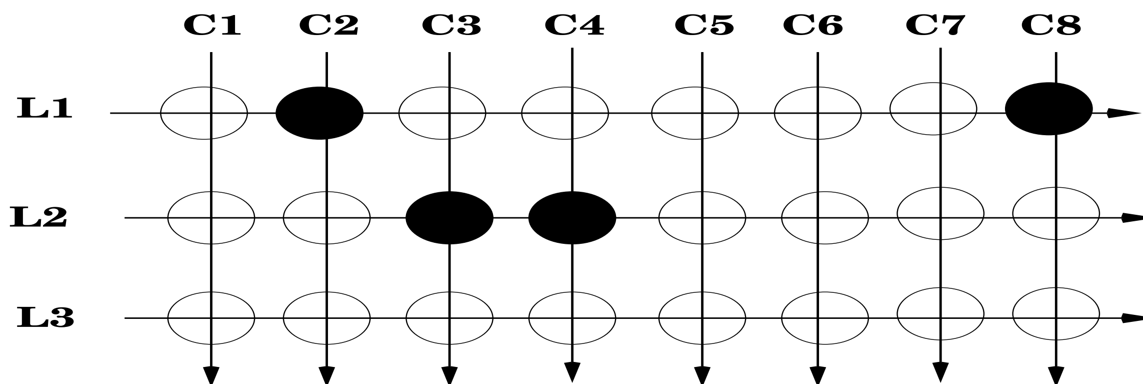


FIGURE 1.4.1: Structure de la mémoire d'ordinateur

Actuellement, on utilise les semi-conducteurs au lieu des tors de ferrite, mais le principe de la représentation reste le même. On peut considérer la mémoire comme un ensemble de cases prenant chacune la valeur 0 ou 1. (voir figure 1.4.1)

1.4.2 Système de Numération

Les nombres sont représentés de plusieurs façons : s'ils sont entiers ils sont codifiés en binaire virgule fixe, s'ils sont réels en binaire virgule flottante. (Nous verrons plus loin la signification de ces mots).

Un nombre quelconque peut être représenté dans une base quelconque et pas uniquement dans la base 10. La forme générale est :

$$\alpha_{-m}B^{-m} + \cdots + \alpha_{-1}B^{-1} + \alpha_0 B^0 + \cdots + \alpha_{-n}B^{-n}$$

B est la base et α_i sont les chiffres qui écrivent le nombre. Si m = 0, le nombre est entier, si m est non nul alors le nombre est réel non entier. Les bases les plus utilisées sont :

Base 2 Deux symboles sont nécessaires$\alpha_i = 0$ ou 1.

Base 8 8 symboles sont nécessaires $\alpha_i = 0, \cdots, 7$.

Base 10 10 symboles sont nécessaires $\alpha_i = 0, \cdots, 9$.

Base 16 16 symboles sont nécessaires $\alpha_i = 0, \cdots 9, A, \cdots, F$.

Le nombre 199 écrit dans différentes bases :

$199 = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11\,000\,111_{(2)}$

$199 = 3 \times 8^2 + 0 \times 8^1 + 7 \times 8^0 = 307_{(8)}$

$199 = 12 \times 16^1 + 7 \times 16^0 = C7_{(16)}$

1.4.3 Représentation des données

Représentation d'un Caractère

Est représenté dans un octet (8 bits), en utilisant le code ASCII. Voir la table ASCII. par exemple le code de "A" est de 65 ans, de "0" est de 48 "$" est de 36, de "a" est de 97. Le code est stocké dans une représentation binaire fixe, qui est une liste de 0 et de 1, de façon à ce que "$" est représentée par la chaîne binaire : 00100100. Un personnage n'a pas de signe, le code est positif.

char x = 'A' ;

si nous imprimons la variable x comme entier (le code), la chaîne "65" est affiché, si nous imprimons "x" comme un caractère alors "A" est affichée.

Représentation des Nombres Entiers

Un nombre entier est stocké dans la mémoire en 32 bits, tel que fixé représentation. Le nombre positif est un format binaire, tandis que nombre négatif est représenté dans l'ordre que A + (-A) = 0. Par exemple, en 16bits 1256 est représenté par 0000 0100 1110 1000 et -1256 est représenté par 1111 1011 0001 1000, de sorte que la somme des 2 donne le nombre 0, en binaire 1 0000 0000 0000 0000, ce qui est de 0, puisque le premier "1" est perdue.

Représentation des Nombres Réels

Les nombres réels sont représentés dans le format flottant. Le flottant est un simple mot (32 bits : 1,8,23 est respectivement de 1 bit pour le signe, 8 bits pour l'exposant et 23 bits de mantisse) la mantisse est suppose commencer a 1 qui n'est pas representé. Le nombre réel double précision est représenté sur 64 bits (64 bits : 1, 11, 52) selon la norme IEEE 754.

Voici la representation des nombre 1.25 en mémoire sur 32 bits :

```
0 01111111  010 0000 0000 0000 0000 0000
```

- bit 0 : signe (0 pour positif , 1 pour négatif)
- bits 1-8 : exposant : valeur =127 [3] , represente l'exposant 0
- bits 9-31 : mantisse 1.010...0 [4] est 1.25 s $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$qui est 0.9375.

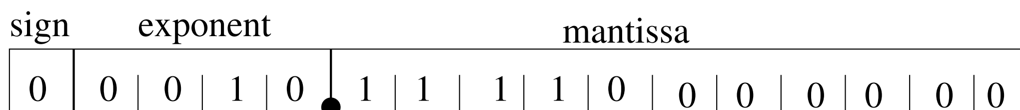
Le nombre -0.125 est representé par

```
1 01111100 000 0000 0000 0000 0000 0000
```

3. le nombre 127 est ajouté
4. Le 1 est cache non represente

Co	Ca	Co	Ca	Co	Ca	Co	Ca	Co	Ca	Co	Ca	Co	Ca	Co	Ca
0	null	16	nd	32	sp	48	0	64	@	80	P	96	‘	112	p
1	nd [2]	17	nd	33	!	49	1	65	A	81	Q	97	a	113	q
2	nd	18	nd	34	”	50	2	66	B	82	R	98	b	114	r
3	nd	19	nd	35	#	51	3	67	C	83	S	99	c	115	s
4	nd	20	nd	36	$	52	4	68	D	84	T	100	d	116	t
5	nd	21	nd	37	%	53	5	69	E	85	U	101	e	117	u
6	nd		nd	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	nd	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	nd	40	(	56	8	72	H	88	X	104	h	120	x
9	TAB	25	nd	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	nd	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	nd	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	nd	44	,	60	<	76	L	92	]	108	l	124	|
13	CR	29	nd	45	-	61	=	77	M	93	\	109	m	125	}
14	nd	30	nd	46	.	62	>	78	N	94	ˆ	110	n	126	˜
15	nd	31	nd	47	/	63	?	79	O	95	_	111	o	127	DL

Table 1.4.1: ASCII code (Co=Code, Ca=Car)



- bit 0 : signe (1 pour negatif)
- bits 1-8 : exposant : valeur =124 represente 124-127=-3,
- bits 9-31 : mantisse : 1.00...0 est 1 . Le nombre est $-1 \times 2^{-3} = -0.125$

1.5 Exercices

1. Donner la représentation en memoire de la chaîne de caractères “Salamou Alaykoum”.
2. Ecrire dans des bases 2, 8 et 16 , les nombres 38, 234, 1482.
3. Donner les valeurs (en base 10) des nombres $ff_{(16)}$, $8109_{(16)}$, $17757_{(8)}$,$1110010110_{(2)}$.
4. Représenter sur 32 bits, les nombres entiers : 38, -38, 1482, -1482.
5. Représenter les nombres réels sur 32 bits selon la norme IEEE 754 (1 bit pour signe, 8 bits pour exposant et 23 bits pour la mantisse) : 32.25, 0.125, 0.1, 325.3125, 51.1875
6. Donner la représentation de la valeur 6.0, et quels sont ces adjacents. Quels sont le plus grand et le plus petit nombres reéls positifs représentable, pour les cas simple et double précision ?

2 Le Langage C Simple

2.1 Introduction

Le langage C est un langage évolue comme le Pascal, le Fortran. Il est proche du Pascal mais il est plus maniable et plus puissant, surtout pour la programmation bas niveau, les structures et les unions.

Un langage évolue est un langage (proche de l'homme) avec lequel on écrit plutôt des macro-instructions que des instructions élémentaires.

Le langage assembleur, par exemple, est plus proche de la machine mais l' écriture de programmes dans ce langage est difficile et fastidieuse. On écrit dans ce langage des fonctions très largement utilisées et devant être rapides, optimisées, et utilisées par d'autres langages plus évolues.

Un problème réel ne peut être résolu et programme que s'il passe par les différentes phases suivantes : (Voir figure (2.1.1)) .

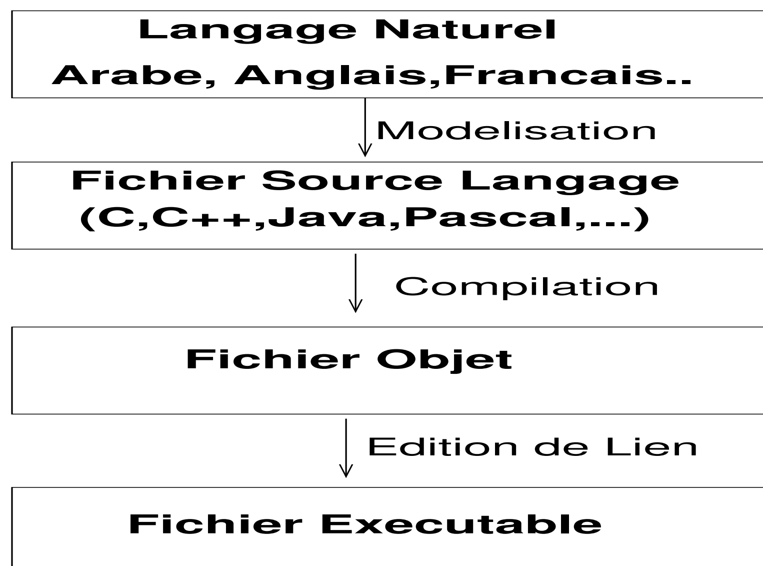


Figure 2.1.1: Modélisation de Programme

2.1.1 Modélisation

Avant d' écrire un programme code, il faut d'abord ramener le problème concret et réel qui est posé dans un langage naturel, à une suite logique de fonctions ou de procédures concises. Chacune fait une fonctionnalité précise, bien déterminée et sans ambiguité. C'est la partie Modélisation qui donne le programme source.

2.1.2 Compilation

Lorsque les objectifs du programme a écrire sont connus, Il faut les traduire en un programme dans un langage évolué (par exemple en C). Ce programme n'est pas encore exécutable par l'ordinateur. Il faut le traduire en une LANGUE comprise par l'ordinateur : c'est la phase compilation qui donne le programme objet.

2.1.3 Edition de lien

Mais le programme objet n'est pas encore exécutable, il est uniquement le programme source traduit mot à mot, et qui contient uniquement des appels à des fonctions de librairie. Il faut ensuite incorporer le code de toutes ces fonctions, offertes par la bibliothèque du langage. Parfois encore le programme lui m même peut être composé de plusieurs modules compilés indépendamment les uns des autres.

Pour ces raisons, une phase d'édition de lien doit être effectuée pour faire le lien entre toutes les parties et les fonctions de librairies utilisées par le programme. C'est l'édition de liens ou link (en anglais) qui fournit alors le programme exécutable.

2.1.4 Structuration de Programme

Un programme en C est formé de fonctions dont une principale appelée main. C'est la première fonction exécutée. Si le programme est formé de plusieurs modules compilés indépendamment, il faut qu'un et un seul module contienne la fonction main.

Un programme bien structuré et fortement commenté est plus facile à maintenir, à modifier ou à adapter à de nouvelles situations non prévues initialement.

