République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Supérieure en Génie Electrique et Energétique d'Oran

Polycopié du cours, travaux dirigés et travaux pratiques des premiéres années du cycle préparatoire

-Module Informatique-

Rédigé par Docteur Sabria HADJ TAYEB

Année Universitaire
2018- 2019

SOMMAIRE

Chapitre 5. Algorithmique

INFORMATIQUE 2

Chapitre 1. De l'algorithmique à la programmation

Avant propos

Ceci est un polycopié pédagogique du module Informatique des premières années classes préparatoires sciences et techniques.
Ce manuel comprend l'ensemble des cours, travaux dirigés et travaux pratiques durant le premier et le deuxième semestre de l'année 2018/2019 conformément au programme ministériel des classes préparatoires en sciences et techniques mis en vigueur en 2015.

Le premier semestre intitulé *Informatique 1* comprend cinq chapitres :

Dans le premier chapitre intitulé « *Architecture des ordinateurs* », l'étudiant aura un aperçu sur la structure de l'ordinateur tout en assimilant les fonctionnalités de chaque composant.
Le but du de second chapitre 2 « *Machine de John von Newmann* » est de connaitre le cycle d'exécution de l'instructions de manière très générale.
Le troisième chapitre intitulé *« Système de numération »* à pour but la maîtrise des quatre opérations de base, la représentation des nombres négatifs, l'arithmétique binaire et les conversions en format IEEE 754.
Le quatrième chapitre *« Algébre de Boole »* comprend généralement la compréhension de la notion d'expression algébrique, des portes logiques et la simplification des expressions algébriquement et via le tableau de Karnaugh.
Enfin, le cinquième et important chapitre *« Algorithmique »* à pour objectif d'assimiler à l'étudiant les bases algorithmiques. A la fin de ce chapitres, l'étudiant sera apte à écrire un algorithme séquentiel, conditionnel, itératif et avec types complexes (tableaux et matrices).
Un ensemble d'exemples et d'exercices seront traités en cours pour chaque partie et à chaque chapitre est associée une fiche de travaux dirigés.

Le deuxième semestre intitulé *Informatique* 2 comprend aussi cinq chapitres :
Le premier chapitre de *« De l'algorithmique à la programmation »* a pour objectif l'étude et la maitrise de la syntaxe et concepts de base du langage du programmation et la translation d'un algorithme en un code C.
L'objectif du deuxième chapitre intitulé *« Les fonctions »* est de prévoir, concevoir, et utiliser les fonctions tout en mettant l'accent sur les limites que peuvent apporter le passage par valeurs d'où l'utilisation des pointeurs.
Le troisième chapitre *« Pointeurs et allocation dynamique »* comprend en premier lieu la maitrise du concept pointeur et la manipulation des pointeurs comme arguments dans les fonctions. La deuxième partie est de maîtriser les principes de l'allocation dynamique.
Le quatrième chapitre *« La récursivité »* est la compréhension et l'implémentation de la récursivité.
Le dernier chapitre *« Structures complexes : listes chainées et piles»* vise à assimiler et implémenter les structures complexes à savoir les listes chainées et les piles.

Chaque partie du cours est suivie par des exemples ou/et exercice en code C.
Trois fiches de travaux pratiques avec corrigés englobent des exercices en C de l'ensemble des chapitres du deuxième semestre.

Chapitre 1

Architecture d'un ordinateur

1. Définition d'un ordinateur

Un ordinateur est une machine de traitement de l'information. Il est capable d'acquérir de l'information, de la stocker, de la transformer en effectuant des traitements quelconques, puis de la restituer sous une autre forme.

2. Composants d'un ordinateur

En ouvrant le boitier d'un ordinateur, nous trouvons une carte mère qui sert à rassembler tous les composants de l'ordinateurs (CPU, mémoire, les disques, les cartes d'extension…).

2.1. Le processeur (CPU)

2.1.1. Définition : Le CPU (***C**entral **P**rocessing **U**nit)* est un circuit électronique complexe permettant de manipuler et de traiter les données qui lui sont fournies.

2.1.2. Composants d'un CPU

CPU= Bloc logique de commande (séquenceur) + Unité de Traitement (UT)

a) ***Bloc logique de commande (séquenceur)*** **:** Il organise l'exécution des instructions au rythme d'une horloge et élabore tous les signaux de synchronisation internes ou externes du microprocesseur. Il contient :

- ***Le Compteur Ordinal (CO)*** appelé aussi *Compteur de programme* ***(CP)***, ou *instruction pointer (IP).* Le CO est un registre dont le contenu est initialisé avec l'adresse de la première instruction du programme. Il contient toujours l'adresse de l'instruction à exécuter.

- ***Le Registre d'Instruction (RI)*** **:** Il contient l'instruction en cours de traitement.

b) ***Unité de Traitement (UT) :*** **C**ette unité contient une UAL et des registres :

- ***Unité Arithmétique et Logique (UAL) :*** Un circuit complexe assurant les fonctions logiques (ET, OU, Comparaison,…) ou arithmétiques.

***- Registre d'état (PSW)* :** C'est un ensemble de bits au sein du CPUoù chacun est un indicateur dont l'état dépend du résultat de la dernière opération effectuée par l'UAL. Les drapeaux présents dans la plupart des microprocesseurs actuels sont :

- ✓ *Drapeau Z (Zéro)* : Indique que le résultat de l'opération est nul.
- ✓ *Drapeau C (Carry) :* Indique que le résultat n'est pas complet puisqu'il ya une retenue.
- ✓ *Drapeau N/S (Negative /Signe) :* Indique que le résultat est inférieur à 0.
- ✓ *Drapeau V/O (OVerflow) :* Indique un dépassement de capacité (la taille du processeur est petite pour stocker le résultat).

- Registre accumulateur (ACC) : C'est des registres de travail servant à stocker une opérande (donnée) au début d'une opération arithmétique et le résultat à la fin de l'opération.

2.2. La mémoire centrale (MC)

La MC est divisée physiquement en cases de taille fixe. Chaque case possède une adresse.
Les cases peuvent être adressées par une opération de lecture ou d'écriture.
Selon le type d'un ordinateur, une case est constituée de 8 bits (**BI**nary digi**T** :élément élémentaire d'information) ou d'un nombre plus grand de bits : 16, 32…, on parle alors d'un *mot mémoire (word)*.
DONC :
Un *mot mémoire* représente une unité d'information adressable (que toute opération de R/W porte sur un mot mémoire).
A chaque mot mémoire est associé une *adresse unique* indiquant la position en mémoire (les adresses sont séquentielles) et un *contenu* représentant une instruction ou une donnée.

La MC contient principalement deux types d'informations :

- Les *instructions* des différents programmes,
- Les *données* nécessaires à l'exécution des programmes.

Les principales caractéristiques d'une mémoire sont les suivantes :

- La ***capacité***, représentant le volume global d'informations (en bits) que la mémoire peut stocker ;
- Le ***temps d'accès***, correspondant à l'intervalle de temps entre la demande de lecture/écriture et la disponibilité de la donnée ;
- Le ***temps de cycle***, représentant le temps minimal entre 2 accées mémoires successifs ;
- Le ***débit***, définissant le nombre d'informations lues ou ecrites /seconde;
- La ***non volatilité*** caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement.

2.2.1. Types de mémoires : Il existe de types de mémoires : les mémoires vives (RAM) et les mémoires mortes (ROM).

a) Les mémoires vives (RAM) : La RAM (**R**andom **A**cces **M**emory) est une unité de stockage permettant le sauvegarde des informations pendant tout le temps de fonctionnement d'un ordinateur.
La RAM est une mémoire volatile et dont son contenu peut être modifié *(R/W)*. On trouve les SRAM, les DRAM, les SDRAM et actuellement c'est les DDR.

b) Les mémoires mortes (ROM): La ROM (**R**ead **O**nly **M**emory) est une mémoire non volatile et dont le contenu ne peut être modifié *(R)*.
Les ROMs stockent des programmes invariables, comme par exemple le programme exécuté au démarrage (BIOS).
Nous trouvons PROM, EPROM, EEPROM, FLASH EPROM.

2.2.2. Les registres de la mémoire centrale : Dans la mémoire, on trouve deux types de registres :
- Le registre d'adresse mémoire (R.A.M) : Il contient l'adresse d'un mot mémoire.
- Le registre de données (mot) mémoire (RDM) : Il contient le contenu du mot mémoire.

2.2.3. Capacité de la mémoire : La capacité de la mémoire s'exprime en fonction du nombre de mots mémoire ainsi qu'au nombre de bit par mot.
- Soit k la taille du bus d'adresses (taille du registre R.A.M)
- Soit n la taille du bus de données (taille du registre RDM ou la taille d'un mot mémoire)

On peut exprimer la capacité de la mémoire centrale soit en nombre de mots mémoire ou en bits (octets, kilo-octets,….)

Calcul de la capacité de la MC

La capacité = 2^k Mots mémoire

La capacité = 2^k * n Bits

Avec k :taille du registre adresse mémoire ;
n : taille du registre données mémoire

Conversions

1 Octet= 1 Byte = 8 bits
1 KO (kilo octet) = 1024 Octets = 2^{10} Octets
1 MO (mega octet) = 1024 KO
1 GO (giga octet) = 1024 MO
1 TO (Tera octet) = 1024 GO
1 PO (Peta octet) = 1024 TO
1 EO (Exa octet) = 1024 PO
1 ZO (Zetta octet) = 1024 EO
1 YO (Yotta octet) = 1024 ZO

2.2.4. Notion de mémoires caches : Le cache est une mémoire rapide contenant une copie d'une zone de mémoire centrale, il sert de couche intermédiaire entre le CPU et la mémoire ceci afin de diminuer les temps d'accès et accélérer le traitement des instructions.

a) Fonctionnement de la mémoire cache

1. Le CPU demande une information.
2. La recherche se fait d'abord dans la mémoire cache :
   - Si l'information existe dans la mémoire cache, elle est transmise au CPU, on parle de *succès de cache* (*cache hit*).
   - Si elle n'existe pas dans la cache, on parle alors de *défaut de cache* (*cache miss*), et la recherche se fera dans la MC. Dans ce cas, la mémoire cache enregistre la copie de l'information trouvée pour utilisation ultérieure si besoin.

b) Type de cache

- Le Cache **L1 (primaire)** directement intégré dans le CPU, cette mémoire est très rapide et de petite taille. Elle est divisée en L1 données et L1 instructions.
- Le Cache **L2 (secondaire)** est situé au niveau du boîtier contenant le processeur (dans la puce).
  Ce cache contient les données qui ne se trouvent pas dans L1.
  Cette mémoire est plus grande que L1.
- Le Cache **L3 (externe)** est située au niveau de la carte mère, il stocke les données qui ne se trouvent pas dans la L1 et la L2.
  C'est une mémoire beaucoup plus lente que L1 et L2.