Pour bien structurer un programme, il faut définir ses principaux axes, puis décomposer chacun de ces axes en d'autres axes secondaires, et ainsi de suite jusqu'à arriver aux instructions élémentaires. C'est certainement la phase la plus difficile pour l'écriture d'un programme, mais si cette architecture descendante est bien réussie, on aboutit à un programme performant.

Soit à écrire un programme lisant une chaîne de caractères à partir du clavier, la triant et sortant sur écran la chaîne triée. Faisons l'analyse de ce programme. Définissons-en les grandes lignes.

1. Lecture des données (lecture de la chaîne de caractères).
2. Procédure de tri (tri de la liste en entrée et obtention d'une nouvelle liste).
3. Sortie des résultats (sortir la liste triée

Ces trois fonctions sont appelées par fonction principale main.

```
main(){
  lecture(Liste1);
  tri(Liste1, Liste2);
  sortie(Liste2);
}
```

Maintenant il faudrait définir de façon précise le rôle de chacune des fonctions. Les fonctions lecture et sortie se ramènent à quelques instructions simples de lecture et d'écriture que le langage possède. Par contre, la fonction tri doit être encore détaillée. Le tri d'une chaîne de caractères peut se ramener à :
- Rechercher le minimum de la liste en entrée.
- Mettre le minimum dans une nouvelle liste.
- Mettre à jour la première liste en enlevant le minimum.
- Rechercher le minimum de la première liste (déjà mise à jour).
- Répéter cette procédure tant qu'il y a des éléments dans la première liste.

Notons que nous avons une chaîne en entrée et une autre en sortie : la chaîne triée.
La fonction tri peut être écrite comme suit :

```
tri {
  Faire{
    min = Min (Liste1 ); /* Récupère le caractère minimum */
    Ajoute (Liste2, min); /* Création de la liste 2 */
    Supprime(Liste1, min); /* Mise à jour de la liste 1 */
  } Tantque(Liste1 nonVide); /* Restent des caractères */
}
```

La fonction tri appelle :
- La fonction Min qui renvoie le caractère minimum de Liste1 (récupéré dans min).
- La fonction Ajout qui ajoute le caractère min dans Liste2.
- La fonction Supprime qui enlève le caractère min de Liste1.

Cette procédure **continue tant que** la liste en entrée Liste1 n'est pas vide. La procédure tri nécessite des zones de travail (Liste1, Liste2, ...) qu'il faut définir.

Il est évident que ce programme n'est pas complet (voir les exemples de programme en fin de chapitre).

Un programme ne s'écrit pas entièrement au départ, on le complète au fur et à mesure qu'on avance dans sa réalisation, et qu'on détaille l'écriture des fonctions.

Au départ donner uniquement les liens entre les fonctions, c'est à dire les entrées et les sorties de chaque fonction.

Dans l'exemple précédant, dans main par exemple, il est nécessaire de savoir que la fonction Lecture doit nous renvoyer la chaîne lue, la fonction tri doit recevoir la liste initiale et nous renvoyer la liste triée. De même dans la fonction tri, la fonction Min reçoit la liste et renvoie son minimum.

Il est clair que lors de l'écriture de la fonction tri, on ne s'intéressera pas à la définition de la fonction Min, c'est encore une boite noire qui admet une entrée et renvoie un résultat.

2.1.5 Quelques Conseils en Programmation

Certaines notions données ci-après ne sont pas encore définies, ce paragraphe doit être relu à la fin de ce chapitre.

1. Définir les fonctions à appeler par le main, et mettre le minimum de fonctions et les plus générales.
2. Eviter dans la limite du possible les variables globales (variables connues par toutes les fonctions), et les utiliser uniquement si elles sont communes à la majorité des fonctions, afin d'éviter d'encombrer les arguments des appels.
3. Faire des fonctions indépendantes, ce qui les rend d'utilité générale, et peuvent donc être utilisées par d'autres programmes. Ne pas hésiter à faire autant de fonctions que nécessaire, mais, précises et concises.
4. Faire plusieurs modules dès que le programme devient long (plusieurs centaines de lignes), et mettre dans des fichiers d'inclusion les paramètres communs. Ceci permettra une mise à jour plus facile lorsqu'elle est nécessaire.
5. Faire des commentaires aux endroits importants, et juste là ou c'est nécessaire.

3 Concepts fondamentaux

Un programme écrit en langage C est formé d'un ensemble de fonctions qui s'appellent les unes les autres. Une fonction principale est obligatoire (main). C'est la première fonction exécutée. Les fonctions peuvent être placées n'importe ou dans le programme ou dans différents modules, l'ordre d'exécution est défini par l'ordre des appels.

Un module est un ensemble de fonctions compilées ensembles, mais ne peut être un programme que s'il possède une fonction main. Dans l'exemple précédant, l'ordre d'exécution des fonctions est le suivant :

1. main appelle Lecture ,
2. puis appelle tri
3. qui appelle à son tour (peut être plusieurs fois)
   a) Min,
   b) Ajout
   c) puis Supprime.
4. main appelle enfin Sortie .

Un bloc ou une fonction est délimitée au début par un accolade ouvrante "{" et à la fin par une accolade fermente "}". Une instruction se termine toujours par un point virgule " ;" .

3.1 Identificateur

Toute variable simple, fonction, tableau, structure ou autre doit être identifiée de façon unique par un nom ou identificateur. Cet identificateur est une adresse symbolique que le compilateur utilise pour faire lien entre l'objet défini par le programmeur et son adresse en mémoire.

L'identificateur est formé d'une suite de lettres alphabétiques (y compris le blanc souligné : _) ou de chiffres dont le premier caractère est obligatoirement une lettre.

Identificateurs valides : nom, x, y, l_12, temperature, Bien_faire.

Identificateurs non valides : 4éme, 3commande, taux de change, table+y .

Les commentaires peuvent être mis à n'importe quel endroit du programme il suffit de les insérer entre les symboles /* et */. /* ceci est un commentaire */.

Tous les mots clés du langage sont en minuscule. Le compilateur fait la distinction entre les majuscules et les minuscules.

Les données sont représentées en mémoire selon un type bien précis en fonction de l'utilisation que l'on veut en faire.
– Le type caractère isolé : char sur 1 octet.
– Le type entier simple : int sur 2 ou 4 octets (selon les machines).
– Le type réel simple : float sur 4 octets.
– Le réel double précision : double sur 8 octets.
Ces types peuvent être précisés par une longueur plus grande en utilisant long avec int ou double pour augmenter la taille, par le signe en utilisant unsigned avec int et char pour préciser que l'objet n'a pas de signe (autrement dit toujours positif). Ainsi donc, nous aurons long int (sur 4 octets), long double (sur 10 octets), unsigned char et unsigned int.

On peut aussi utiliser unsigned ou long seul (sans préciser int qui est pris par défaut).

3.1.1 Visibilité et durée de vie d'une variable

Une variable peut être visible (ou connue) dans tout ou une partie du programme. Une variable est globale si elle visible dans tout le module ou elle est définie, elle doit être alors déclarée en dehors de toute fonction.

Une variable est locale si elle est définie dans le bloc (délimité par des accolades { }) ou dans une fonction ; elle est alors visible uniquement là ou elle est définie. Pour utiliser une variable (à l'exception des noms de fonctions) définie dans un autre module, il faut la déclarer extern.

Les noms de fonction sont par défaut extern. Une variable locale (appelée aussi automatique) est temporaire. Sa durée de vie est limitée au temps d'exécution de la fonction ou elle est définie. Elle est réservée à l'entrée du bloc (ou fonction) et libérée à la sortie.

Une variable globale est permanente, c'est à dire qu'elle garde sa valeur pendant toute la durée d'exécution du programme.

Une variable locale perd sa valeur dès qu'on quitte la fonction. On peut cependant lui faire garder sa valeur, qu'on peut récupérer si on entre de nouveau dans la fonction, en utilisant la déclaration static (elle devient alors permanente).

Exercice 7. Ecrire et tester le programme du listing3.1 . Ce programme peut etre simplement compilé tel qu'il est, l'edition de lien necessite le fichier contenant la variable "xx". Nous verrons dans le détail la compilation séparée, à la section 5.1, pour l'instant commenter la ligne “extern int xx”, et celle qui imprime la variable “printf(..... , xx) ;”, et creer l'executable avec la commande :

```
gcc typeVariable.c -o typevar
```

gcc nom du programme compilateur

typeVariable.c nom du programme source

-o output de gcc, c'est a dire l'executable

typevar typevar est le nom de l'executable, mettez ce que vous voulez, -o

La compilation seule se fait avec l'option “-c” , utiliser la commande

```
gcc -c typeVariable.c
```

Listing 3.1: Visibilite de variable

```
1  #include <stdio.h>
2  // entete pour les fonctions printf et scanf
3  double x=12.3; // globale: visible partout
4  float y=22.6;
5  //extern int xx; // definie dans un autre module
6  // ce programme peut etre simplement compile,  l'edition
7  // de lien nÃ©cessite le fichier contenant la variable "xx"
8  void F1( int ); // declaration de la fonction F1,
9  // accepte un entier pour argument
10 // ne retourne rien : "void"
11 #define PI 3.1459 // PI est remplace avant compilation
12
13 int main(){   // Fonction Principale
14   int a=12,b=23,y=11; // cette y cache la y globale
15   char c='A';
16   F1(65);  // F1 est  appelee avec argument 65
17   printf("\na␣=␣%d␣␣y=␣%d␣␣x␣=␣%lf", a,y,x);
18   {
19     // Les declarations restent locales
20     float a=123.5;  // nouvelle a
21     printf("\nDans␣le␣bloc:␣a␣=␣%f␣␣x␣=␣%lf", a,x);
22   }
23   printf("\nHors␣bloc:␣a␣=␣%d␣", a);
24   printf("\n␣valeur␣de␣PI␣=␣%f␣",PI) ;
25   // printf("\n valeur extern  de xx =%d ",xx);
26   printf("\n␣Salam␣\n");
27 }
28 void F1( int K){
```

```
29    char tt= 'R'; // tt connue uniquement ici
30    printf("\nDans F1: tt=%c tt=%d K=%c  K=%d ",tt,tt,K,K);
31  }
```

Exercice 8. Modifier le programme3.1 et l'essayer. Par exemple, Ajouter de nouvelles variables, réelles, entiers et caractères, les initailiser puis les afficher.

3.1.2 Constantes

Une constante est un objet qui ne peut être modifié contrairement aux variables dont les valeurs peuvent changer pendant l'exécution du programme.

Chaîne de caractères C'est une suite de caractères par exemple : « c'est mon premier programme », ou un caractère isolé : 'A', '}', '0'.

Constante entière décimale comme 345, Hexadécimale 0x3A7 précédé par 0X ou 0x ou Octale 0345 précédé par le chiffre 0

Constante réelle : 30.7 ou 3.07E2 qui vaut 3.07×10^2

Constante symbolique On utilise define : #define PI 3.14159 ou encore #define VRAI 1.

#define fait le remplacement de PI partout dans le module à partir de l'endroit ou elle est définie par le nombre 3.14159, avant la compilation du programme. De même pour VRAI.

3.1.3 Tableaux et Pointeurs

Plusieurs variables de même type et de même longueur peuvent être regroupées sous un même nom appelé tableau.

Un pointeur est une variable qui contient l'adresse de l'objet défini et non pas la valeur de celui-ci. Un pointeur se reconnaît par l'étoile ¨*¨ mise devant la variable.

- int A[3] ; A est un vecteur de 3 entiers.
- int B[5][4] ; B est une matrice 5 x 4 entiers , ou tableau à deux entrées.
- int * xx ; La variable xx contient l'adresse d'un entier.

On dit aussi qu'elle pointe sur un entier. Dans int *xx = A ; xx est initialisé à la déclaration par A qui est une adresse, alors que dans *xx = 10 ; xx pointe sur un objet de valeur 10 (utilisation et non déclaration).

*xx signifie "objet pointé par xx " ou encore "objet dont l'adresse est donnée par xx". Ces notions seront reprises en détails plus loin.

3.2 Expression et Instruction

Une expression est une donnée élémentaire : une variable, une constante ou une relation logique.

Une instruction est formée d'une ou de plusieurs expressions qui se terminent par un point virgule.

Exemple 9. Définition de fonction

```
1  fonction1(){          // Définition de la fonction
2    int i, nombre;       // déclaration de variables
3    printf("Donnez un nombre entier: "); // Ecriture
4    scanf("%d",&nombre);
5    for (i=1; i<= nombre; i++)
6        // Boucle sur une instruction
7    printf("\t %d",i); // \t = tabulation
8    fonction2( ); // Appel de la fonction2
9  }
```

Il faut noter dans cet exemple le retrait des instructions qui permet de rendre plus facile la détermination des début et fin de d'instruction, bloc ou de fonctions.

3.2.1 Pre-Compilation

Avant de compiler un programme il est parfois utile de définir certaines constantes symboliques ou d'inclure des déclarations de variables déjà sauvegardées dans d'autres fichiers. On utilise alors des directives de pré-compilation commençant par le symbole #.

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

Le fichier stdio.h contient les déclarations des instructions standards d'entrée/sorties. Il est nécessaire de l'inclure avant la compilation si on procède à des écritures ou lectures.
De même on inclue math.h (ensemble de declarations de fonctions matematiques, ne pas confondre avec la librarie), si des fonctions mathématiques telles que sinus, cosinus sont utilisées.

```
#define PI 3.14 // remplace PI par 3.14 avant la compilation
```

Ces directives sont exécutées avant la compilation.

3.2.2 Opérations

Les différents opérateurs élémentaires que nous aurons à utiliser sont les suivants :

Opérateurs de bases

Arithmétiques Multiplication (produit) / Division (rapport) + addition (somme) - soustraction (différence) % reste de la division (entière)

Relationnels < (inférieur) <= (inférieur ou égal) >= (supérieur) == (comparaison) != (différent)

Logiques && (ET logique) || OU logique) ! (NON)

Logiques binaires & (ET binaire) | (OU binaire) ~ (NON binaire)

Ces opérateurs interviennent sur le bit et non pas sur le mot ou l'octet et ne s'utilisent que sur les variables de type carèctère ou entière. La comparaison de zone se fait bit à bit.

Opérateurs complémentaires

? : Si...alors sinon... : opérateurs trinaires (il a 3 opérandes)

++ incrémentation

– décrementation

Op= Op suivi du signe, ou Op est un opérateur quelconque (+, -, *, /, %, >, <, ...) suivi de =. L'opération Op est d'abord réalisée puis il y a affectation du résultat. Exemple : au lieu de A= A+ b on ecrit A += b.

Dans la déclaration :

```
i=4; indic = (i < 3)? 0 : 100
```

L'expression trinaire " ? :" remplace

```
if (i <3)  indic = 0 else indic = 100
```

Dans notre cas indic=100.
Ces opérateurs sont utilisés pour simplifier les expressions et écrire des instructions plus condensées et plus concises.

```
x += 3 on lit : 3 est ajouté à x, équivaut à x = x + 3
x *= 3 équivaut à x = x * 3
j >>= 2   j est décalée de 2 bits vers la droite
x++   veut dire x = x + 1
```

Tous ces opérateurs sont affectés d'une priorité qui permet ainsi de déterminer l'ordre dans lequel les opérations doivent être exécutés dans une expression. Comme les parenthèses () ont la priorité la plus élevée, on peut toujours changer l'ordre des opérations en introduisant celles-ci.

Consulter la table des priorités des opérateurs donnée en annexe pour comprendre les exemples suivants.

Exemple 10. Calculons la valeur de A dans :

```
int A, B,C, j = 6 ; char d = 'd' ;
j++; B= j/2;  C = 4*(d-3);  A= 2*(1/5 + C%B)
```

- j devient 7, la valeur de B est alors 3, puisque la division j/2 est entière.
- La valeur de C est 4 * (100-3) = 388 puisqu'elle contient le caractère 'd' qui a pour code la valeur 100, (code ASCII de d). On a C % B = 1 (reste de la division de 388 par 3), et comme 1/5 = 0 (division entière).
- La valeur de A se calcule comme suit : à cause de la priorité de la parenthèse, et de l'opérateur % sur +, on évalue d'abord C%B. On aura finalement A = 2 * (0 +1) = 2.

Exemple 11. Déterminer les valeurs de K1 et K2 à partir des déclarations :

```
int i = -2, j = 6, K1, K2; float x = 5.8, y = 4.5;
K1=(i>0) || (j<7);  K2=(x>y) && (x>j) || (j <5)
```

- La comparaison de (i>0) est fausse , le résultat est donc 0.
- La comparaison de (j<7) est vraie, le résultat est donc 1, donc K=0 || 1 = 1 .

Le Ou logique est vrai si une des 2 expressions est vraie. Toute valeur non nulle est considérée comme vraie, et 0 est considéré comme faux.

On trouvera en fin de chapitre suffisax++mment d'exercices. Il est conseillé au lecteur d'en faire plusieurs avant de passer aux chapitres suivants.

Conversion implicite

Lorsque dans une expression interviennent plusieurs variables de types différents, il y a une conversion implicite qui se fait. Les conversions sont faites ainsi : le type de priorité inférieure est converti en type priorité supérieure.

voir le tableau (3.2.1), la priorité est donnée du plus haut vers le plus bas (premiere colonne du tableau puis deuxieme colonne). Su la meme colonne selon le sens GD : de gauche a droite, et DG de droite a gauche. Par exemple, "+" (addition) est prioritaire sur le "&" (ET binaire), ce dernier est prioritaire sur le "&&" (ET logique).

Par exemple a+b+c-d, se fait de gauche à droite : a+b puis + c puis - d alors que a=b=c ; se fait de droite à gauche : b=c puis a=b ;

<table>
<tr><th>opérateur</th><th>sens</th><th>opérateur</th><th>sens</th></tr>
<tr><td>() [] . -> ++ −(post)</td><td>GD</td><td>== !=</td><td>GD</td></tr>
<tr><td>++ − (pré) ~ ! sizeof</td><td>DG</td><td>&</td><td>GD</td></tr>
<tr><td>* & (pointeurs)</td><td>GD</td><td>^</td><td>GD</td></tr>
<tr><td>+ - unaire</td><td>GD</td><td>|</td><td>GD</td></tr>
<tr><td>cast (type)</td><td>DG</td><td>&&</td><td>GD</td></tr>
<tr><td>* / %</td><td>GD</td><td>||</td><td>GD</td></tr>
<tr><td>+ -</td><td>GD</td><td>? :</td><td>DG</td></tr>
<tr><td><< >></td><td>GD</td><td>= *= /= %= += -= >>= <<= &= ^= |=</td><td>DG</td></tr>
<tr><td></td><td></td><td>, (virgule)</td><td>GD</td></tr>
</table>

TABLE 3.2.1: Priorité des Opérateurs (ordre décroissant)

3.2.3 Fonctions Externes

Les plus utilisées et qui figurent dans la bibliothèque standard sont :

Fonctions Mathématiques

`log, sin, cos, asin, acos, exp, tan, atan, pow, abs, fabs`

Ces fonctions admettent en argument un nombre réel double et renvoient aussi un nombre réel double.

Fonctions de caractères

`isprint(K)`

retourne la valeur 1 si K est imprimable, sinon 0. De même les fonctions

```
isdigit(K), isupper(K),toupper(K), tolower(K),
isalnum(K), isalpha(K).
```

Testent (respectivement) si K est un chiffre, une majuscule, une minuscule, alphanumérique ou alphabétique.

Fonctions de chaînes

strlen(CH) retourne la longueur de la chaîne de caractères CH (nombre de caractères)

strcpy(CH1,CH2) copie la chaîne CH2 dans CH1.

strcat(CH1,CH2) fait la concaténation de la chaîne CH2 à la chaîne CH1.

strcmp(CH1,CH2) compare la chaîne CH1 et la chaîne CH2, et renvoie 0 si égalité, un nombre négatif si CH1<CH2, un nombre positif sinon.

Fonctions sur la mémoire

memcpy(D, S, T) copie T octets de la position S a la position D.

memset(D, C, T) initialise la zone D par le caractère C sur T positions.

memove comme memcpy.

Exemple 12. Tester le programme du listing

Listing 3.2: Fonctions sur la mémoire

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <memory.h>
4  #define entier unsigned int
5  #define byte unsigned char
6  #define COL 4
7  #define LIG 3
8  #define TAILLE COL * LIG
9
10 int main (){
11   byte ch[20]="Salam  Alaykoum";
12   byte ptr[20]="Oualaykoum ";
13   byte *ptr_C, i, j;
14   entier vect[]={1, 2, 3, 4, 11, 12, 13, 14,  21, 22, 23, 24 };
15   entier mat[LIG][COL];
16   memcpy(ptr+11, ch,6);
17   memset(ch, 'A', 7);
18
19   printf("\nchaine = %s  ptr= %s\n", ch, ptr);
20   printf("\n\tLe Vecteur \n");
21   for(i=0; i< TAILLE; i++)
22     printf(" %d ",vect[i]);
23   printf("\n\tLe Matrice \n");
24   memmove( mat,vect, TAILLE*sizeof(entier) );
25   for(i=0; i< LIG; i++)
26     for(j=0; j< COL; j++)
27       printf(" %d ",mat[i][j]);
28   putchar('\n');
29 }
```

Un complément d'information peut être trouvé en utilisant le help du langage.

3.2.4 Structures de Contrôle

Un programme est construit à partir d'instructions de branchement et de répétitions. Ci-après sont données les instructions essentielles.

if et switch

```
if ( Condition) { instructions1; }
else {instructions2;}
```
[1]

Si 'Condition' est vraie, la première partie (instructions1) est exécuté par ailleurs le second bloc (instructions2) est exécuté. le reste n'est pas nécessaire actuellement. Toute valeur non nulle est considérée comme vrai, 0 est faux. Ainsi, dans le code suivant si val est non nulle, la valeur est imprimée, sinon nulle est affiche.

```
if (val == 0) printf( "la valeur est null");
else printf( "la valeur est %d ", val );
```

La commande switch est un if multiple, Noter les commentaires sur le "break"

```
1
2  switch (a) {
3    case 1: printf( "votre choix est de 1 \n";
4      break; // nécessaire, sinon cas suivant est exécute
5    case 2: printf( "votre choix est de 2 \n";
6    case 5: printf( "votre choix est de 2 ou 5 \n";
7      break; / / quitter le switch
8    default: printf( "votre choix n'est pas 1, 2 ou 5";
9  }
```

Exercice 13. Remplacer le "switch" par son équivalent en "if".

Les boucles : while, do while et for

```
while (condition) {instructions}
```

Bien que la condition est vraie, le bloc de déclarations sont exécutées, la condition est testée en premier. l'autre boucle (do-while le test est exécuté en dernier.

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

Une présentation plus complète la boucle est pour :

```
for (initialisation; condition; dernière) {
       instructions;}
```

La première partie "init" est exécutée une fois avant le test lors de la "condition" est vrai, les déclarations bloc est exécuté, après que le "dernier" est exécuté, puis la "condition" est testé à nouveau pour continuer. Les 2 boucles suivantes sont équivalentes.

```
for (i=0; i<10; i++)
        printf("\n i = %d",i);
```

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

break et continue

La commande break s'arrête et laisse le passage ou de la boucle interne, alors que se poursuivent, de continuer la boucle, il faut que la prochaine itération.

```
1  for (int i = 0; i <4; i++)
2    for (int j = 0; j <8; j++) {
3       if (j == 2) continue;
4       if (j == 5) break;
5     printf( "\ni =" <<i << "j =" <<j;
6  }
```

1. '{' et '}' sont nécessaires s'il y a plusieurs instructions

Ce code ne fait pas l'affichage pour j = 2, il va à la prochaine j = 3, mais jamais pour j supérieur ou égal à 5 ;

continue permet de sauter les instructions qui viennent après jusqu'à la fin de la boucle. Contrairement au break, il n'y a pas de sortie de la boucle.

goto

C'est l'horrible instruction de la programmation anarchique, à éviter en tous les cas. Se débranchées une étiquette.