2.3. Les périphériques

Nous distinguons quatre sortes de périphériques :

- Les ***périphériques d'entrée*** **:** Ils permettent effectivement de fournir à l'ordinateur les données à traiter (Clavier,souris, scanner …)
- Les ***périphériques de sortie*****:** Ils permettent à l'utilisateur de recevoir des informations venant de la machine : écran, imprimante...
- Les ***Périphériques d'entrée sortie***: Ils permettent la circulation de l'information dans les deux sens : disque dur, lecteur de disquettes...
- Les ***péripheriques de stockage*****:** Ils sont des péripheriques d'Entrée sortie permettant le stockage d'information de maniére permanente (disque dur …)

2.4. Les bus

Un bus est l'ensemble de liaisons physiques (câbles, pistes de circuits imprimés, etc.) pouvant être exploitées en commun par plusieurs éléments matériels afin de communiquer.

Il existe 3 sous ensembles de bus :

a) Le bus d'adresses **:**

- Il transporte les adresses mémoire auxquelles le processeur souhaite accéder pour lire ou écrire une donnée.
- Il s'agit d'un bus unidirectionnel.

b) Le bus de données **:**
- Il véhicule les instructions en provenance ou à destination du processeur.
- Il s'agit d'un bus bidirectionnel.

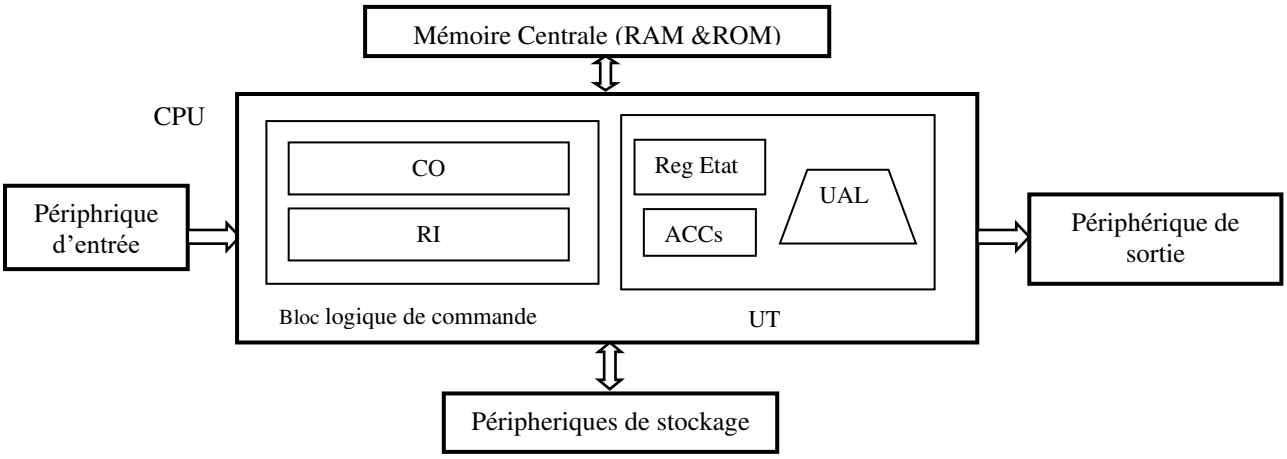
*c) Le bus de contrôle (bus de commandes) :*Il transporte les ordres et les signaux de synchronisation en provenance de l'unité de commande et à destination de l'ensemble des composants matériels.

Nous trouvons 2 types de bus :

a) Le bus système (bus interne) permet au processeur de communiquer avec la MC.

b) **Le bus d'extension (b*us d'entrée/sortie*)** permet aux divers composants liés à la carte-mère de de communiquer entre eux.

3. Schema récapitulatif



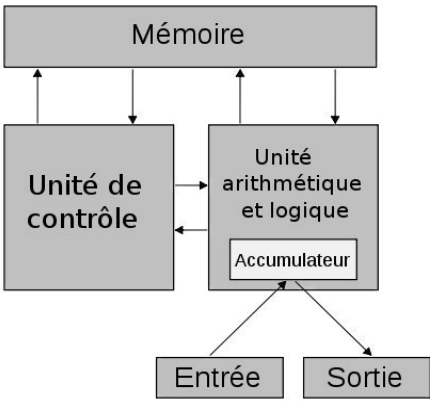
Chapitre 2

Machine VON NEUMANN et exécution d'instruction

1. Introduction

En 1945, le mathématicien John VON NEUMANN proposa la construction de la machine modèle EDVAC. Une machine caractérisée par la présence d'*une unité arithmétique et logique*, *une unité de contrôle*, d'une unique *mémoire centrale* qui contient à la fois les instructions du programme et les données manipulées et de *dispositifs d'Entrée sortie*.

1. *L'**unité arithmétique et logique** (UAL)* pour d'effectuer les opérations de base ;
2. *L'**unité de contrôle***, chargée du séquençage des opérations ;
3. La ***mémoire*** (RAM et ROM)
4. ***Les dispositifs d'entrée-sortie*** pour la communication avec le monde extérieur.



Pour exécuter une instruction au niveau, il faut :
- Rechercher l'instruction dans la mémoire,
- Décoder l'instruction,
- Recherche dans la mémoire les données concernées par l'instruction,
- Déclencher l'opération adéquate sur l'UAL ou l'E/S,
- Range au besoin le résultat dans la mémoire.

2. Cycle d'exécution d'une instruction

2.1. Phase 1. Recherche d'instruction

Etape 1 : Le compteur ordinal contient l'adresse de l'instruction suivante du programme. Cette valeur est placée sur le bus d'adresse par l'unité de commande qui émet un ordre de lecture.
Etape 2 : Au bout d'un certain temps (temps d'accès à la mémoire), le contenu de la case mémoire est mis dans sur le bus de donnée.
Etape 3 : L'instruction est stockée dans le registre d'instruction RI.

Etape 4 : Le RI contient maintenant le premier mot de l'instruction qui peut être codée sur plusieurs mots. Ce premier mot contient le code opératoire qui définit la nature de l'opération à effectuer (addition, ...) et le nombre de mots de l'instruction.
Etape 5 : L'unité de commande transforme l'instruction en une suite de commandes élémentaires nécessaires au traitement de l'instruction.

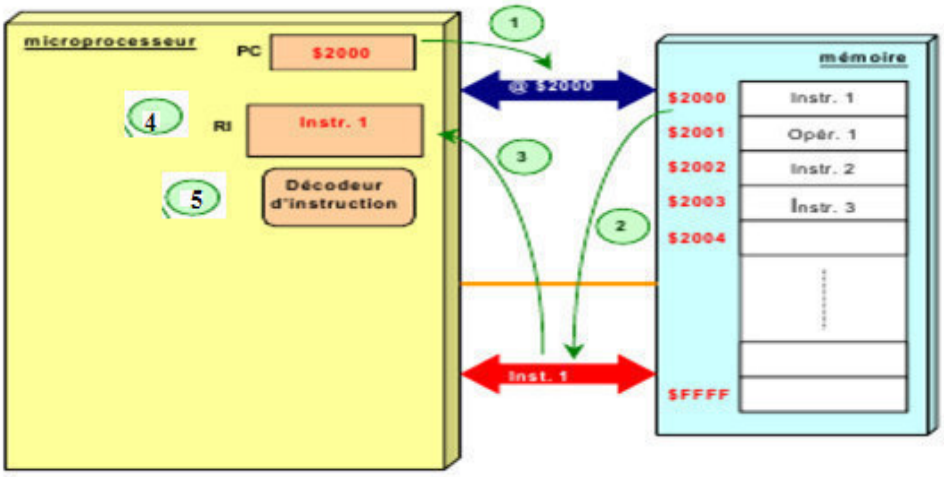


Figure 1. Phase 1 d'exécution d'une instruction

2.2. Phase 2. Recherche des opérandes et exécution

Etape 1 : Si l'instruction nécessite une donnée en provenance de la mémoire, l'unité de commande récupère sa valeur sur le bus de données.
Etape 2 : L'opérande est stockée dans un registre.
Etape 3 : Un ordre est donné par l'unité de commande à l'UAL pour effectuer l'opération.

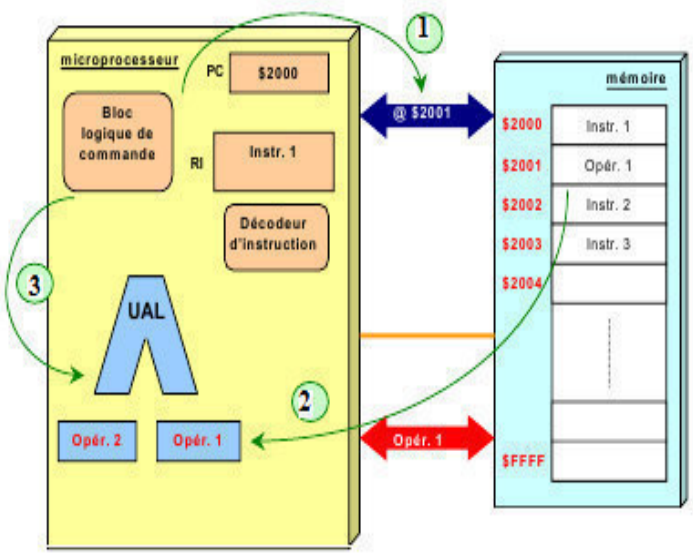


Figure 2. Les 3 étape de la phase de recherche d'opérande et d'exécution d'une instruction

Etape 4 :. Le micro-programme réalisant l'instruction est exécuté.
Etape 5 : Les drapeaux sont positionnés (registre d'état).

2.3. Phase 3. Passage à l'instruction suivante

L'unité de commande positionne le PC pour l'instruction suivante.

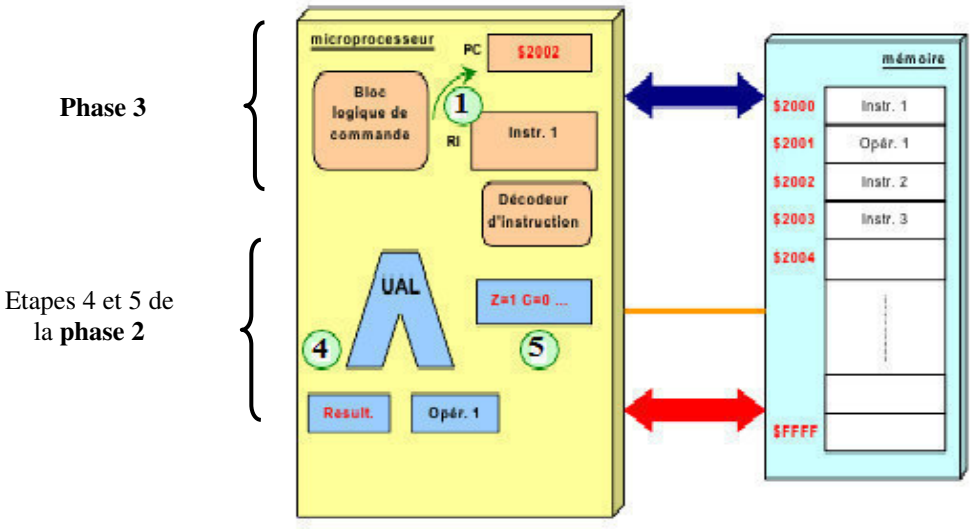


Figure 3. Etapes 4-5 de la phase 2 et etape1 et phase 3 d'exécution d'une instruction