```
goto etiq; /* etiq est l'endroit de débranchement */
    .........
etiq : instructions à exécuter.
```

Exemple 14. Le programme suivant montre rapidement l'utilisation de ces instructions. Le compiler, l'exécuter puis apporter les changements suivants :

Listing 3.3: Instructions de Base

```
1  #include <stdio.h>
2  int main() {
3    int i,a,b, N, R, Mask =0x80000000;
4    printf("\nEntrer un Entier\t");  scanf("%d",&N);
5    for(i=0; i < 32; i++) {
6      if(i%8 == 0) putchar(' '); // groupe par 8
7      R = N & Mask; //  '&' binaire
8      if( R == 0) putchar('0');
9      else  putchar( '1');   N <<= 1;  // decalage a gauche
10   }
11    printf("\nDonner une fraction ( 2 entiers) \t ");
12   scanf("%d%d",&a,&b);
13   for (i = a * b; i > 1; i--)
14     if ((a%i == 0) && (b%i == 0))  a /= i ,  b /= i;
15   printf( "\nFraction reduite :\t %d / %d", a ,b);
16   printf( "\nChoisir une valeur \t 1 \t 2 \t 3 \n");
17   scanf("%d",&a);
18   switch (a){
19   case 1: printf("Votre choix est 1 \n");  break;
20   case 2: printf("Votre choix est 2 \n");  break;
21   case 3: printf("Votre choix est 3 \n");      break;
22   default: printf("Votre choix n'est ni 1 ni 2 ni 3 \n");
23   }
24   putchar('\n');   return 0;
25 }
```

3.3 Entrées-Sorties Standards

Les supports d'entrées/sorties standards sont l'écran et le clavier. Les autres entrées/sorties seront examinées dans le chapitre '¨fichiers'¨. Ces fichiers sont connus sous les noms stdin (entrée), stdout (sortie) et stderr (sortie d'erreur).

Les fonctions utilisées pour permettre ces transferts sont :

K=getchar() Lecture d'un caractère isolé du clavier.

```
K=getchar(); //K reçoit le code lu au clavier.
```

putchar(K) Écriture sur écran d'un caractère isolé.

```
char K='r'; putchar(K);// Affiche caractère 'r'
```

gets(Str) Lit une chaîne de caractères à partir du clavier et la met dans Str.

```
char Str[10]; gets(Str);
```

Le tableau Str (ou chaîne de caractères) est rempli à partir du clavier.

puts(Str) Écrit la chaîne de caractères Str sur l'écran.

```
char Str[10]; puts(Str);
```

printf() Fait un affichage formaté sur écran voir le tableau des formats essentiels.. Les types des variables à écrire doivent être précisés en utilisant le symbole % suivi du type. voir tableau 3.3.1 et le listing

```
char K='a'; int i=12; float vv=1.35;
printf("Car: %c, Entier: %d, Réel: %f",K,i,vv);
```

scanf() Lecture formatée [2] : les types des variables à écrire doivent être précisés en utilisant le symbole % suivi du type.

Format	Type	Format	Type
c	caractère	s	chaîne de caractères
i, d	entier décimal	x	entier hexa
e,f,g	réel	lf	double précision
u	non signe	o	entier octal

TABLE 3.3.1: Format Entrées/Sorties

Les valeurs a et b mises comme cela a.b entre le "%" et le type voir le tableau , précise la longueur de la zone a utiliser et la précision. Dans l'instruction suivante la valeur de vv est affichée sur 10 positions dont avec 3 decimales : Voir le listing pour plus de détail.

```
printf ("un réel: %10.3lf",vv);
```

Listing 3.4: Format des Entrées Sorties

```
1  #include <stdio.h>
2  int main(){
3    char C, nom [20];      int compte, N;
4    float x, moy, somme;
5    printf("\nEntrer un entier ");   scanf("%d",&N);
6    printf("\nLe nombre sur 10 positions en Dec ");
7    printf("  puis Non Signe, Octal et Hexa \n\t ");
8    printf("%10d %10u %#10o %#10X\n",N,N,N,N);
9    printf("Donner caractere");
10   C=getchar( );getchar(); //enlever \r
11   printf("son code est :%d",C); getchar();
12   puts ("\nEntrer votre nom ");   gets(nom);
13   // Affiche une  chaine des caracteres
14   printf("Donner Nombre de valeurs ? ");
15   scanf("%d",&N); // lit un  entier decimal
16   printf("Entrer les %d valeurs ?: ",N);
17   for (compte = 1; compte <= N; compte++) {
18     printf("x = ");      scanf("%f", &x);
19     printf("valeur  lue = %14.4f ", x);
20     somme += x;
21   }
22   moy=somme/N;
23   printf("\nLa moyenne est %8.2f\n",moy);
24 }
```

3.4 Fonctions

Une fonction est un ensemble d'instructions regroupées sous un nom. Le type de la fonction correspond à la valeur retournée, il peut être simple (int, float, char, struct, int *) ou complexe (tableau, structure). Les arguments peuvent aussi être simples, complexes ou ne pas exister du tout.

Une fonction doit être connue avant son utilisation. Elle est alors soit :

2. scanf s'arrête au premier blanc pour les chaînes de caractères et au premier nombre non valide

définie Tous les détails de ce que fait la fonction sont explicitement donnés. le corps de la fonction est mis entre "{ }".

```
type nom_fonction (arguments){
      instructions
     return (valeur ou adresse)
}
```

déclarée son type (ce que renvoie la fonction) est connue et sa définition peut alors être donnée n'importe ou dans le programme.

Exemple 15. Dans le code suivant, la première fonction sort juste un message et ne retourne rien, il est donc déclarée de type vide "void". La suivante accepte une chaîne de caractères comme argument, la dernière est plus complète avec des arguments en entrée et des valeurs retournées.

```
void F1 () { printf("Bonjour tout le monde" );}
void F2 (char  *txt) { printf("votre texte est %s ",txt);}
int F3 (int a, int b) {int c = a + b; retour c;}
```

Pour utiliser ces fonctions, il faut les appeler a partir, du "main" par exemple, comme ceci : F1 () pour appeler la fonction sans argument F1, F2("un message"), la chaîne est envoyée à F2 qui imprime cette chaîne.

La fonction retourne une valeur F3. int y ; y = F3 (5, 12), y reçoit la valeur de retour. Bien sûr, on peut utiliser des variables comme arguments si elles sont renseignées.

Avant d'utiliser une fonction il faut d'abord la déclarer ou la définir. La déclarer signifie donner uniquement son entête, comme par exemple **int F3(int, int) ;** La définir signifie donner aussi le code du corps de la fonction.

Les fichiers d'en-tête, comme par exemple math.h, qu'on inclue, si on utilise les fonctions mathématiques, contiennent justement les entêtes de ces fonctions. Ces entêtes sont dans le répertoire /**usr**/**include**. Ne pas les confondre avec la bibliothèques des objets (code compile) des fonctions qu'on trouve dans /**usr**/**lib**.

3.4.1 Fonction Récursive

Enfin, une fonction récursive est une fonction qui s'auto appelle. C'est le cas de la fonction factorielle. La sortie "coût" pour montrer que les adresses de variables sont différentes. Faire un essai avec par exemple n=4.

```
1 int fact (int n) {
2    int b;
3    printf("\n adresse  de b=%x adresse de  n ", &b,&n);
4    if (n == 1) return 1;
5    b = n * fact (n-1);
6    return b;
7 }
```

3.5 Les Tableaux

Un tableau est un ensemble de même type de données. Alors **int AA [10**], déclare un tableau de 10 nombres, il est équivalent à réserver 10 variables. ces variables sont à savoir AA [0], AA [1], ..., AA [9].

Un tableau peut être initialisé à la déclaration à temps, comme **double d [] = {12.5, 13, 25.6, 4.5} ;** La taille n'est pas nécessaire, étant donné, ici, il est défini par le nombre de valeurs déclarées (ici 4). Pour attribuer une valeur, d [2] = 15.9, nous précise l'indice, ici 2. Rappel de l'indice du premier élément est 0.

Le contenu de la d est maintenant (12,5, 13, 15,9, 4,5).

L'indice n'est pas nécessairement positif, il peut etre de l'intervalle. Les affectations d [-5] = 23.6 ou d [34] = 55, sont correctes pour le compilateur.

Attention : mais elle peuvent conduire à d'étranges résultats.

Un tableau peut avoir plusieurs dimensions, ainsi float M [2] [5] est un tableau à 2 dimensions avec 2 x 5 variables, chacun est un nombre réel. Pour initialiser cette série, nous utilisons des accolades comme ceci :

```
float M [2] [5] = { { 1.4, 2.4, 3.3, 2.1, -4 },
                    { 112, -2.1,  7, -3, 0.5 }
                  };
```

Exemple 16. L'argument de la fonction peut être un tableau, ici, la fonction retourne la somme des éléments d'un vecteur reçu comme argument. N'oubliez pas le symbole "[" dans la déclaration, mais a l'appel seul le nom du tableau est donne :

Listing 3.5: tableau en argument

```
1  #include <stdio.h>
2  int LectVect (int V []){
3    int N, i;
4    puts("Entrer nombre de valeurs:  "); scanf("%d",&N);
5    printf ("Entrer les %d valeurs  ",N);
6    for(i=0; i<N; i++) scanf("%d",&V[i]);
7    return N;
8  }
9  int PrintVect (int nb, int v []) {
10    int i,p = 0;
11    for (i = 0; i <nb; i++)  p += v [i];
12    return p;
13 }
14 int main () {
15   int A [20];int nbVal;
16   nbVal=LectVect(A); //remplir le vecteur
17   int y=PrintVect(nbVal, A);//somme des valeurs
18   printf("\nLa somme est %d\n",y );
19   return 0;
20 }
```

3.6 Exercices

1. Que donne le code suivant :

```
1 int i = 8, j=5, k=0; float x=0.005, y=-0.01;
2 char c='c', d='4'; int mm=c, nn=d, exp1, exp2, exp3;
3 printf("c= %c  mm= %d  d= %c  nn= %d", c,mm,d,nn);
4 exp1 =(3 *i*j-2)% (2*d); exp2= 5*(i+j)>c == k;
5 exp3 = x> y && i> 0 || j <5;
6 printf("(3 *i*j-2)% (2*d) = %d: ",exp1) ;
7 printf("(5 * (i + j)> c) == k  : %d",exp2 );
8 printf( "(x> y) && i> 0 || j <5: %d",exp3);
```

2. Détailler l'instruction :

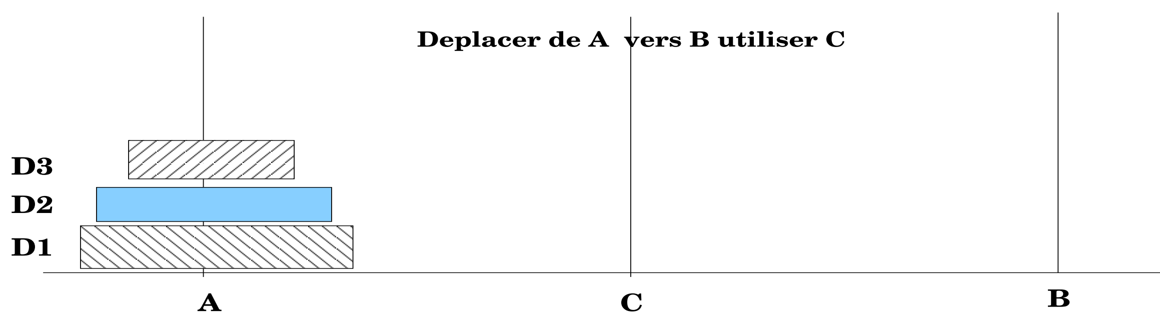
```
if (abs(x) < xmin) x=(x>0)? xmin : -xmin;
```

3. Quelles sont les sorties des procédures suivants :

```
1 int i=j=x=0; int k;
2 do {  if (i % 5 == 0) {
3          x++; printf(%d ,x);
4       }
5       i++;
6    } while(i<20);
7 printf ("\nx=%d",x);
8 for(i=1,x=0;i<10;i++)  {
```

```
 9      if(i%2==1) x += i; else x--;
10      if (i==5 || i==7) continue;
11      if (x >9) break;
12      printf("%d",x);
13   }
14   printf("x=%d",x);
```

4. Écrire une fonction qui donne les multiples de 3 dans un intervalle [n, m], m et n sont lus à partir du clavier.
5. Écrire un programme qui saisit et stock dans un tableau une liste de 20 caractères , puis les affiche sur écran. Utiliser getchar, putchar puis scanf et printf. Reprendre la question avec un nombre quelconque de caractères.
6. Lire une liste de nombres, les stocker dans un vecteur de taille n x m, (n et m sont lus du clavier), imprimer les nombres sous forme de matrice (avec n lignes et m colonnes).
7. Reprendre l'exercice précédant en utilisant différentes boucles : for, while et do-while.
8. Calculer factorielle d'un nombre donné, de 2 manières : en tant que fonction récursive puis itérative.
9. Écrire la fonction de la suite Fibonacci (1, 1, 2, 3, 5, 8, 13, 21.....) et faire une procédure de test.
10. Programmer le fameux jeu “Tour de Hanoi” en utilisant la fonction récursive. Dans ce jeu il faut déplacer une tour de N disques de tailles différentes de la position initiale à la position finale, on utilise un position de transit. En doit respecter les 2 conditions : Un grand disque ne peut être pose sur plus petit et déplacement d'un disque à la fois. Voir figure



11. Écrire la fonction puissance entière d'un nombre x est un réel et n un entier : $y = x^n$. Faire une procédure de test pour la fonction.
12. Quel est le plus grand nombre entier (32 bits), soit maxInt. si on doit faire N1+N2 avec N1 ou N2 >maxInt, peut on utiliser le float ou le double pour faire cette opération ?

4 Concepts Avancés

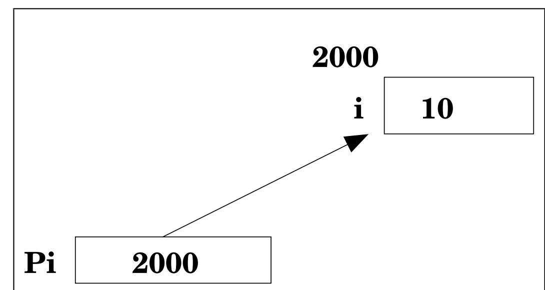
4.1 Pointeurs

4.1.1 Définition

Une variable pointeur est une zone qui contient l'adresse d'un objet (d'une variable qui contient une valeur). La notion de pointeur est très importante car elle permet de travailler sur des objets non encore connus lors de l'écriture du programme.
Les pointeurs permettent d'utiliser des fonctions générales non connues à la compilation, des tableaux dynamiques ou des structures complexes (voir section suivante), dont la réservation se fait dynamiquement lors de l'exécution.

```
int i = 10 ;  // i contient la valeur 10.
int *pi ;   // pi contient l'adresse d'un entier
pi = &i;   // pi contient l'adresse de i
       // ou en une seule instruction comme ceci
int *pi = &i;
```

Maintenant ***pi*** pointe sur la zone ***i***, et donc ****pi*** (lire : objet pointe par ***pi*** ou l'objet dont l'adresse est ***i***) donne la valeur de *i* c'est a dire 10. Supposant l'adresse de i est 2000, nous aurons le schéma de mémoire suivant.



A l'initialisation nous aurions pu écrire **int *pi=&i;** au lieu et place des deux instructions

Exemple 17. Soit le programme suivant :

Listing 4.1: Utilisation de pointeurs

```
int main(){
  char *CH[]={"Programme␣","sur␣les␣","pointeurs"};
  int A[]={1,3,5,7,9,10,-2,-4,11,26  };
  int i, *p=A; //int i;  int p;  p=A;
  printf("\nA␣␣␣␣␣␣␣contient␣adr␣du␣tableau␣(A)\t␣:␣%#X␣", A);
  printf("\np␣aussi␣contient␣adr␣du␣tableau␣(A)\t␣:␣%#X␣", p);
  printf("\n\tadresse␣du␣tableau␣CH␣␣\t␣␣␣:␣%x", CH);
  printf("\n\t\tvaleur␣A[1]␣␣\t␣:␣%d" ,  A[1]);
  printf("\nadresse␣A+9␣:␣%#x␣␣&A[9]␣=␣%#x", A+9, &A[9]);
  printf("\nadresse␣du␣pointeur␣des␣chaines:␣␣CH␣␣␣:␣%#x", CH);
  printf("\nadresse␣du␣pointeur␣des␣chaines:␣␣CH+2␣:␣%#x", CH+2);
  printf("\nChaine␣:␣*(CH+1)␣=␣%s␣", *(CH+1) ) ;
  // se rappeler  "CH" a besoin de 2 '*' pour atteindre un caractere
  for(i=0; i < strlen(CH[2]); i++)
    printf("\n\tCH[2][%d]␣=␣%c␣␣*(*(CH+2)+i)=␣%c",i,  CH[2][i], *(*(CH+2)+i)) ;
  putchar('\n');return 0;
```

Exercice 18. Tester le programme précédant. Modifier le types des variables, l'ordre de leur déclaration et la taille des tableaux.

4.1.2 Pointeur comme Argument de Fonction

Pour récupérer l'information à travers les arguments de la fonction, il faut faire un passage par adresse et non par valeur. En effet, une variable locale dans une fonction est crée à chaque entrée dans la fonction, et libérée à chaque sortie. Par conséquent les changements sur la variable dans la fonction ne sont pas effectifs sur les variables de la fonction appelant.

En d'autres termes, si on veut récupérer la valeur d'une variable à travers l'argument de la fonction, il faut passer l'adresse de la variable comme argument. Une autre facon est évidemment de la récupérer par retour de la fonction, comme déjà vu. Il faut alors donner le pointeur sur la zone.

Exemple 19. Utilisation de pointeur. Dans la fonction swap2 , la valeur pointée par "a" est remplacée par la valeur pointée par "b".

Listing 4.2: echange de valeur dans fonction

```
1  #include <stdio.h>
2  void  swap1( int a, int b){
3    int tmp=a;     a=b;   b=tmp;
4    printf("\n\t␣adr␣de␣a␣=%#X␣b=␣%#X", &a,&b);
5  }
6  void  swap2( int *a, int *b){
7    int tmp=*a;    *a = *b;   *b=tmp;
8    printf("\n\t␣␣a␣=%#X␣b=␣%#X", a,b);
9
10 }
11 int main(){
12   int x, y;
13   printf("\n\tadr␣de␣x␣=%#X␣y=␣%#X", &x,&y);
14   printf("\nDonner␣2␣entiers:␣");
15   scanf("%d%d",&x,&y);
16   swap1(x,y);
17   printf("\nApres␣swap1,␣␣x=%d,␣y=%d",x,y);
18   swap2(&x,&y);
19   printf("\nApres␣swap2,␣␣x=%d,␣y=%d␣\n",x,y);
20   return 0;
21 }
```

4.2 Tableaux et Pointeurs

Un tableau à une dimension peut être utilisé comme pointeur. Il suffit alors de réserver un pointeur de même type que le tableau et de l'initialiser par l'adresse du début du tableau. Au fait le nom du tableau est considéré comme pointeur. Étudions l'exemple suivant :

```
int main( ) {
  int i, x[6] ={10, 11, 12, 13, 14, 15};
  for (i = 0; i < 3; i++)
     printf("\nx[%d]= %d *(x+%d)=%d", i,x[i],i,*(x+i) );
}
```

Les sorties de ce programme sont :

```
i= 0, x[i]= 10, *(x +i)= 10
i= 1, x[i]= 11, *(x +i)= 11
i= 2, x[i]= 12, *(x +i)= 12
```

x : est l'adresse du tableau ou de l'élément **x[0]** du tableau. **x + i** : on ajoute **i** à l'adresse x, c'est à dire donner l'adresse du ième élément du tableau x. *(x+i) : donne le contenu de l'objet pointé par **x + i,** c'est à dire **x[i]**.

Exercice 20. Considérons la déclaration, :

```
double M[3][4] ={ {-1, 12, 13, 14}, { 15,16,17,18},
                  {19,20,21,22}    };
double *PM; PM=M;
```

On suppose que l'adresse de M soit 1000. Donner les adresses des nombres -1, 15 et 17. A quoi correspond PM+7, *(PM+7), *PM+7. Donner PM+k en fonction des indices i et j de M.

4.2.1 Opérations sur les Pointeurs

Les opérations d'addition, de soustraction et de comparaison sont possibles sur des pointeurs de même type. L'existence et la validité des objets pointés restent sous la responsabilité du programmeur.
Regardons les instructions suivantes :

```
int *px , i, Tab[10] = { 2, 4, 6, 8 , 10, 12, 14, 16};
px = &Tab[3]; // px pointe sur l'élément 3 du vecteur Tab
py = &Tab[1]; // py pointe sur l'élément 1 du vecteur Tab
px++ ;        // px pointe sur l'élément suivant,  Tab[4]
py--;          // py pointe sur l'élément précédant, Tab[0]
```

L'instruction if (py > px) {......} est légale car py et px pointent sur objets de même type. Les priorités doivent être respectées, comme dans toute expression du langage, il faut faire attention et consulter les tables de priorités.
Dans la première[1], i reçoit l'objet pointé par px+2, dans la deuxième i reçoit l'objet pointé par px, il sera ensuite augmenté de 2, car le symbole * est prioritaire sur le symbole +.

Exercice 21. On suppose que les tableaux sont adjacents et que l'adresse de C est 12000. Donner les adresses des éléments D, E, C[2], D[4], D[6], E[3] et E[-1]. Utiliser les pointeurs pour donner la valeur de T[4] puis T[i] en général.

```
char C[8]= {'a','b','c','d','e','f','g','h'};
int  D[5]= {12,13,14,15,16};
double  E[]={1.5, 2, 2.5, 3};
double T[40];
```

4.2.2 Allocation Dynamique

Pour bien utiliser les pointeurs il faut bien comprendre la notion de "allocation dynamique de la mémoire ".

L'allocation dynamique combinée nécessairement aux pointeurs est une autre caractéristique intéressante du Langage C. Au lieu de faire la réservation statique (à l'écriture du programme : taille figée à la compilation de la zone mémoire, on peut réserver la mémoire de façon dynamique (pendant le déroulement du programme à l'exécution). Cette façon a plusieurs avantages, entre autres :

1. La réservation des espaces ne sont pas obligatoirement contigus
2. La réservation et la libération de mémoire se fait à la demande
3. On peut libérer la mémoire pour l'utiliser pour une autre procédure. Par exemple si on a réserver une tableau pour la résolution d'un système d'équations, on peut libérer cet espace dés l'obtention de la solution pour utiliser cet espace pour stocker une image.
4. La programmation est plus générale, on n'a pas besoin de surdimensionner les tableaux, ni retoucher à chaque nouvelle application le source du programme.