Chapitre 3

Représentation des nombres

1. Système de numération

Le codage d'une information consiste à établir une correspondance entre la représentation externe ou décimal et sa représentation interne (suite de 0 et 1).
4 systèmes sont identifiés :

a)Le système décimal : Le système décimal est un **système de numération** utilisant la **base** dix. Dans ce système, les puissances de **dix** et leurs multiples bénéficient d'une représentation privilégiée.

b) Le système binaire : Le système binaire est un système de numération utilisant la base 2. Les valeurs permises sont 0 et 1.
On passe d'un nombre binaire au suivant en ajoutant 1, comme en décimal, sans oublier les retenues et en utilisant les tables d'additions suivantes :
0+0=0, 0+1=1, 1+0=1, 1+1=10
L'arithmétique binaire est utilisée par les machines électroniques les plus courantes (calculatrices, ordinateurs, etc.) car la présence ou l'absence de courant peuvent servir à représenter les deux chiffres 0 et 1.
0 représente l'état fermé, 1 représente l'état ouvert.

c) Le système octal : Le système octal utilise un système de numération ayant comme base 8.
Il faut noter que dans ce système nous n'aurons plus 10 symboles mais 8 seulement :
0, 1, 2, 3, 4, 5, 6, 7.

d) Le système hexadécimal : Le système hexadécimal utilise les 16 symboles suivants : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Base 10	Base 16	Base 2
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

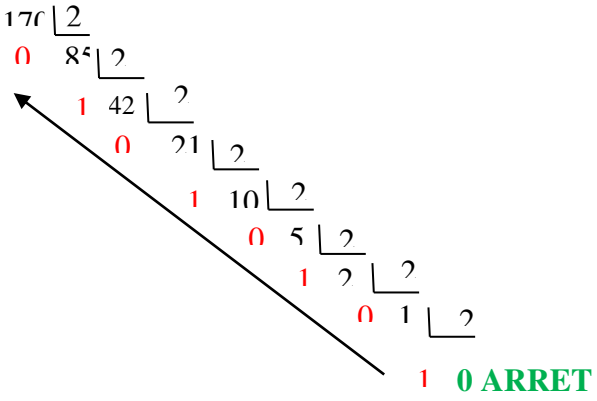
Tableau 1 : Tableau de correspondance base 2, base 10 et base 16

2. Conversions et changements de base

2.1. Conversion d'un nombre décimal en binaire

La méthode consiste à diviser le nombre donné par la base demandée tant que c'est possible. On rassemble ensuite les restes en partant de la fin et on obtient l'écriture dans la nouvelle base.

Exemple 1 : $(170)_{10}=(\ ?\ )_2$



On fait des divisions successives, on s'arrête quand le quotient =0

$(193)10 = (10101010)_2$.

2.2. Conversion d'un nombre de base N vers la base 10

Pour passer d'un nombre en base *N* à un nombre en base 10, on peut appliquer la méthode suivante : Soit *K* le nombre en base *N* à convertir. Pour tout chiffre *c* de rang *r* dans *K*, on calcule $c \times N^r$. La représentation de *K* en base 10 est la somme de tous les produits.
Le comptage de r commence à zéro de la droite vers la gauche.

Exemple 2 : $(10110)_2 = 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0 = (22)_{10}$
$(3FA)_{16} = 3\times16^2 + 15\times16^1 + 10\times16^0 = (1\,018)_{10}$
$(745)_8 = 7\times8^2 + 4\times8^1 + 5\times8^0 = (485)_{10}$
$(5AF)_{16} = 5\times16^2 + A\times16^1 + F\times16^0 = (1455)_{10}$

3. Représentation des nombres relatifs

Il faut pouvoir écrire des entiers négatifs en prévoyant un bit de signe, placé en tête. Un bit de signe nul indique une valeur positive, un bit de signe positionné à 1 indique une valeur négative.

- Complément à 1 : Ce codage, fort simple, consiste à inverser la valeur de chaque bit composant une valeur binaire si le nombre est négatif.
- Complément à 2 : On a introduit la représentation par complément à deux. Celle-ci consiste à réaliser un complément à un de la valeur, puis d'ajouter 1 au résultat.

Remarques importantes :

1. Représentation du nombre :
   -Si le nombre est positif alors Binaire pur=binaire signé = Complément à1= Complément à2
   -Si le nombre est négatif alors Complément à1= Inverser tous les bit sauf le bit du signe.
   Complément à2= Complément à 1 +1
2. Les additions sont font en complément à 2 :
   Nous avons un débordement :
   - Si la somme de deux nombres positifs donne un nombre négatif.
   - Ou la somme de deux nombres négatifs donne un nombre positif .

   Il ne peut jamais y avoir un débordement si les deux nombres sont de signes différents. Ce qui est logique. Supposant que j'ai A+ B (avec A positif et B négatif) avec A et B sont représentés sur 8 bits. L'addition me donne un résultat qui doit être entre B et A, ce qui veut dire que le résultat est représentable sur 8 bits.

Exemple 3: Sur une machine de 8 bits
$(-7)_{10} = (?)_{ca2}$:

$(-7)_{10} = (0\ 0000111)_{bp}$ (bp : binaire pur ou valeur absolue)
$(-7)_{10} = (1\ 0000111)_{bs}$ (bs : binaire signé => introduire le signe)

Exemple 4 : Voici une addition de -7 et +9 réalisée en Ca2 sur une machine de 8 bits:

Nombre	Binaire pur	Binaire signé	Ca1	Ca2
$(-7)_{10}$	(00000111)	(10000111)	(11111000)	(11111001)
$(+9)_{10}$	(00001001)	(00001001)	(00001001)	(00001001)

L'addition en ca2 : $(\mathbf{1}1111001)_{ca2} + (\mathbf{0}0001001)_{ca2} = (\mathbf{10}0000010)_{ca2}$
Nous avons une machine de 8 bits et le resultat est sur 9 bits.
On effectue l'addition de 2 nombres de signes differents
DONC :
on **'ignore'** la retenue Donc le résultat = $(\mathbf{0}0000010)_{ca2} = (+2)_{10}$

Exemple 5: Coder les entiers 61 et -61 sur un octet en utilisant la représentation par le signe et la valeur absolue. Montrer que l'addition binaire de ces entiers ainsi codés produit un résultat incorrect. Montrer qu'en revanche le résultat est correct si ces entiers sont codés en utilisant la représentation par le complément à 2.

Addition en binaire

$(61)_{10}$ $(\mathbf{0}0111101)_2$
$(-61)_{10}$ $+(\mathbf{1}0111101)_2$

$= (11111010)_2 = (-122)_{10}$
C'est incorrect

Addition en Complément à 2

$(61)_{10}$ $(00111101)_{ca2}$
$(-61)_{10}$ $+ (11000011)_{ca2}$

00 (0)

4. Représentation en virgule fixe

4.1. Conversion d'un nombre décimal en binaire

Exemple 6: Soit le nombre $(10,625)_{10}$ en convertir en binaire en virgule fixe.
La partie entière 10 pas de changement on utilise la méthode des divisions successives par 2.

$(10)_{10}=(1010)_2$
On prend 0,625x2 (base)=**1**,25 (on garde le 1)
On prend 0,25x2=**0**,5(on garde le 0)
On prend 0,5x2=**1**,**0** (on garde le 1 et on s'arrête car après la virgule il ya un zéro)

On prend les chiffres du haut vers le base, ce qui donne en virgule fixe $(1010,101)_2$

4.2. Conversion d'un nombre binaire en décimal

Exemple 7 : $(1010,101)_2=(\ ?)_{10}$

$$= 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3}$$

5.Représentation des nombres réels suivant la norme IEEE 754

Les nombres à **virgule flottante** sont les nombres les plus souvent utilisés dans un **ordinateur** pour représenter des valeurs non **entières**. Ce sont des approximations de **nombres réels**. Les nombres à virgule flottante possèdent :

- un signe *s* (dans {-1, 1}),
- une **mantisse** *m*
- un **exposant** *e*.

Un tel triplet représente un réel $s.m.b^e$ où *b* est la base de représentation (généralement 2 sur ordinateur, mais aussi 16 sur certaines anciennes machines, 10 sur de nombreuses calculatrices, ou éventuellement toute autre valeur). En faisant varier *e*, on fait « flotter » la virgule décimale. Généralement, *m* est d'une taille fixée.
Ceci s'oppose à la représentation dite en **virgule fixe**, où l'exposant *e* est fixé.

La **norme IEEE 754** spécifie deux formats de nombres en virgule flottante (et deux formats étendus optionnels) et les opérations associées. Les deux formats fixés par la norme **IEEE** 754 sont sur 32 bits (« simple précision ») et 64 bits (« double précision »).

	Encodage	Signe	Exposant	Mantisse	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	24 bits	7
Double précision	64 bits	1 bit	11 bits	52 bits	53 bits	16

Dans le format IEEE simple précision :
Exposant Biaisé = Exposant normalisé + Biais
Biais=$2^{\text{taille de l'exposant-1}}$-1=127

Exemple 8: Convertir $(-118{,}625)_{10}$ en virgule flottante selon la norme IEEE 754

- Le signe : négatif => bit de signe égal à 1
- Convertir en binaire le nombre 118,625 en virgule fixe
  0 ,625x2=1,25
  0,25x2=0,5
  0,5x2=1,0 on s'arrête car il ya un zéro après la virgule
  118,625= $(1110110\ ,101)_2$
- Décalage :Il faut décaler la virgule vers la gauche de façon à ne laisser qu'un seul 1sur la gauche ce qui donne $(1110110\ ,101)_2$= **1**,110110 101x2^6
  On a trouvé exposant=6 et Mantisse pseudo normalisé =110110101
- Calcul de l'exposant biaisé

Exposant Biaisé = Exposant normalisé + Biais =6+127=$(133)_{10}$=**$(10000101)_2$**

Donc : $(-118{,}625)_{10}$ en virgule flottante est

Signe	Exposant biaisé	Mantisse
1	10000101	110110 10100000000000000

Exemple 9: $(525{,}5)_{10}$ en virgule flottante IEEE .

- 525,5 est positif donc le 1er bit sera 0.
- Sa représentation en base 2 est la suivante : $(1000001101{,}1)_2$
- En normalisant, on trouve : 1,0000011011*2^9
- On ajoute 127 à l'exposant qui vaut 9 ce qui donne 136, soit en base 2 : 10001000
- La mantisse pseudo normalisé 0000011011.
  La représentation du nombre 525,5 en binaire avec la norme IEEE est donc :

0 1000 1000 0000 0110110000000000000 = $(4403600)_{16}$

Chapitre 4

Algèbre DE BOOLE

1. Définition

L'algèbre de Boole est une algèbre binaire qui étudie la logique. Elle est utilisée pour faire l'étude des système possédant deux états qui s'exclus mutuellement.

- Un interrupteur est ouvert ou non ouvert (fermé)
- Une lampe allumée ou non allumée (éteinte)
- Une porte ouverte ou non ouverte (fermée)…

- Une **variable booléenne** ou logique est une variable qui prend deux valeurs : ***VRAI*** ou ***FAUX*** ou bien encore 1 ou 0.

- Une **fonction logique ou booléenne** relie N variables via un ensemble d'opérateurs logiques.