```
float *PP; PP = (float *) malloc (10 * sizeof(float));
```

L'instruction précédante fait la réservation de 10 zones pour de nombres réels. L'adresse de la zone est mise dans la variable PP.

1. L'expression i = *(px+2) ; est légale, mais différente de i = *px+2 ;

4.2.3 Pointeur sur Pointeurs

Tout comme un tableau à une dimension (un vecteur), un tableau à plusieurs dimensions peut être réservé dynamiquement, c'est à dire sans connaître sa taille à la compilation du programme. Son utilisation peut se faire soit par les indices soit par l'indexation (à travers des pointeurs de pointeurs).

Étudions le cas de tableaux à 2 dimensions, le lecteur pourra facilement généraliser à plus de 2 dimensions. Nous avons : float Tab[12][23] ; réserve une zone mémoire de 12 x 23 cases de type float, la taille de la zone doit être connue lors de l'écriture du programme. Cette réservation est statique, c'est à dire fixée par le programmeur et l'utilisateur ne peut la changer.

Nous allons maintenant faire une réservation dynamique d'un tableau, le programmeur ne fixe pas la taille de la zone, c'est l'utilisateur qui décide de la taille de cette zone, et cette taille peut être changée à chaque fois qu'il le désire.

Exemple 22. Allocation d'un tableau à 2 dimensions, les résultats d'exécution sont données dans le programme :

Listing 4.3: allocation dynamique

```
1  main ( ) {
2    float **pp; int i,j,n,m;
3    puts ("\nEntrer␣nombre␣de␣lignes␣et␣de␣colonnes:␣");
4    scanf("%d%d",&n,&m );
5    pp = (float **) malloc (n * sizeof (float *));
6    puts("\t\tAffichage␣Tableau");
7    for (i = 0; i < n; i++)
8      pp[i] = (float*) malloc (m*sizeof(float));
9    for(i=0; i<n; i++)
10     for(j = 0; j<m; j++) pp[i][j] = i*j;
11   for (i = 0; i < n; i++) {
12     putchar('\n');
13     for (j=0; j<m;j++) printf("%10.1f", pp[i][j] );
14   }
15   puts("\n\n\t\tAffichage␣Adresse");
16   for (i = 0; i < n; i++) {
17     putchar('\n');
18     printf("pp[%d]=␣%#x\n", i,(unsigned) &pp[i] );
19     for(j=0;j<m;j++) printf("␣%#x", (unsigned) &pp[i][j] );
20   }
```

Le cast (float *) est nécessaire car malloc retourne un pointeur sur caractère, alors que pp[i] est un pointeur sur float.

L'instruction malloc (Taille), réserve en mémoire dynamique Taille octets. Il est recommandé d'utiliser l'instruction sizeof qui donne la taille en octets de son argument lors de l'exécution du programme. En effet la taille d'un objet peut changer d'une machine à une autre et d'un modèle de compilation à un autre.

pp reçoit l'adresse d'une zone de N éléments, chacun étant un pointeur sur un float. Chaque élément pp[i] reçoit l'adresse de la zone réservée de M éléments du type float. Chacun des **pp[i]** peut être donc interprété comme un vecteur de float, ou encore comme une ligne du tableau pp.
Ces instructions sont équivalentes à : **int pp[N] [M] ;** ou N et M des constantes fixées à la compilation. On peut donc utiliser soit les indices pour accéder à un élément du tableau, soit l'indexation par les pointeurs.

`pp[3][2] = 10; est identique à *(*(pp+3)(+2) = 10; pp[i][j] équivaut à *(*(pp+i)+j)`

Au fait pp contient l'adresse du début en tableau qui coincide avec l'élément pp[0][0] et pp[0] ; pp+1 contient l'adresse de pp[1].

* (pp+i) a même valeur que pp[i] lequel contient l'adresse de pp[i][0] et *(pp+i)+j contient l'adresse de pp[i][0] de même *(*(pp+i)+j) a la même valeur que pp[i][j] .

Lors de la réservation statique d'un tableau à 2 dimensions toutes les zones sont contigues, ce qui n'est pas forcément le cas lors de la réservation dynamique. La zone pp[1] ne suit pas forcément la zone pp[0]. Lors de la réservation dynamique de pp[0] par exemple p[0][2] suit l'élément p[0][1].

Attention à l'ordre des opérateurs ! *pp+i est différent de *(pp+i) le premier signifie la valeur i est ajoutée au contenu de pp, alors que pour le deuxième, la valeur de i (cette valeur est évidemment traduite en adresse selon le type d'objet pointé) est ajoutée à l'adresse pp , puis on prend l'objet pointé par cette nouvelle adresse.

4.2.4 Fonction Argument de Fonction

Une utilisation importante et efficace des pointeurs est l'envoi de pointeur de fonction comme argument. Cela permet d'exécuter des fonctions non connues à la compilation, et de ce fait construire des fonctions très générales. Il suffit pour cela de renseigner l'adresse de la fonction que l'on veut utiliser lors de l'exécution.

Tout comme le nom d'un tableau (sans les crochets [ ]) est interprété comme adresse de tableau, le nom de fonction (sans les parenthèses ( )) est interprété comme adresse de fonctions.

Exemple 23. Lors de l'écriture de la fonction Evaluer, la fonction argument à intégrer n'est pas connue. Cette fonction peut être enregistré dans une librairie. La fonction argument doit être seulement connue a l'utilisation.

Listing 4.4: Pointeur sur Fonction

```
1  #include <stdio.h>
2  #include <math.h>
3  double Evaluer(double x, double Fonc(double) ){
4    // on peut aussi declarer  double (*Fonc)(double)
5    double y;
6    printf("\nDans␣Evaluer:␣&y=%#X␣␣Fonc␣=%#X", &y, Fonc );
7    y=  Fonc (x);
8    return y;
9  }
10 double carre (double x){
11   return x*x;
12 }
13 int main(){
14   double y, x=1.3;
15   printf("\nDans␣main:␣␣&x=%#X␣␣&y=%#X", &x,&y );
16   y= Evaluer( x ,sin  );
17   printf("\nadresse␣sin=%#X␣\tsin(%lf)␣=␣%4.2lf",sin, x,y );
18   y= Evaluer( x ,cos  );
19
20   printf("\nadresse␣cos=%#X␣\tcos(%lf)␣=␣%4.2lf",cos, x,y );
21   y= Evaluer( x ,carre  );
22   printf("\nadresse␣carre=%#X␣\tcarre(%lf)␣=␣%4.2lf\n",carre,  x,y );
23  }
```

Les fonctions carre, sin et cos doivent être du même type (même type du return et avoir les mêmes types d'arguments que ceux déclarés dans l'argument de traiter.

Noter aussi que dans la fonction traiter l'appel à ces fonctions (carre, sin ou cos) se fait par l'utilisation du pointeur pf avec l'appel y= Fonc (x) ; ou y = (*Fonc) (x) ; les deux ecritures sont accepter car Fonc est consideré comme pointeur comme le nom de tableau.

4.2.5 Arguments de main

comme toute autre fonction, la fonction main peut avoir des arguments et retourne un entier, par convention la valeur 0 est retournée quand le programme se termine correctement, autre valeur qui peut indique un type d'erreur. Cette valeur est utile si le programme est appelé par un autre.

```
int main(int nb, char *arg[]){
        ......
   if (entree == NULL) exit(1);
        .......
   return 0;
}
```

L'instruction return 0 est exécutée s'il n y a pas d'erreur d'exécution, si la variable entree est NULL alors 1 est retournée. nb contient le nombre d'arguments[2] a l'appel , arg est un tableau de chaîne de caractères qui contient la liste des arguments.

Le programme argMain.c est compilé avec : gcc argMain.c -o argMain. regarder les valeurs retourne par le programme, voir le contenu de la variable shell "$?", taper dans le terminal : echo $? .

4.3 Les Structures

Une structure est un ensemble de données regroupées<stdio.h> sous un même nom. Ces éléments peuvent être des variables simples de type entier, réel, ou caractère, des tableaux, ou encore une autre structure. Un élément de la structure est appelé champ ou membre et est repéré par son nom (ou identificateur). Un tableau est une structure particulière, tous ses éléments sont de même type.
Une structure est définie en utilisant le mot struct.

```
typedef struct  {
  char  nom[20];
  float note;
} matiere;
matiere m1, m2;
```

On peut ensuite déclarer des variables du type de la structure prédéfinie. Dans l'exemple suivant, **m1** et **m2** sont des variables du type structure matiere.

Une fois la structure définie, elle devient un type (comme float, int, etc.) utilisable pour une variable. Les champs sont manipulés comme des variables. Leur identification se fait en utilisant l'opérateur point.

Pour la structure date donnée ci-dessus, **m1.nomMat="physique"** et **m1.note=13.5 ;**

La déclaration et l'initialisation des structures se font comme pour les variables ordinaires.

typedef

une facile utile de définir les structure es d'utiliser typedef, comme dans

```
typedef struct { int jj; int mm; int aa; } date;
```

le mot "date" équivaut a la structure qui la précède, il n'est plus nécessaire d'ajouter le mot clé "struct", on écrit simplement date maDate ; pour déclarer la variable maDate de type date, ainsi

```
date maDate;  maDate.mm = 12;
```

enum

L'instruction enum permet de donner des noms à des valeurs pour faciliter l'utilisation, comme dans l'exemple le mode Mode prend les valeur 0,1,2 représente par Car, Entier et Reel. par défaut le debut et 0, le suivant est 1 , etc.... Mais on peut choisir d'autres valeur : lundi prend valeur 2, mardi 3 et vendredi 8 . On peut utiliser a la place #define. Les commandes suivantes sont equivalentes.

```
enum jourSemaine {
   lundi=2, mardi, vendredi=8
};
#define lundi 2
#define mardi 3
#define vendredi 8
```

2. Le nombre du programme est le premier argument

4.3.1 Exemples de structures

Un tableau dans une structure

De même qu'on peut imbriquer des structures, et combiner des tableaux avec des structures complexes. Les structures peuvent aussi s'imbriquer, **matiere** doit être connue avant son utilisation comme par exemple :

```
1  typedef struct  {
2      int matricule;
3      char nom [30];      char *adresse;
4      struct matiere module[2]; //tableau de structure
5      float moyenne;
6  } Etudiant;
7  Etudiant etud1;
8  Etudiant  aval4[25]; //tableau de 25 Etudiant
```

Exercice 24. Considérer la structure précédante Etudiant, écrire les commandes qui remplissent les informations de l'etudiant "Benali", de matricule 434 et calcule la moyenne. Les notes sont 12.6 en maths, 14.2 en physique.

Problème 25. TP2 : Écrire un programme qui lit les informations pour chaque etudiant (voir la structure Etudiant), calcule sa moyenne et affiche ses informations y compris la moyenne.

Reprendre la question de l'exercice mais en lisant les informations de tous les étudiants de les stocker puis ensuite de les donner avec la moyenne. ind : Utiliser les tableaux

Pointeur sur Structure

Comme tout autre objet de C, on peut avoir un pointeur sur structure, qui indique l'adresse de a structure.

Exemple 26. On peut envoyer une adresse a une fonction :

Listing 4.5: structure en argument

```
1  #include <stdio.h>
2  typedef struct {
3    int a; float b;
4  } arg;
5  void fonct1(arg *s){
6    s->a = 10; s->b=16.6;
7    printf("\nDans␣fonct1:\t␣s->a=␣%d␣␣␣\t␣s->b=␣%.3f",s->a,s->b);
8    printf("\nAdresse␣de␣␣\t␣s->a=␣%#X,␣\t␣s->b=␣%#X",&s->a,&s->b);
9  }
10 void fonct2(arg s){
11   s.a = 17; s.b=22.6;
12   printf("\nDans␣fonct2:\t␣s.a=␣%d␣␣␣␣\t␣s.b=␣%.3f",s.a,s.b);
13   printf("\nAdresse␣de␣␣\t␣s.a=␣%#X,␣␣\t␣s.b=␣%#X",&s.a,&s.b);
14 }
15 int main(){
16   arg X, *FX;
17   X.a=1; X.b=2.5;
18   printf("\nAvant␣fonct1:\t␣X.a=␣%d␣␣␣\t␣X.b=␣%.3f␣",X.a,X.b);
19   printf("\nAdresse␣de␣␣␣\t␣X.a=␣%#X,␣\t␣X.b=␣%#X",&X.a,&X.b);
20   FX=&X;
21   fonct1(FX);
22   printf("\nApres␣fonct1:\t␣X.a=␣%d␣␣␣\t␣X.b=␣%.3f\n",X.a,X.b);
23   fonct2(X);
24   printf("\nApres␣fonct2:\t␣X.a=␣%d␣␣␣\t␣X.b=␣%.3f\n",X.a,X.b);
25 }
```

Modifier l'exemple précédant, en ajoutant fonct2, qui recoit la structure X et non un pointeur sur X. Vérifier si la fonction fonc2 fait les mises à jour de X. Sortir aussi les adresses des variables (X, s, ...)

Création d'une liste chainée

On peut créer des listes chainées en utilisant les pointeurs de structure. On utilise comme champ d'une structure X un pointeur sur cette même structure X. On écrit :