- Il existe **3 opérateurs logiques** de base:
  - Un operateur unaire: Non (NOT) " ¯ "
  - Deux opérateurs binaires: l'opérateur ET (AND) ". " et l'opérateur OU (OR) "+ "

- Nous pouvons consigner le résultat obtenu dans une **table de vérité**.

- Les opérateurs logiques de base peuvent être réalisés par des circuits électroniques : ils sont alors appelés ***Portes logiques.***

2. Axiomes et postulats

Une algèbre de Boole est constituée de :
1. un ensemble E,
2. deux éléments particuliers de E : **0** et **1** (correspondant respectivement à *FAUX* et *VRAI*),
3. deux opérations binaires sur E : **+** et **·** (correspondant respectivement au OU et ET logiques),
4. une opération unaire sur E : ¯ (correspondant à la négation logique).

On acceptera les postulats suivants :

1.	$0 \cdot 0 = 0$	5.	$1+1 = 1$
2.	$0 \cdot 1 = 1 \cdot 0 = 0$	6.	$1+0 = 0+1 = 1$
3.	$1 \cdot 1 = 1$	7.	$0+0 = 0$
4.	$\overline{0} = 1$	8.	$\overline{1} = 0$

De ces postulats découlent les axiomes suivants :
Soient *a*, *b* et *c* des éléments de E :

Commutativité	$a+b = b+a$	$a \cdot b = b \cdot a$
Associativité	$(a+b)+c = a+(b+c)$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$
Distributivité	$a \cdot (b+c) = a \cdot b + a \cdot c$	$a+(b \cdot c) = (a+b) \cdot (a+c)$
Élément neutre	$a+0 = a$	$a \cdot 1 = a$
Complémentation	$a+\overline{a} = 1$	$a \cdot \overline{a} = 0$

3. Fonctions logiques à deux variables

1. **Conjonction:** A.B est VRAI (a and b) si et seulement si A est vrai et B est vrai.
2. **Disjonction :**A+ B est VRAI (A ou B) si et seulement si A est vrai ou B est vrai
3. **Negation :** Le contraire de A est vrai est faux.
4. ***Le OU EXCLUSIF (XOR)*** ou (⊕) :Il se compose de la manière suivante :

   $A \oplus B = \bar{A}B + A\bar{B} = (\bar{A} + \bar{B})(A + B)$

5. ***Le OU EXCLUSIF (XNOR)*** ou (⊗) Le XNOR représente la négation de XOR .

   $\overline{A \oplus B} = A \otimes B = A\,B + \bar{A}\bar{B} = (A + \bar{B})(\bar{A} + B)$

4. Propriétés de l'algébre de Boole

Théorème 1 : Involution :	$\bar{\bar{a}} = a$
Théorème 2 : Idempotence	$a + a = a$ *et* $a.a = a$
Théorème 3: Elément absorbant	$a + 1 = 1$ *et* $a.0 = 0$
Théorème 4: Absorption	$a+a.b=a$ *et* $a.(a+b)=a$
Théorème 5 : Loi de Morgan	$\overline{a+b} = \bar{a}\,.\,\bar{b}$ *et* $\overline{a.b} = \bar{a} + \bar{b}$

$$a + \bar{a}b = a + b \quad et\ a.(\bar{a} + b) = a.b$$

Démonstration du théorème 3 : élément absorbant

$a + 1 = a + (a + \bar{a})$ D'après l'axiome de complémentation

$= (a + a) + \bar{a}$ D'après l'axiome de l'associativité

$= a + \bar{a}$ D'après le théorème 2

$= 1$ D'après l'axiome de complémentation

Démonstration du théorème 6 : Morgan

$a.(\bar{a} + b) = a.\bar{a} + a.b$ D'après l'axiome de la distributivité

$a.(\bar{a} + b) = 0 + a.b$ D'après l'axiome de complémentation

$a.(\bar{a} + b) = a.b$ D'après l'axiome de l'élément neutre

5. Passage d'une table de vérité en une en une fonction booléenne

À partir de la table de vérité, nous pouvons avoir deux formes analytiques, dénommées formes canoniques :
– *1ere forme canonique* : somme canonique de produits
– *2eme forme canonique* : produit canonique de sommes

5.1. Première forme canonique

- La première forme canonique est une forme ΣΠ.
- Chaque intersection contient les n variables.
- Ses intersections sont appelées « mintermes ».
- Le résultat est la somme des mintermes vrais de la fonction.

Exemple 1: Soit la table de vérité suivante. Donner la 1ere forme canonique

A	B	C	F	$P_3 + P_5 + P_6 + P_7$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

=> **F(A, B, C) = $P_3 + P_5 + P_6 + P_7$**

$$F(A,B,C) = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

5.2. Deuxième forme canonique

- La deuxième forme canonique est une forme ΠΣ.
- Chaque réunion contient les n variables.
- Ses réunions sont appelées « maxtermes ».
- Le résultat est le produit des maxtermes complémentaires des points faux de la fonction.

Exemple 2 : Soit la table de vérité suivante. Donner la 2eme forme canonique

X	Y	Z	F	$S_0 \cdot S_1 \cdot S_2 \cdot S_4$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

$$F(X, Y, Z) = S_0 \cdot S_1 \cdot S_2 \cdot S_4$$

$$F(X, Y, Z) = (X + Y + Z)(X + \overline{Y} + Z)(\overline{X} + Y + \overline{Z})(X + Y + Z)$$

6. Tableau de Karnaugh

6.1. Principe

Le tableau de Karnaugh est une représentation sous une forme particulière de la table de vérité d'une fonction logique.

Il consiste en la détermination des blocs rectangulaires de taille 2^n (16,8, 4, 2, 1) de *1 adjacents.*

Les etapes sont :

- On en déduit la fonction simplifiée associée à la table de vérité.
- On représente un tableau à 2 dimensions.
- Le passage d'une colonne à une colonne adjacente ou d'une ligne à une ligne adjacente *modifie la valeur d'une seule variable*.
- Le tableau se referme sur lui-même : la colonne la plus à gauche est voisine de la colonne la plus à droite, idem pour les lignes du haut et du bas.
- Une *case du tableau* contient une *valeur booléenne(1-0), déterminée à partir de la table de vérité* ou de la *fonction booleenne* 1 FC ou 2FC.
  Dans le cas d'une fonction en entrée 1ere FC, nous remplissons le tableau par des 1 en nous basant sur le principe des variables booléennes de la 1ere FC.
  Dans le cas d'une fonction en entrée 2éme FC, nous remplissons le tableau par des 0 en nous basant sur le principe des variables booléennes de la 2éme FC.

6.2. Regroupement en blocs rectangulaires des bits à 1 adjacents

Le regroupement doit suivre des conditions qui sont :

- Tous les 1 du tableau doivent être englobés *dans au moins un bloc* (un bloc à une taille de 16, 8, 4, 2, 1 de bits 1 adjacents). Un *bit à 1 peut appartenir à plusieurs blocs*
  On doit créer les blocs *les plus gros possibles*
- A chaque bloc correspond un terme formé comme suit :

- ✓ Pour *le bloc*, si une une variable *prend les valeurs 0 et 1, on ne la prend pas*.
- ✓ On ne *conserve que les variables qui ne varient pas*. *Si une variable a reste à 1 : on note a, si reste à 0 : on note $\bar{a}$*
- ✓ Le terme logique *du bloc correspond au ET* de ses variables qui ne changent pas.
- ✓ La fonction logique simplifiée est le *OU de tous les termes des blocs* trouvés.

Exemple3 : Tableau de Karnaugh à deux variables

Table de vérité

a	b	f(a,b)
0	0	0
0	1	1
1	0	1
1	1	1

tableau de Karnaugh

b \ a	0	1
0	0	1
1	1	1

On remplie le tableau de karnaugh en nous basant sur la table de vérité , les cas où la fonction *f (a,b)=1,*

Nous avons 2 groupes de 2 bits adjacents :

- Pour le vertical : on a toujours a = 1 => on aura le terme a.
- *Pour l'horizontal : on a toujours b =1=> on aura le terme b.*
  Donc la simplification est ***f(a,b) = a + b***

Exemple 4: Tableau de Karnaugh à 3 variables

Table de vérité

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tableau de Karnaugh

C \ AB	00	01	11	10
0			1	
1		1	1	1

F = AB + BC + AC

Exemple 5 : Tableau de Karnaugh de 4 variables à partir d'une fonction booléenne

$$F = \bar{a}\,\bar{b}\,\bar{c}\,d + \bar{a}\,\bar{b}\,c\,d + \bar{a}\,b\,\bar{c}\,d + \bar{a}\,b\,c\,d + \bar{a}\,b\,c\,\bar{d} + a\,b\,\bar{c}\,d + a\,b\,c\,d + a\,b\,c\,\bar{d} + a\,\bar{b}\,\bar{c}\,d + a\,\bar{b}\,c\,d$$

Cette fonction est donnée en 1ere forme canonique, ceci veut dire que la fonction vaut 1 dans un des chaques termes.

cd ab	00	01	11	10
00		1	1	
01		1	1	1
11		1	1	1
10		1	1	

Le 1er regroupement est de huit uns adjacents :

cd ab	00	01	11	10
00		1	1	
01		1	1	1
11		1	1	1
10		1	1	

Il reste deux 1 qui n'appartiennent à aucun groupe , il faut faire un groupe de 4 mieux que un groupe de 2.

cd ab	00	01	11	10
00		1	1	
01		1	1	1
11		1	1	1
10		1	1	

Le d ne change pas =1

Le c ne change pas =1
Le b ne change pas =1

La fonction simplifiée

F= d + bc

Nou pouvons obtenir le même resultat en nous basant sur la simplification algébrique :

$$F = \bar{a}\,\bar{b}\,\bar{c}\,d + \bar{a}\,\bar{b}\,c\,d + \bar{a}\,b\,\bar{c}\,d + \bar{a}\,b\,c\,d + \bar{a}\,b\,c\,\bar{d} + a\,b\,\bar{c}\,d + a\,b\,c\,d + a\,b\,c\,\bar{d} + a\bar{b}\,\bar{c}\,d + a\,\bar{b}\,c\,d$$

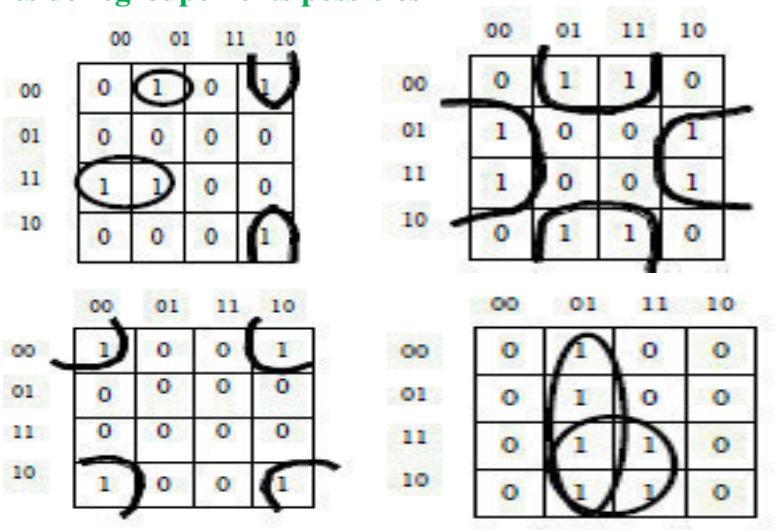
$$= \bar{a}\,\bar{b}\,d + \bar{a}\,b\,d + b\,c\,\bar{d} + a\bar{b}\,d + a\,b\,d$$

$$\bar{a}\,\bar{b}\,d + \bar{a}\,b\,d + b\,c\,\bar{d} + a\bar{b}\,d + a\,b\,d$$

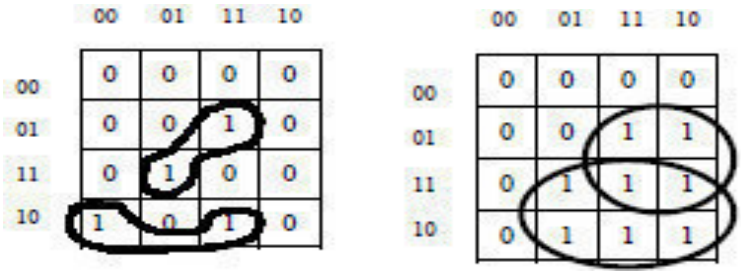
$= \bar{a}\, d + ad + b\, c\, \bar{d}$
$= \quad d \quad + b\, c\, \bar{d}$
$= (d + b\, c)(d + \bar{d})$
F= $(d + b\, c)$

Pour finir le cours voici des exemples de rgroupement de Karnaugh :

Exemples de regroupements possibles



Exemples de regroupement impossibles



Chapitre 5

Introduction à l'algorithmique

1. Définition d'un algorithme

L'algorithme est une suite d'actions appelées *instructions* dont l'exécution fournit le résultat recherché.

2. Structure générale d'un algorithme

Un algorithme se compose de :

Entête : Identifie le problème à résoudre. Il est introduit par le mot clé Algorithme nom algorithme.

Un bloc composé de :

- Un **environnement** de résolution du problème qui rassemble les déclarations d'objets non primitifs concernés par les traitements
- Un **algorithme** qui décrit les traitements.

Entête

Environnement

Algorithme

```
Algorithme nom_algorithme

Constantes
        Liste des constantes
Variables
        Liste des variables

Début
        Instruction 1
        Instruction 2
        ….
Fin
```

3. Notions algorithmiques

3.1. Identificateur

Un identificateur est un nom explicite d'une constante, d'une variable ou d'une fonction.

3.2. Variable et constante

Une variable ou une constante sert à mémoriser une valeur donnée, durant un algorithme.

- **Constante** : Une donnée manipulée par un programme et ne pouvant être modifiée.
  Syntaxe : **Constante NomConstante = Valeur**

  Exemple : Constante Pi = 3.141559
  Constante Nombrejour = 7

- **Variable :** Une donnée manipulée par un programme et pouvant être modifiée.
  Une variable peut être une donnée d'entrée, le résultat final d'un calcul, un résultat intermédiaire de calcul.

  Syntaxe : **Variable NomVariable : Type**

  Exemple *:* Variable Rayon : Reel
  Variable Compteur : Entier

3.3. Les types standards

Tout langage de programmation offre un certain nombre de ***types standards*** préalablement définis. Il existe 5 types standards :

1. **Type logique (booléen)** : Valeur pouvant être soit Vraie, soit Fausse.

Exemple : **Constante** true = Vrai
Variable B1,B2 : booleen

Opérateurs du type Booléen : la conjonction (et), la disjonction (ou) et la négation, =, <, >, ‡ …

2. **Type entier :** Valeur numérique entière pouvant être signée ou non signée (codée sur un ou plusieurs octets).

Exemple : **Constante** moins_quarante= -40
Variable E1 : entier

- Les opérations possibles sur les entiers sont :
  - Operations mathématiques : +,-,*, div (division entière ou euclidienne), mod (reste de division entière).

  - Les fonctions standards :
  sqr (n) : cette fonction fournit le carré d'un entier n,
  abs (n) : cette fonction fournit la valeur absolue d'un entier n,
  succ (n) : cette fonction fournit le successeur d'un entier n = n+1,
  pred (n) : cette fonction fournit le prédécesseur d'un entier n = n-1,

3. **Type réel** : Les valeurs numériques du type réel sont codées avec une mantisse et un exposant.

Exemple : **Constante** PI=3,14
Variable R1,R2 : réel

- Les opérations des réels sont : addition, soustraction, multiplication division.
- Exemples de fonctions mathématiques classiques :

- sin, tg, … : Les fonctions trigonométriques,
- sqr : La fonction carré,
- abs : La fonction qui renvoie la valeur absolue,
- sqrt : La fonction racine carrée.

- Les fonctions spécifiques au traitement informatiques :

- trunc (r) : Cette fonction fournit la partie entiere d'un réel.
- round (r) : cette fonction engendre l'entier le plus proche d'un réel.

4. **Type caractère** : Ce type comporte les lettres de l'alphabet, les chiffres de 0 à 9, les signes de ponctuation (. , ? ! etc), les caractères spéciaux (% @, etc), Les operateurs (+, -, *, /, > , etc).

Exemple : **Constante** six = '6'
Variable C1, C2: caractère

- Les fonctions prédéfinis sur les caractères:

- succ (c): fonction successeur qui fournit le caractère suivant du caractère c,
- pred (c): fonction prédécesseur qui fournit le précédent du caractère c.

5. **Type chaîne**: Une chaine est une suite de caractères du code Ascii.

Exemple : **Constante** Bahia = 'ORAN'
Espace= ' '
Variable Nom,prenom: chaine

- Les fonctions prédéfinis sur les chaines:

- Length (c): Cette fonction fournit la longueur de la chaine c,
- concat (c1,c2): Cette fonction fournit une chaine obtenue par concaténation de la chaine c1 et c2.

3.4. Affectation

Une affectation est une instruction qui stocke dans une variable une valeur d'une expression.

Syntaxe générale : Variable ← Expression
Cette écriture se lit : La variable reçoit l'expression

Exemple	Etat de la mémoire
A, B : entier	
A ← 5	A 
B ← A -2	B 
A ← B-A	A  (La valeur 5 est écrasée)

4. Les fonctions de lecture / Ecriture

L'algorithme a besoin de données en entrée, et fournit un résultat en sortie. Lorsqu'on utilise un ordinateur, le clavier permet de saisir les données et l'écran d'afficher un résultat ou des textes qui donnent des directives sur les données à fournir. Lorsqu'on voudra afficher un texte sur l'écran, on utilisera une fonction nommée **Ecrire** permettant d'affiche à l'écran les arguments qu'on lui demande d'afficher. Nous utiliserons également une autre fonction nommée **Lire** qui permet de stocker la ou les données saisies au clavier dans des variables.

4.1. Écriture des données

Trois cas peuvent être rencontrés :

- Afficher un texte : ***Ecrire ('texte à afficher')***
- Afficher la valeur d'une variable : ***Ecrire (nom_de_la_variable)***
- Mélange de texte et de valeurs : ***Ecrire ('texte ', nom_de_la_variable, ' texte', nom_de_la_variable)***

Remarques :
La virgule sépare les chaînes de caractères et la variable.
Tout le texte contenu entre des guillemets est écrit à l'écran, alors que lorsqu'une variable apparaît dans l'instruction Ecrire c'est sa valeur qui est affichée.

Exemple :

Algorithme

Ecrire ('nombre ? ');
nb ← 10;
Ecrire (nb);
Ecrire ('nb vaut ', nb, '. ');

Affichage

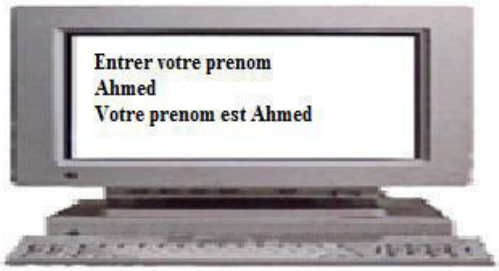
4.2. Lecture des données

Syntaxe: lire (nom de variable)

Cela va nécessiter l'utilisation d'un clavier, la valeur saisie au clavier va être enregistrée dans variable.
Attention une constante n'est **jamais** lue.
Exemple

Algorithme saisi
variable nom : chaine
Debut
Ecrire (' entrer votre nom de famille ')
Lire (nom)
Ecrire (' Votre nom est' , nom)
Fin

Exécution



Etat de la Mémoire

nom	
nom	Ahmed

Exercices faits en cours

1. Ecrire un algorithme qui calcule et affiche la somme de deux entiers
2. Ecrire un algithme qui affiche le carré d'un nombre réel
3. Ecrire un algorithme qui lit une chaine de caractéres et affiche sa longueur

5. Les structures alternatives

La sélection exprime un enchainement conditionnel (en fonction d'une condition, le programme exécute des opérations différentes)
On distingue l'alternative réduite, complète et imbriquée.

5.1. Alternative réduite

Si la condition est **VRAI**, le bloc d'instructions est exécuté.

Si condition **Alors**
Bloc d'instructions
Finsi

Algo 1 : Ecrire un algorithme qui calcule la racine d'un nombre

```
Algorithme racine
Variable x : entier
Debut
écrire ("Saisir le nombre x")
lire (x)
Si (x > 0) Alors  r ←sqrt (x)
                  écrire  ("la racine de", x, " est", r)
FinSi
Fin
```

5.2. Alternative complète

Si la condition est **VRAI**, le bloc 1 d'instructions est exécuté.
Si la condition est **FAUSSE**, le bloc 2 d'instruction est exécuté.

Si condition **Alors**
Bloc 1
Sinon
Bloc 2
Finsi

Algo 2 : Ecrire un algorithme qui calcule la racine d'un nombre et affiche erreur un message si ya erreur

```
Algorithme racine2
Variable x : entier
Debut
écrire ("Saisir le nombre x")
lire (x)
Si (x > 0) Alors
                r ← sqrt (x)
                écrire  ("la racine de", x, " est", r)
            Sinon
                écrire  ("Erreur, veuillez saisir un nombre
                         positif")
FinSi
Fin
```

Algo 3 : Ecrire un algorithme qui affiche la valeur absolue d'un nombre

```
Algorithme absolue
Variable n: entier
Debut
écrire ("Saisir le nombre ")
lire (n)
Si (n >= 0) Alors
             écrire  ("la valeur absolue de", n, " est", n)
            Sinon
             écrire  ("la valeur absolue de", n, " est", -n)
FinSi
Fin
```

5.3. Alternative imbriquée

Plusieurs alternatives peuvent être imbriquées, il s'agit d'un choix de multiple.

Si condition1 **Alors**
bloc 1
Sinon Si condition2 **Alors**
bloc 2
Sinon Si condition3 **Alors**
bloc 3
Sinon
bloc 4
Finsi
Finsi
Finsi

Algo4: Afficher l'état de l'eau (glace, eau, vapeur) suivant les différentes températures.

```
Algorithme température
Variable Temp: Entier
Début
Ecrire ("Entrez la température de l'eau :")
Lire(Temp)
Si (Temp <=0) Alors  Ecrire ("C'est de la glace")
              Sinon
              Si (Temp < 100) Alors Ecrire ("C'est du liquide")
                              Sinon  Ecrire ("C'est de la
                                     vapeur")
              Finsi
Finsi
Fin
```

6. Les structures itératives

On appelle itération, toute répétition de l'exécution d'un traitement.
A la notion d'itération est associée la notion de boucle.
Le nombre d'itération doit être fini : soit par une condition, soit par un compteur.
Il existe trois types de structures d'itérations (boucles) : *répéter, tant que, pour.*

6.1. Structure REPETER ... JUSQUA ...

Une action ou un groupe d'actions est exécuté répétitivement jusqu'à ce qu'une *condition soit vérifiée.* La condition est formulée par une expression booléenne.