```
struct el{
    char nom[20]; char prenom [20]; int age;
    struct element *suivant;
};
typedef struct el element;
```

Chaque élément de la liste possède des membres et des pointeurs sur d'autres structures. On peut avoir des listes simples ou doubles. Ici suivant contienne l'adresse d'une autre structure de type **"element"**.

On arrête la liste en mettant le dernier "pointeur" à NULL.

Exemple 27. Le programme suivant permet de créer une liste chainée simple. La variable deb contient l'adresse de S1, qui contient la chaîne de caractères "rouge" et la zone next pointe (contient l'adresse) sur la structure S2, qui elle a son tour renvoie sur la chaine S3. voir le programme et la figure.

Listing 4.6: Liste chainee

```
1  #include <stdio.h>
2  struct sstr  {
3    char coul[12];    struct sstr *next;
4  } ;
5  typedef struct sstr str;
6  int main(){
7    str S1={"rouge",NULL},S2={"vert",NULL},S3={"bleu",NULL};
8     str *deb = &S1;
9    S1.next = &S2;  S2.next = &S3;
10   printf("\ndebut+1 = %X \tdeb->coul =%s \tdeb->next= %X",
11          deb, deb->coul, deb->next);
12   printf("\n&S1   = %X \tS1.coul   =%s \tS1.next  = %X",
13          &S1,S1.coul, S1.next);
14   printf("\n&S2   = %X \tS2.coul   =%s \tS2.next  = %X",
15          &S2, S2.coul, S2.next);
16   printf("\n&S3   = %X \tS3.coul   =%s \tS3.next  = %X\n",
17          &S3, S3.coul, S3.next);
18 }
19 /*------------- Resultat -------------
20 debut  = BF9016BC        deb->coul =rouge        deb->next= BF9016AC
21 &S1    = BF9016BC        S1.coul   =rouge        S1.next  = BF9016AC
22 &S2    = BF9016AC        S2.coul   =vert         S2.next  = BF90169C
23 &S3    = BF90169C        S3.coul   =bleu         S3.next  = 0
24 */
```

Les valeurs hexadécimales BF83F478 et BF83F488 sont les adresses respectivement des structures **S1** et **S2.** les champs ptr contiennent les adresses des structures pointées.

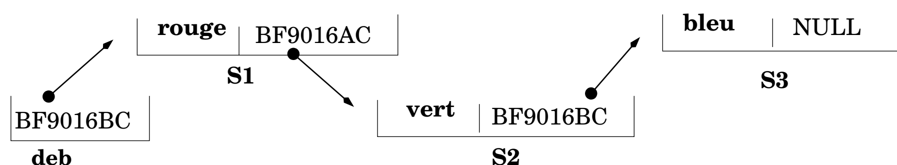


FIGURE 4.3.1: liste de pointeurs

4.3.2 Les Unions

Une union est une structure particulière pour laquelle une même zone peut avoir plusieurs types. On utilise l'opérateur point pour se référer aux membres de l'union.

```
union exposant {
      float x; int a;
};
union exposant y;
```

La variable y de type exposant peut être utilisée comme float ou comme int. Sa taille est de quatre octets (égale à la plus grande taille de ses membres qui est dans ce cas x).

4.3.3 Programmation bas niveau

Une des particularités du langage C, est sa possibilité de travailler directement sur les bits, ce qui peut se faire avec le langage assembleur mais pas avec des langages évolués.

Pour réaliser ce traitement, on utilise les opérateurs logiques binaires et les opérateurs de décalage .

Exemple 28. Le programme suivant donne une utilisation intéressante de l'union et des opérateurs de décalage :

```
#include <stdio.h>
typedef union {
  int  Int;    unsigned char  Ch[4];
} IntUn;
void AfficherBits(int Nombre){
  unsigned int  Resultat, i, Mask =0x80000000;
  for( i=0; i <32; i++){ //
    if ( i%8  == 0 ) putchar(' ');
    Resultat = Mask & Nombre; // Selectionne du premier bit
    if( Resultat == Mask)  putchar('1');
    else putchar('0');
    Nombre <<= 1; // Decaler pour recuperer le bit suivant
  }
}
int  main(){
  int i, val;  IntUn Affiche;
  puts("Entrer code de 4 caracteres :");
  for(i=0; i<4;i++) {
    scanf("%d", &val);    Affiche.Ch[i]=val;
  }
  // caracteres sont stockes de droite a gauche
  // de poids  faible vers poids fort
  puts("\tLe nombre entier correspondant");
  AfficherBits(Affiche.Int);
  printf("\nNombre : %d \t en Hexa : %#x\n",Affiche.Int ,Affiche.Int);
  puts("Entrer 4 caracteres(pas code) :");
    getchar();
  scanf("%c%c%c%c",&Affiche.Ch[0],&Affiche.Ch[1],&Affiche.Ch[2],&Affiche.Ch[3]);
 puts("\tLe nombre entier correspondant");
  AfficherBits(Affiche.Int);
  printf("\nNombre : %d \t en Hexa : %#x\n",Affiche.Int ,Affiche.Int);

}
```

4.4 Traitement de Fichiers

4.4.1 Notion de fichier

Un fichier est un ensemble d'enregistrements, chaque enregistrement étant un groupe d'information, ou ensemble de valeurs, qu'on transmet de la mémoire centrale au disque dur pour les sauvegarder. Les enregistrement sont en général de taille fixe. Un fichier est sauvegardé sur un support permanent (disque, disquette, flash ...).

Un fichier possède un nom externe (identificateur) qui lui est attribué par l'utilisateur. Le gestionnaire de fichier donne un nom interne au fichier, autrement dit, toutes les informations liées à ce fichier : nom, taille, adresse, buffer...

Les supports standards des fichiers ont des noms prédéfinis : stdout pour l'écran, stdin pour le clavier et stderr pour message d'erreur sur écran.

Les fichiers standards sont ouverts et fermés automatiquement. Les autres, doivent être ouverts avant d'être utilisés, et fermés en fin d'utilisation.

4.4.2 Différents types de fichiers

On distingue principalement deux types de fichiers : fichier texte et fichier binaire. Pour ces derniers , l'information contenue dans la mémoire (sous forme binaire, bien sûr) est transcrite telle qu'elle dans le fichier. Il n' y a aucune transformation de l'information lors du transfert. L'information contenue dans le fichier est en quelque sorte une image de celle de la mémoire.

Pour les fichiers texte, l'information est interprétée. On utilise à cet effet les formats. Par exemple, supposons que la valeur à écrire est i égale à 00110000. Que représente-t-elle ? Tout dépend du format utilisé. Ainsi avec printf("%d", i) on aura dans le fichier la valeur 48. par contre avec printf("%s",i) on aura dans le fichier 0 (caractère de code ASCII 48). La même information est donc interprétée différemment.

4.4.3 Entrées/Sorties dans un fichier

Le nom interne d'un fichier est du type FILE qui est un modèle de structure définie dans stdio.h. On déclare le fichier par l'ordre : FILE *ptr_fich ; ptr_fich est un pointeur sur la structure FILE .

Ouverture d'un fichier

L'ouverture d'un fichier se fait par fopen qui permet de donner un nom externe au fichier, et de spécifier le mode d'ouverture (lecture, écriture etc.) : **ptr_fich = fopen (nom, mode) ;**

```
FILE *ptr_fich;
ptr_fich = fopen ("tp1.c", "w");
if (ptr_fich == NULL) puts(“Erreur d'ouverture”);
```

ptr-fich pointe sur la structure contenant les informations du fichier. FILE est une structure interne. tp1.c est le nom du fichier sur disque. w est mis pour write (écriture). Les autres modes d'ouverture sont :

mode	sens	mode	sens
r	read	r+	read et write
w	write	w+	write et read
a	append	a+	append (r+w)

On peut ajouter la lettre b, a chacun des types, pour dire que le fichier est binaire, sinon il est texte (caractère) par défaut.

On utilise pour lire un fichier ou y écrire, différentes instructions, selon le type du fichier.

Fichiers textes

On utilise les ordres fscanf, fprintf etc.. de la même façon que pour les lectures/écritures standard.

```
fscanf (nom_interne, "format", &var);
c = getc(nom_interne);
fprintf (nom_interne, "format", var);
putc(c, nom_interne);
```

Fichiers binaires

On utilise les ordres fread et fwrite pour les lectures/écritures de tels fichiers, après ouverture en rb, ou en wb. (ptr_zone : adresse de la zone (émettrice ou réceptrice) du buffer, taille : celle de la zone à lire, nbre : nombre de zones à lire, ptr_fich : pointeur sur le fichier utilisé.)

```
fread(ptr_zone, taille, nbre,ptr_fich);
/* ptr_zone: adresse zone */
fwrite(ptr_zone, taille, nbre,ptr_fich);
```

Dans ce cas le transfert de l'information mémoire vers le fichier sans interprétation : La copie exacte du bloc mémoire est stocke dans le fichier. Un fichier binaire ne peut être lu avec un éditeur de texte, contrairement au fichiers textes. On utilise l'instruction sizeof pour déterminer automatiquement la taille de la zone (ou élément).

1. La structure de l'instruction fwrite est pareille au fread.
2. Lors de l'ouverture d'un fichier, il peut arriver qu'on se trompe en écrivant le nom du fichier par exemple. Pour éviter tout problème, on utilise un "if" pour écrire un message à l'écran au cas où il y a erreur (le fichier n'existe pas).
3. Il y a deux avantages à utiliser les fichiers binaires. D'abord la rapidité (donc gain de temps) car il n'y a pas de transformations du binaire vers le type caractère au sens de l'interprétation, puis la connaissance de la taille du fichier. Les ordres fprintf et fscanf dépendent de la valeur de la zone et pas uniquement du type.

Fermeture de fichier

La fermeture de fichier se fait automatiquement. Cependant, si l'on désire utiliser plus tard le fichier, il est recommandé de le fermer. A l'ouverture, on se positionnera en début de fichier. L'ordre de fermeture d'un fichier a pour syntaxe : fclose (nom_interne) Exemples d'applications

Exemple 29. Dans ce programme

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

char nomfich[20];
FILE *in, *out;
void ouvrir() {
  printf("\nNom de fichier a creer : ");
  scanf("%s",nomfich);
  if ( (out = fopen(nomfich, "a") ) == NULL )
    printf("***** erreur ouverture ***"),
  exit(-1);
}
void entree() {
  int nb_vect, i; char c;
  printf("\n Entrer votre message :");
  getchar();
  do {      c = getchar(); putc( c, out);
  } while (c != '\n');
}
void sortie() {
  char c;
  printf("\n Contenu du fichier : \t %s\n ", nomfich);
  do {     c = getc(in); putchar( c);
  } while (c != '\n');
}
int  main() {
  int a;
  printf("\n adresse de a =%#X",&a);
  ouvrir(); entree();    fclose(out);
  in= fopen(nomfich, "r");
  sortie(); return 0;
}
```