Syntaxe
Répéter
bloc d'instruction
Jusqu'a Condition vrai

Remarque : la vérification de la condition s'effectue après les actions. Donc le bloc est exécuté au moins une fois.

6.2. Structure TANT QUE ... FAIRE ...

Le bloc d'instructions est exécuté répétitivement *tout le temps où une condition est vraie*.

Syntaxe
TantQue Condition vrai **Faire**
Bloc d'instructions
FinFaire

Remarque : la vérification de la condition s'effectue avant les actions. Donc le bloc peut ne jamais être exécuté (minimim 0 fois).

6.3. Structure POUR Indice DE ... A .... FAIRE ...

Le schéma pour utilise une variable appelée *compteur d'itération* afin de contrôler le nombre de répétitions.

Syntaxe

Pour i de Val1 **a** Val2 **pas** Val3 **Faire**
bloc d'instructions
FinFaire

Avec : i : variable compteur qui doit absolument être de type entier
Val1 : valeur initiale
Val2 : valeur finale
Val3 : le pas

Remarque Les valeurs initiale (Val1) et finale (Val2) sont incluses dans le comptage.
Par défaut le pas d'incrémentation est à 1, mais il est éventuellement possible de spécifier un autre pas d'incrémentation (+2,+10,-1, -2....)

Algo 5: Ecrire un algorithme qui demande un nombre de départ n, et qui calcule la somme des entiers jusqu'à ce nombre. On souhaite afficher uniquement le résultat final.
(Exemple, si l'on entre n = 5, le programme doit calculer:1+ 2+ 3+4+5 =15
Faut vérifier que n saisi est positif

Solution avec la boucle POUR

```
Algorithme Sommation
Variables N, i, Som: Entier
 Debut

Répéter
 Ecrire( "Entrez un nombre : ")
 Lire (N)
Jusqu'à (n>0)

Som ← 0

Pour i de 1 à N faire
 Som ← Som + i
fin pour

Ecrire( "La somme est : ", Som)
Fin
```

Cette boucle vérifie que le N saisi est positif ; Donc A chaque fois où on entre un nombre négatif, il n'est pas accepté et on doit re saisir un autre. On sort de la boucle dès qu'on saisi un nombre positif

Il faut absolument initialiser la Som parce que dans la boucle vous avez ancienne et nouvelle valeur de som

Pour chaque i allant de 1 à N, on calcule la somme Som.
Puisque le pas =1, on ne l'écrit pas.

Solution avec la boucle TANT QUE

```
Algorithme summation 2
Variables N, i, Som: Entier
 Debut
 Répéter
 Ecrire( "Entrez un nombre : ")
 Lire (N)
Jusqu'à (n>0)

 Som ← 0

i ←1
Tant que i<=N faire
 Som ← Som + i
i  ← i+1
fin faire
Ecrire( "La somme est : ", Som)
Fin
```

Solution avec la boucle REPETER

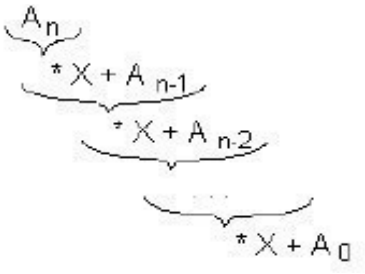
```
Algorithme summation 3
Variables N, i, Som: Entier
 Debut
 Répéter
 Ecrire( "Entrez un nombre :
")
 Lire (N)
Jusqu'à (n>0)


 Som ← 0
i ←1
repeter
 Som ← Som + i
i  ← i+1
jusqu'a (i>N)
fin pour
Ecrire( "La somme est : ",
Som)
Fin
```

Exercices faits en cours

1. Ecrire un algorithme qui calcule le factoriel d'un nombre entier N
2. Ecrire un algorithme qui calcule la puissance X^n
3. Ecrire un algorithme qui calcule pour une valeur X réelle la valeur numérique d'un polynôme de degré n:

$P(X) = A_nX_n + A_{n-1}X_{n-1} + ... + A_1X + A_0$

La valeur de n et X et les coefficients A_n, ... , A_0 et de X seront entrées au clavier.
Utilisez le *schéma de Horner*



7. Les tableaux statiques

7.1. Définition

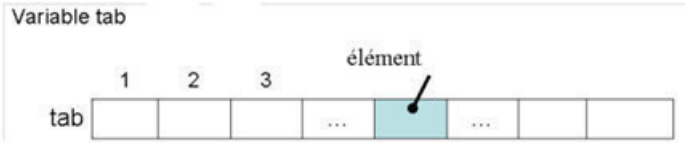
Un tableau unidimensionnel ou tableau linéaire est une variable indicée permettant de stocker n valeurs de même type.
Le nombre maximal d'éléments précisé à la déclaration, s'appelle la dimension (ou capacité) du tableau.
Le type du tableau est le type de ses éléments.
La position d'un élément s'appelle ***indice*** ou ***rang*** de l'élément.
Puisque la dimension est précisée dans le code, nous parlerons d'un tableau statique.

7.2. Déclaration d'un tableau statique

Syntaxe

Variable nomtableau : Tableau [1..dimension] de type

Exemple : Déclaration d'un tableau entier de 8 cases :
Variable Tab : Tableau [1..8] de entier



7.3. Saisi et affichage d'un tableau (Lecture et écriture)

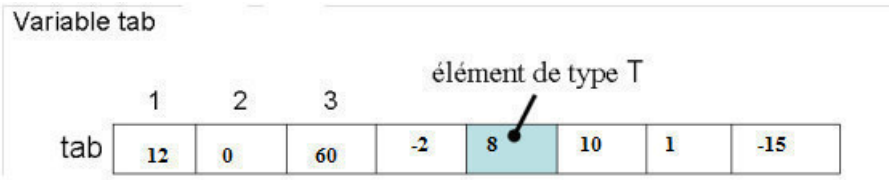
Pour pouvoir remplir (ou afficher) un tableau, il faudra saisir (ou afficher) des valeurs dans les cases, ceci nécessitera l'utilisation d'une boucle pour parcourir le tableau.

7.3.1. Lecture d'un tableau

Syntaxe

```
Pour i de 1 à dimension faire
Ecrire ("donnez la valeur de la case [",i, "]")
Lire (Tab[i])
Finfaire
```

Nous aurons donc après exécution



7.3.2. Ecriture ou affichage d'un tableau

Syntaxe

```
Pour i de 1 à dimension faire
Ecrire ("la case",i, "contient la valeur ", Tab[i])
 Finfaire
```

Nous obtenons donc après déroulement cet affichage à l'écran :
La case 1 contient la valeur 12
La case 2 contient la valeur 0
La case 3 contient la valeur 60
La case 4 contient la valeur -2
La case 5 contient la valeur 8
La case 6 contient la valeur 10
La case 7 contient la valeur 11
La case 8 contient la valeur -15

Exercice

Soit un tableau de n cases entières, écrire un algorithme qui somme les valeurs positives et négatives de ce tableau.

Solution
Etapes

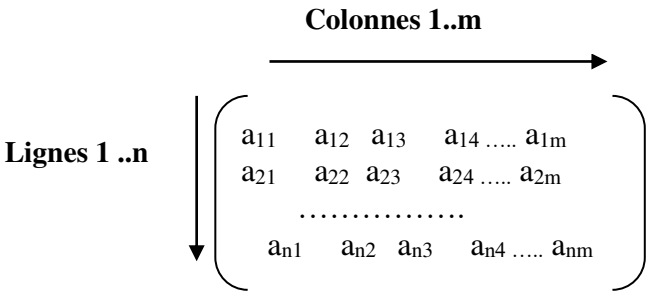
- Déclarer un tableau avec une taille maximale prédéfinie par exemple 50 cases.
- Entrer le nombre de cases réel n de notre tableau .
- Saisir les n valeurs du tableau tout en testant le signe

```
Algorithme sommation
Variable Tab :tableau [1..50] de entier
                    i,n,somP,somN :entier

Debut
Ecrire ("donnez la dimension réelle du tableau")
Lire (n)
SomP ←0
SomN←0
Pour i de 1 à n faire
   Ecrire ("donnez la valeur de la case[",i, "]")
   Lire (Tab[i])
     Si (Tab[i]>=0) alors SomP ←SomP+Tab[i]
                      Sinon SomN ←SomN+Tab[i]
Finfaire
Ecrire ("La somme des valeurs positives est ", SomP, "et laa somme
des valeurs negatives positive est ", SomN )
Fin
```

8. Matrices statiques

8.1. Définition

Les matrices sont des tableaux à deux dimensions.



8.2. Déclaration d'une matrice

Syntaxe
On déclare une matrice à deux dimensions de la façon suivante :

Variable nomVariable : **Tableau[***Dimension 1***][***Dimension2* **]** de *type*

Exemple : Déclarer une matrice entière de 2 lignes e 3 colonnes
Variable Mat : Tableau [1..2][1..3] de entier

Nous avons donc déclaré un matrice entiére de 2 lignes et 3 colonnes

8.3. Saisi et affichage d'un tableau (Lecture et écriture)

On accède (en lecture ou en écriture) à la i ème ligne et la j ème colonne de la matrice, ceci nécessitera l'utilisation d'une boucle pour les lignes et une autre boucle pour les colonnes.

8.3.1. Lecture d'une matrice

Syntaxe

```
Pour i de 1 à dimension1  faire
Pour j de 1 à dimension2  faire
    Ecrire ("donnez la valeur de la case [",i, "][",j, "]")
    Lire (M[i][j])
Finfaire
Finfaire
```

Exemple

```
Pour i de 1 à 2  faire
Pour j de 1 à 3  faire
  Ecrire ("donnez la valeur de la case [",i, "][",j, "]")
  Lire (M[i][j])
Finfaire
Finfaire
```

Nous aurons donc après exécution

Pour i=1 (1ere ligne) et variation de j (les colonnes)

M[1][1]=15
M [1][2]=2
M [1][3]=0

Pour i=2 (2eme ligne) et variation de j (les colonnes)

M [2][1]=-2
M [2][2]=13
M [2][3]=-20

$$\begin{pmatrix} 15 & 2 & 0 \\ -2 & 13 & -20 \end{pmatrix}$$

8.3.2. Ecriture de la matrice

Syntaxe

```
Pour i de 1 à dimension1  faire
Pour j de 1 à dimension2  faire
Ecrire ("la ligne",i, "et la colonne ",j, "contient la valeur",
M[i][j])
Finfaire
Finfaire
```

Nous aurons donc après execution pour 2lignes et 3 colonnes

La ligne 1 et la colonne 1 contient la valeur 15
La ligne 1 et la colonne 2 contient la valeur 2
La ligne 1 et la colonne 3 contient la valeur 0
La ligne 2 et la colonne 1contient la valeur -2
La ligne 2 et la colonne 2 contient la valeur 13
La ligne 2 et la colonne 3 contient la valeur -20

Exercice

Soit une matrice réelle, écrire un programme qui permet d'extraire la valeur maximale de chaque ligne et stocke le résultat dans un tableau.