Utilisation de fichier binaire

Exemple 30. A travers le programme suivant, on voit comment on crée puis utilise les fichiers binaires.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int NBetud=0;
typedef struct { int jj; int mm; int aa; }date;
typedef struct { char nom[40]; date naiss; }etudiant;
FILE *sortie, *entree;
etudiant etud;
void creation() {
  printf("\n\tPour finir donner ZZZ ---\n");
  do {
    printf("nom  :");     scanf("%s",etud.nom);
    if (!strcmp(etud.nom,"ZZZ") ) break;
    printf("date de naissance sous forme jj/mm/aa :");
    scanf("%d/%d/%d", &etud.naiss.jj,  &etud.naiss.mm,
  &etud.naiss.aa);
    NBetud++;
    fwrite(&etud, sizeof(etudiant),1,sortie);
  } while(1);
}
void affichage() {
  int i,j,num = 1;
  for(j=0; j< NBetud; j++) {
    fread(&etud, sizeof(etudiant),1,entree);
    printf("\n\tNumero: %d", num++);
    printf("\n\tNOM : %s \n", etud.nom);
    printf("\n\tDATE DE NAISSANCE: %d/%d/%d \n", etud.naiss.jj,
   etud.naiss.mm, etud.naiss.aa);
  }
}
int  main(int nb, char * arg[]){
  int i;
  if(nb < 2)
    puts("\nNom de Fichier en Argument?: "),  exit(1);
  if( (sortie = fopen(arg[1], "wb") ) == NULL )
    printf("Erreur d'ouverture en  Ecriture "), exit (-1) ;
  printf("\n-----DEBUT DE  CREATION FICHIER_______");
  creation();
  fclose(sortie);
  printf("\n-----FIN CREATION FICHIER_______");
  printf("\n-----DEBUT DE L' UTILISATION_______");
  if( (entree = fopen(arg[1], "rb") ) == NULL){
    puts("Erreur d ouverture en: Lecture ");
    return -1;
  }
  affichage();
  printf("\n-----FIN DE L' UTILISATION_______\n");
  //  fclose(entree);
  return 0;
}
```

4.5 Exercices

1. Donner la sortie du programme suivant :

```
1 int main(){
2   char *c[]={ "This", "is My", "first", "program"};
3   char **cp=c+2;
4   int a[][5]={{1,2,3,4,5},{6,7,8,9,10} };
5   int *p=a[0];
6   cout << "\naddr a: "<< a <<\taddr c: "<< c;
7   cout << "\n a[1][2]= " << a[1][2];
8   cout << "\n *(a+1)+ 2 = " << *(a+1)+2;
```

```
9   ␣␣cout␣<<"\n p + 1 = "␣<<␣p+1␣;
10  ␣␣cout␣<<"\n *(p + 1)+ 2 = "␣<<␣*(p+1)+2;
11  ␣␣cout␣<<"\n c[2] = "␣<<␣c[2]␣;
12  ␣␣cout␣<<"\n *(*cp + 1) = "␣<<␣*(*cp+1)␣;
13  ␣␣return␣0;
14  }
```

2. Ecrire la fonction Puissance qui calcule z^m , z est un réel et m est un entier positif.
3. Ecrire programme qui donne les valeurs de P(x) pour N valeurs de $x \in [A\ ,\ B]$; Le degré et les coefficients de P, le nombre de valeurs N et l'intervalle $[A\,,\,B]$ sont lus à partir du clavier. La fonction polynome appelle une fonction Puissance (voir l'exercice précédant) .
4. Modifier le programme de l'exemple 30, en faire 2 programmes indépendants, le premier crée le fichier et le second utilise ce fichier.
5. **TP :** Ecrire un programme qui fait la somme de 2 grands nombres entiers (supérieur à 10^{10}). Faire des fonctions pour la lecture, la somme, ... Ind : considérer le nombre comme une chaîne de caractères.
6. Reprendre la question en donnant les nombres comme paramètre au programme. Voir la section 4.2.5

5 Création de Bibliothèque

5.1 Programme Multi-fichiers

Lorsque nous avons un gros projet, il est préférable de diviser le programme en plusieurs fichiers, de les compiler indépendemment les uns des autres
Le programme est eclaté sur 3 fichiers myMain.c (procedure principale), myFonc.c (contient le code des fonctions) et myFonc.h (les entetes des fonctions). Voir listing 5.1

Listing 5.1: Compilation separée

```
1  //------entete myFonc.h
2  double carre (double);
3  float puissance(float, int);
4  //------ fonctions : myFonc.c
5  double carre (double a) {  return a * a; }
6  float puissance(float x, int n ){ // x^n
7    float y=1; int i;
8    if (x == 0)    return 0;
9    for (i=1; i <=n; i++)       y *=  x;
10   return y;
11 }
12 //----------- fichier principal  myMain.c ----------------
13 #include <math.h>  // pour utiliser sinus
14 #include <stdio.h> // pour utiliser printf
15 #include "myFonc.h" // pour mes fonctions
16 int main () {
17   double x = 14.5, y;
18   printf("\nvaleur de x: %.2lf",x);  y = sin (x);
19   printf("\n valeur de sin (x) =%.3lf", y);
20   printf("\n x^2 est %.2lf",carre (x) );
21   printf("\n x^4 = %.2lf", puissance(x, 4) );
22 }
```

Chaque fichier peut être compilé et le fichier objet est obtenue. mais pour obtenir l'exécutable nous les deux fichiers. Ainsi, dans le premier fichier, nous utilisons des fonctions qui ne sont pas là (le péché, et la somme carrés), le second fichier ne contient pas la fonction principale qui est requis pour tout programme.
Pour créer le programme exécutable, on compile chaque fichier seul, et ensuite un lien avec la bibliothèque mathématique.

```
gcc -c myMain.c
gcc -c myFonc.c
gcc -lm myMain.o myFonc.o  -o myexec
```

Les deux premières commandes de compiler seulement (-c option : la compilation), la source prog1.c et mylib1.c pour obtenir les objets fichier myMain.o et myFonc.o (les noms sont donnés par défaut). La dernière commande de relier tous les éléments du programme, c'est-à-dire les deux objets de l'utilisateur (myMain.o, myFonc.o) et la bibliothèque mathématique ("-l" mis pour lib et "m" pour math.a), qui contient le code de la fonction sinus . Le programme est maintenant myexec (option -o : o mis pour output)
On peut également écrire en une seule commande comme cela (la compilation et l'edition de lien dans la même commande) :

```
gcc -lm myMain.c myFonc.c  -o myexec
```

-lm est synonyme de **/usr/lib/libm.a**, en général, **-lYYY** est equivalent a **/usr/lib/libYYY.a**. Par défaut, les bibliothèques sont installees dans /usr/lib, et l'en-tête des fichiers dans /usr/include.

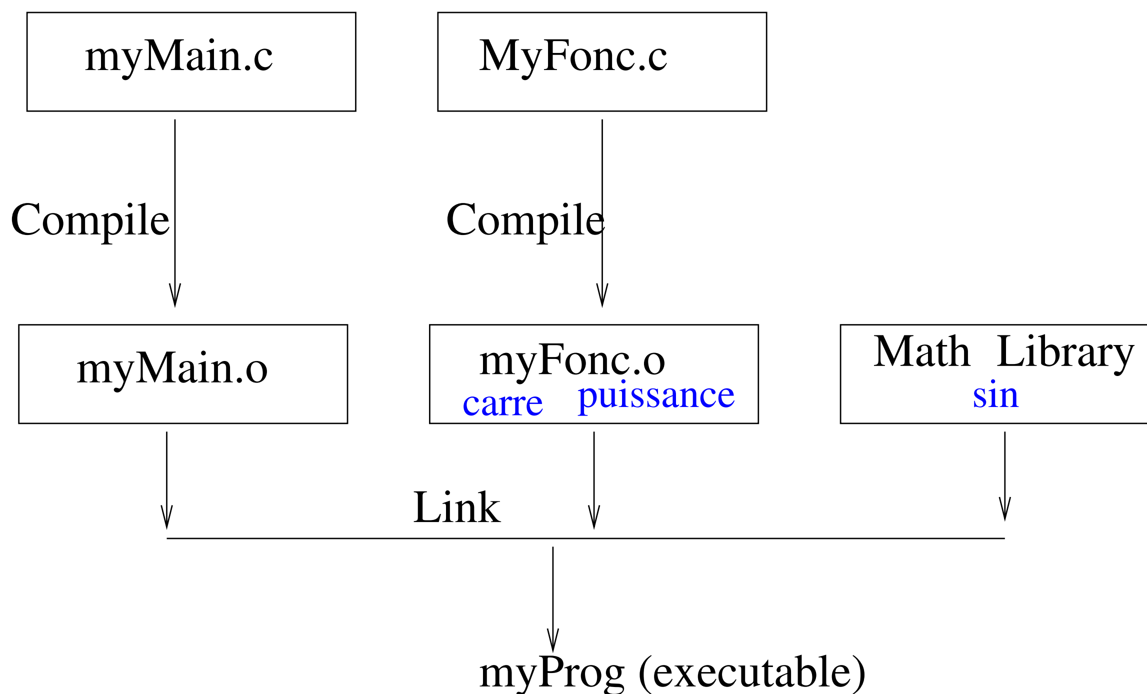


Figure 5.1.1: Edition de plusieurs files

5.2 Archivage de Fonctions

Nous te mylib1.c utiliser le fichier (qui contient deux fonctions : la place et la somme) comme exemple pour montrer comment créer un fichier de bibliothèque d'archives. Cette bibliothèque peut être utilisée par toute personne comme un objet compilé, cela est le cas, par exemple, de la bibliothèque mathématique, en libmath.a fichier, nous avons toutes les fonctions standard (sin, cos, exp, log, ..).

ar (pour archive) de commande permet de créer des archives, avec diverses options. Le plus utile est r, de remplacement ou de création de fichiers dans les archives. Pour plus de détails, de type "man ar" (rappelons "l'homme" est synonyme de manuel, afin de lire le manuel sur commande donnée).

Après que l'utilisation ranlib pour créer un index de toutes les fonctions dans les archives, afin de faciliter la recherche. Pour lister le contenu d'une archive utilisation t option, vérifier par exemple **ar t /usr/lib/libm.a**

Supposons que nous travaillons en Prog (ie /home/user1/Prog) répertoire, de sorte que l'archive est créee dans la bibliothèque Lib (i.e /home/user1/Lib) et de l'inclure dans le fichier Include (i.e /home/user1/Include). En Lib, nous créons libpers.a, qui contient les deux l'objet de somme et de la fonction carre.

```
ar r libpers.a myFonc.o
ranlib libpers.a
```

Dans le fichier Include/myFonc.h, nous avons mis la fonction déclarations :

```
double carre (double); double somme (double, double);
```

Maintenant, pour créer le programme exécutable, il nous suffit d'écrire la commande suivante,

```
gcc myMain.c -LLib -lpers -IInclude
```

La première option **-L**, donne le chemin pour des répertoires de bibliothèque, le second **-l** pour le nom de la bibliothèque et la troisième pour le répertoire include.

Si nous voulons faire de la bibliothèque d'information pour le public d'utilisateurs, en tant que super-on le mettre en place la norme "/usr /lib" de la bibliothèque et "/usr/include" de fichier d'en-tête, dans ce cas, nous n'avons pas besoin options précédentes (**-L/home/user1/Lib, -I/home/user1/Include**).

5.3 Exercices

1. Reprendre le programme complet de la section 5.1, changer son nom (prog2.c). Mettre dans myLib2.c les fonctions somme de trois nombres, et produit des éléments d'un vecteur. Sortir les résultats sur écran. Faire des compilations séparées.
2. Reprendre l'exercice précédant, en faisons l'archivage des fonctions Puissance et Polynome de mylib.c. Faire un appel a ces fonctions archivées depuis votre programme test.
3. Créez dans votre répertoire home, 3 répertoires : Prog, Inclure, Lib. Placez les fichiers sources (prog2.c, myLib2.c, myLib2.h) dans les répertoires correspondants. Créer l'exécutable Prog2 à partir du main et de la bibliothèque archivée.
4. Déplacez-vous dans le répertoire Prog, y créer des bibliothèques pour les fonctions de mylib2.c, puis de créer l'exécutable pour prog2.c.
5. Ajouter à votre librairie, la fontion somme de grands nombres, vu au TP 5 .