Solution

Etapes :

-Déclarer une matrice réelle avec par exemple de 20 lignes et 30 colonnes**.**
-Déclarer un tableau réel de 20 lignes
- donner le nombre de lignes et le nombre de colonnes réel (n, m).
- Saisir toutes les valeurs de la matrice de n lignes et m colonnes.
- Pour chaque ligne i de la matrice, supposer que le premier élément M[i][1] est ma maximal.
- Parcourir les colonnes j+1 à m et tester les valeurs.
- Avant de passer à la ligne suivante, enregistrer dans le tableau résultat la valeur maximale.
- Refaire le procédé jusqu'à la fin des lignes.
- Afficher le tableau résultat.

```
Algorithme maximum
Type Mat =tableau [20][30] de reel
         Tab=tableau [20] de reel

Variable
M  :Mat
T :Tab
i,n,m :Entier
max :reel

Debut
Ecrire (“donnez le nombre de lignes et le nombre de colonnes de
votre matrice“)
Lire (n,m)

Pour i de1 à n faire
    Pour j de 1 à m faire
       Ecrire (“donnez la valeur de la case [“,i, “][“,j, “]“)
```

```
       Lire (M[i][j])
   Finfaire
Finfaire

Pour i de1 à n faire
     Max← M [i][1]
     Pour j de 2 à m faire
        Si (M [i][j]>=Max) alors Max← M [i][j]
        fsi
      Finpour
T[i] ←Max
Finpour

Pour i de1 à n faire
Ecrire ("la case",i, "contient la valeur ", T[i])
Finfaire
Fin.
```

TD 1
Architecture de l'ordinateur

I. Généralités

- C'est quoi une carte mère ?
- C'est quoi le programme BIOS et à quoi sert la pile située sur la carte mère ?
- Quelle est la signification des acronymes CPU, UAL, RAM, ROM ?
- Donner une brève définition du CPU, UAL, Socket, Chipset, RAM, ROM ?

II. QCM

Choisissez-la ou les bonnes réponses :

1. Il est possible de démarrer un pc sans la carte mère
   a. Vrai
   b. Faux

2. L'ordinateur nécessite
   a. La mémoire morte pour démarrer
   b. L'ordinateur nécessite la mémoire vive pour démarrer
   c. Aucune bonne réponse

3. L'horloge système permet de :
   a. Synchroniser les différentes opérations de base
   b. Afficher la date est l'heure
   c. Faire circuler les informations

4. Le microprocesseur comprend
   a. L'UAL ou l'UCC
   b. Unité arithmétique et logique
   c. Unité de commande et de contrôle
   d. L'UAL et l'UCC
   e. Aucune bonne réponse

5. L'unité de commande et de contrôle
   a. Produit des ordres
   b. Réalise des instructions de lecture
   c. Réalise des instructions arithmétiques et logiques

6. La ROM et la RAM représentent des mémoires
   a. Secondaires
   b. Principales
   c. De stockage

7. Le Disque dur, CDROM, flash disque représentent des mémoires
   a. Auxiliaires
   b. Secondaires
   c. De masse

8. Le compteur ordinal stocke
   a. L'instruction en cours exécution
   b. L'exécution en cours d'instruction
   c. L'adresse de l'instruction en cours exécution
   d. Aucune bonne réponse

9. L'accumulateur stocke
   a. L'opérande et résultat de l'instruction en cours exécution

b. L'adresse de l'instruction en cours exécution
c. L'instruction en cours d'exécution

10. Le registre d'instruction stocke
a. Toutes les instructions du programme
b. L'adresse et l'instruction en cours d'exécution
c. L'instruction en cours d'exécution

11. La RAM est le lieu
a. De stockage des programmes
b. D'exécution des programmes

III. Mémoire et registres

- C'est quoi le débit d'une mémoire centrale ?
- Quelles sont les opérations que la mémoire peut effectuer et quelle est la différence entre elles ?
- En se basant sur un tableau, donnez 4 différences entre la RAM, la ROM, et les mémoires de masse.
- Quelle est la différence entre la SRAM et la DRAM ?
- Quels sont les liens physiques entre le CPU et la mémoire ?

Corrigé

I. Généralité

1. Une **carte mère** est une plaque métallique composée de circuits et de ports de connexion permettant la liaison de tous les composants et périphériques (disque dur, mémoires vives, CPU, cartes d'extension…) afin qu'ils puissent être reconnus et par le CPU grâce au programme BIOS.

2. Le programme **BIOS** (Basic input output system) est petit programme situé sur la carte mère dans une puce de type ROM. Le bios est le premier programme chargé en mémoire dés que le pc est mis en tension.

Le bios est stocké dans une mémoire de lecture seule, les modifications que l'user peut apporter comme effectuer un mot de passe au niveau du bios ou régler la date ou… sont enregistrés dans une mémoire volatile appelé la CMOS. Puisqu'elle est volatile, donc il lui faut une alimentation, quand le pc est mis en tension, le problème ne se pose pas mais quand le pc est éteint c'est la pile qui prend le relai. Donc de manière générale, la pile maintient le bios en tension.

3. Acronymes
CPU : Central Processing Unit
UAL : Unité Arithmétique et Logique
RAM : Random Access Memory,
ROM : Read Only Memory

4. Brèves définitions de :

- **CPU** : Le CPU (***C**entral **P**rocessing **U**nit)* est le cerveau de l'ordinateur. Il permet de manipuler et de traiter les données qui lui sont fournies.
- **UAL** : Un circuit complexe assurant les fonctions logiques (ET, OU, Comparaison, Décalage, etc…) ou arithmétiques (Addition, soustraction).
- **Socket** : est l'emplacement du processeur, nous trouvons deux types de sockets ZIF (Zero insertion force) et LIF (Low insertion force).
- **Chipset** : Le chipset (jeu de composants) est une puce électronique chargée de coordonner les échanges de données entre les différents composants de l'ordinateur : le processeur, la mémoire vive, etc. En d'autres termes, c'est la plateforme centrale de la carte mère. Nous trouvons le :
  - **North bridge** (pont nord) est la partie la proche du cpu, il contrôle les éléments les plus rapides du pc en gérant les échanges avec la mémoire, le bus pc express (les anciens bus :AGP pour la carte graphique et pc pour les autres cartes d'extension).
  - **Le south bridge** : Il est relié au CPU à travers le north bridge, il gère les entrées sorties (contrôleur Pata, Sata, port USB…).

II. QCM

1. Il est possible de démarrer un pc sans la carte mère **Faux**
2. L'ordinateur nécessite **a,b**
3. L'horloge système permet de **a**
4. Le microprocesseur comprend **b,c,d**
5. L'unité de commande et de contrôle **a**
6. La ROM et la RAM représentent des mémoires **b**
7. Le Disque dur, CDROM, Flash disque représentent des mémoires **a,b,c**
8. Le compteur ordinal stocke **c**
9. L'accumulateur stocke **a**
10. Le registre d'instruction stocke **c**
11. La RAM est **le lieu** **b**

III. Mémoire et registres

1. **Le débit** représente le nombre d'informations (exprimé en bits) lues ou écrites par seconde.
2. **Les opérations** possibles en MC sont la lecture et l'écriture.
3. **Définition** de RAM et ROM

RAM est une mémoire stockant les programmes et les données en cours d'exécution donc elle est directement accessible par le processeur.
ROM est une mémoire dont le contenu ne peut être modifié en usage normal.

4. Tableau comparatif entre RAM et ROM et mémoire de masse

	RAM	**ROM**	**Masse**
Appellation	Vivante	morte	Masse
Volatilité	Oui	Non	Non
Lecture/écriture	Oui	Lecture seul	Oui
Contenu	Programme et données en cours d'exécution	Programme de base	Les programmes et données (ex disque dur, CD…)
Taille	En GO	En Ko	Tera

5. La SRAM est basée sur des transistors et la DRAM sur des condensateurs ce qui nécessite leur rafraichissement.

6. Les 3 liens physiques sont bus de données, bus d'adresse, bus de systéme.

Fiche TD 2
Représentation des nombres et algèbre de BOOLE

PARTIE 1 : Représentation des nombres

Exercice 1 : Conversion de base en base

Convertir ces nombres dans leurs bases appropriées

$(10011001)_2 = (\quad)_{10}$
$(10011110)_2 = (\quad)_8$
$(12,6875)_{10} = (\quad)_2$
$(BAFFE)_{16} = (\quad)_2$

Exercice 2 : Conversions et opérations en complément à 2

1. Soit une machine à 8 bits, remplissez le tableau :

Décimal	Binaire pur	Binaire signé	Ca1	Ca2
+71				
-51				
+1				
-10				
-560				
+127				

2. Effectuer en complément à 2, les opérations suivantes :

$(+71)_{10} + (-51)_{10} = (\quad)_{ca2} = (\quad)_{ca1} = (\quad)_{2=} = (\quad)_{10}$
$(+1)_{10} + (+127)_{10} = (\quad)_{ca2} = (\quad)_{ca1} = (\quad)_{2=} = (\quad)_{10}$
$(+127)_{10} + (-10)_{10} = (\quad)_{ca2} = (\quad)_{ca1} = (\quad)_{2=} = (\quad)_{10}$
$(-51)_{10} + (+1)_{10} = (\quad)_{ca2} = (\quad)_{ca1} = (\quad)_{2=} = (\quad)_{10}$

Exercice 3 : La norme IEEE 754

1. Donnez la conversion des nombres $(-35,50)_{10}$ et $(-123.75)_{10}$ en format IEEE 754 simple précision puis convertir les résultats sous forme octale et hexadécimale.
2. Quelle est la valeur décimale de la représentation IEEE 754 suivante? (ES 1)

0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

PARTIE 2 : ALGEBRE DE BOOLE

Exercice 1: Simplifications Algébriques

A	B	C	$F(A,B,C)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

1. Démonter algébriquement que

$$AB + \overline{B}\,C = (A + \overline{B})(B + C)$$

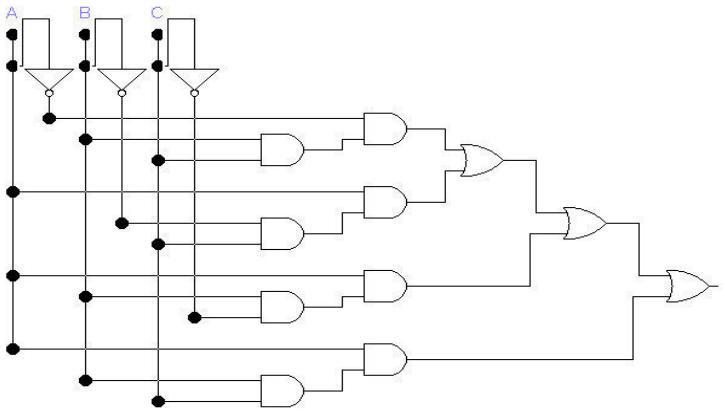
$$\overline{(A + B)(\overline{A} + C)} = (A + \overline{B})(\overline{A} + \overline{C})$$

2. Considérer la fonction définie par la table de vérité :

- Générer une expression logique correspondante sous forme de première forme canonique et deuxième forme canonique.
- Simplifier les deux fonctions trouvées algébriquement.

Exercice 3 : Karnauguh (ES

1. Donner l'expression équiv ce circuit.
2. Donner l'expression simpl utilisant le tableau de Kar



Exercice 4 : Problème de contrôle qualité

Un contrôle de qualité est effectué sur des briques dans une usine, chaque brique possède quatre critères de qualités : son poids P, son épaisseur e, sa longueur L, sa largeur l.

Ces quatre grandeurs sont mesurées sur chaque brique. Elles sont classées en trois catégories :

- ***Qualité A*** : Le poids et deux dimensions au moins sont corrects.
- ***Qualité B*** : Le poids est incorrect et les autres dimensions correctes ou le poids correct et au moins deux dimensions incorrectes.
- ***Qualité C*** : Le poids P est incorrect ainsi qu'une ou plusieurs dimensions.

- Extraire la table de vérité.
- Simplifier les fonctions A, B, C avec KARNAUGH.
- Tracer le logigramme des fonctions simplifiées avec :
  - ✓ Les portes ET, OU et les inverseurs.
  - ✓ Les portes NON ET et les inverseurs

Exercices supplémentaires

Exercice 1 : Soient les deux nombres suivants, codés selon la norme IEEE 754 simple précision et représentés en hexadécimal : $(3EE00000)_{16}$ et $(3D800000)_{16}$

Calculez la somme de ces deux nombres et donnez le résultat sous forme IEEE 754 simple précision et sous forme décimale.

Exercice 2 : (ES1 2016- 2017)

Les conditions d'inscription au concours n° 2 sont les suivantes :

- Avoir été inscrit au concours n° 1, être du sexe masculin et marié, ou bien,
- N'avoir pas été inscrit au concours n° 1, être du sexe féminin et mariée, ou bien,
- Avoir été inscrit au concours n° 1, être marié et âgé de moins de 25 ans, ou bien,
- Être marié et avoir plus de 25 ans, ou bien,
- Être du sexe masculin et âgé de moins de 25 ans.

1. Exprimez sous forme d'une expression logique F les conditions d'inscription au concours n°2.
2. Donnez la table de vérité correspondante.
3. Simplifiez l'expression logique par le tableau de karnaugh**.**

TD 3
Algorithmique

PARTIE 1. Les structures séquentielles

Exercice 1

Dérouler les deux algorithmes	Citer et corriger les erreurs commises dans les 2 algorithmes suivants :
Algorithme affecter1 **Variables** A,B,C : entier **Début** A ←6 B ←12 C ←B+2 A←A-2 B←C*2 **Fin** **Algorithme** affecter 2 **Variables** A,B,C : entier **Début** A ←7 B ←A+1 C ←B/2 C←C-2 A←B **Fin**	**Algorithme** erreur1 **Constante** R=3 **Variables** X : entier **Début** Lire (R) X ←1 Y ←R-X Ecrire ("Y= Y ") **Fin** **Algorithme** erreur2 **Variables** X : chaine de caractère Y :entier **Début** X ← "Annee" Y ← 2018 Ecrire ("La concatenation obtenue est concat (X,Y)" **Fin**

Exercice 2

- Ecrire un algorithme qui calcule et affiche la surface (S) et le périmètre (P) d'un rectangle.
- Ecrire un algorithme qui affiche la surface (S) et le périmètre (P) d'un rectangle
- Quel est l'algorithme le plus optimal et pourquoi ?

Exercice 3 : Soit l'algorithme de permutation

```
Algorithme permuter
Variable A, B : entier
Début
Ecrire("donnez deux valeurs entières")
Lire (A, B)
 A ← B
 B ← A
Ecrire ("Les deux valeurs après permutation sont", A, B)
 Fin
```

1. Quelles sont les valeurs finales de A, B ?
2. Inversez l'ordre des instructions 3 et 4 et redéroulez ? Que concluez- vous?
3. Que doit- on ajouter pour effectuer la permutation ?
4. Proposer un algorithme en ce sens.

PARTIE 2. Les structures conditionnelles

Exercice 1 : Donnez l'algorithme qui permet de calculer U tel que :

$$U = \begin{cases} \frac{A^3}{2} + 1 & \text{Si } A \geq 0 \\ A^2 & \text{Sinon} \end{cases}$$

Exercice 2 : Déroulez l'algorithme suivant étape par étape

```
Algorithme Affecter
variable X,Y,Z: entier
debut
X ← 1
Y ← 3
Z ← 0
Si (X > Y) Alors   X ← 5
              Sinon
               Si (X > 0) Alors   X ← - 1
                          Sinon   X ← 4
              finSi
finSi
Si (X > Z) Alors     X ← 2
             Sinon     X ← 3
finSi ;
Ecrire ('X=',X, 'Y=' ,Y, 'Z=',Z)
Fin.
```

Exercice 3 : Ecrire un algorithme qui affiche les jours d'un mois donné de l'année 2018 (ES1 2017).

PARTIE 3. Les structures itératives

Exercice 1: Ecrire un algorithme qui permet de calculer la somme (utiliser les 3 boucles):

$$\mathbf{S= -1+ \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \frac{1}{5!} + \frac{1}{6!} - \frac{1}{7!} + \frac{1}{8!} - \ \ldots \frac{1}{n!}}$$

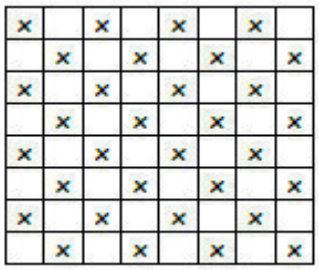
Exercice 2 : Ecrire un algorithme qui permet de calculer la moyenne d'une suite des nombres positifs ou nuls. La fin de cette suite est déterminée par la lecture d'un nombre négatif.

Exercice 3: Calculez par des soustractions successives le quotient entier et le reste de la division entière de deux entiers entrés au clavier.

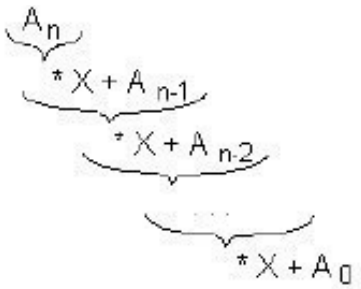
Exercices supplémentaires

Exercice 1 : Ecrire un algorithme qui affiche le carré de 5 nombres paires saisis au clavier (Vérifier d'abord si le nombre est paire).

Exercice 2 : Écrire un algorithme permettant d'écrire un échiquier de 8 fois 8. On représentera les case noires par des 'x' et les cases blanches par des espaces.



Exercice 3 : Ecrire un algorithme permettant de calculer pour une valeur X réelle la valeur numérique d'un polynôme de degré n: $P(X) = A_nX_n + A_{n-1}X_{n-1} + ... + A_1X + A_0$
Les valeurs de n, des coefficients A_n, ... , A0 et de X seront entrées au clavier.
Utilisez le schéma de Horner



PARTIE 4. Tableaux et matrices

Exercice 1. Ecrire un algorithme qui :
- Lit la dimension N d'un tableau réel,
- Saisi les valeurs du tableau,
- Affiche le tableau ainsi que la somme de tous ses éléments.

Exercice 2. Ecrire un algorithme permettant d'ordonner un tableau d'entiers de manière croissante.

Exercice 3. Ecrire un algorithme qui:
- Demande la dimension N d'un tableau d'entiers,
- Rempli un tableau de N entiers,
- Saisi une position d'un tableau,
- Supprime la valeur relative à cette position (ceci implique le décalage des valeurs),
- Affiche le tableau après suppression.

Exercice 4. Ecrire un algorithme qui calcule la somme des éléments de la première diagonale d'une matrice entière.

Exercice 5. Soit une matrice entière de N lignes, M colonnes. Ecrire un algorithme qui permet d'extraire le minimum de chaque ligne de la matrice et enregistre les résultats dans un tableau de N lignes.

Exercices supplémentaires

Exercice 1. Ecrire un algorithme qui :
- Demande la dimension N d'un tableau d'entier *tab* avec le test,
- Rempli un tableau de N entiers,
- Met dans un tableau *tab_P* les valeurs positives du *tab,*
- Met dans un tableau *tab_N* les valeurs négatives ou nulles du *tab,*
- Affiche les 2 tableaux résultats *tab_P* et *tab_N*.

Exercice 2. Ecrire un algorithme qui calcule la somme des éléments de la deuxième diagonale d'une matrice entière.

Exercice 3. Soit un tableau réel de N lignes. Ecrire un algorithme qui extrait la valeur minimale et maximale avec leurs positions respectives.

Exercice 4. Ecrire un algorithme qui fait la multiplication entre deux matrices entières M1 et M2.

Corrigé

Partie 2- Exercice 2

```
Algorithme calcul
Variable
A : entier
U :réel
Debut
Ecrire ('Donner un nombre')
Lire (A)
Si (A>=0) alors U←(A*A*A)/2 +1
          Sinon U←A*A
Finsi
Ecrire (U)
Fin.
```

Partie 3. Exercice 2

```
Algorithme moyenne
Variable S,X reels
Début
S←0
Ecrire ('Donner un nombre')
Lire (X)
Tant que (X≥0) faire
  S←S+X ;
  Lire (X) ;
Finfaire
Ecrire (S)
FIN.
```

Partie 4. Exercice 4

```
Algorithme diagonale
Variable
Mat=tableau [1..20][1..20] entier
S,i,j,n : entier
Debut
Ecrire ('Donner le nombre de ligne de votre matrice')
Lire (n)
Pour i =1 à n faire
Pour j =1 à n faire
Ecrire ('Donner une valeur de la ligne',i,'colonne ',j)
Lire (Mat[i][j])
Fpour
Fpour
S←0
Pour i =1 à n faire
S←S+Mat[i][i]
```

```
Fpour
Ecrire ('la somme de la 1ere diagonale est',S)
Fin
```

INFORMATIQUE

2

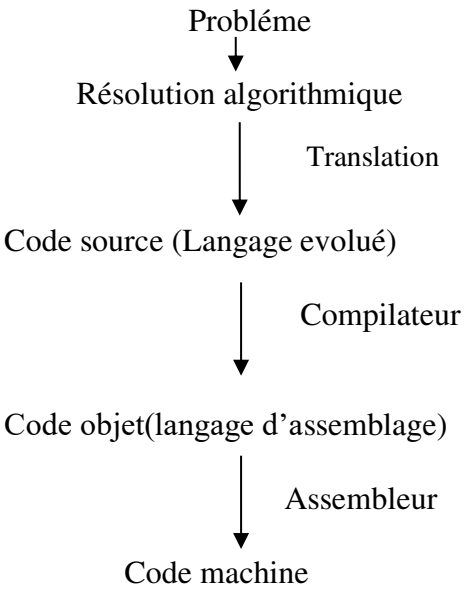
Chapitre 1

De l'algorithmique à la programmation

1.Introduction

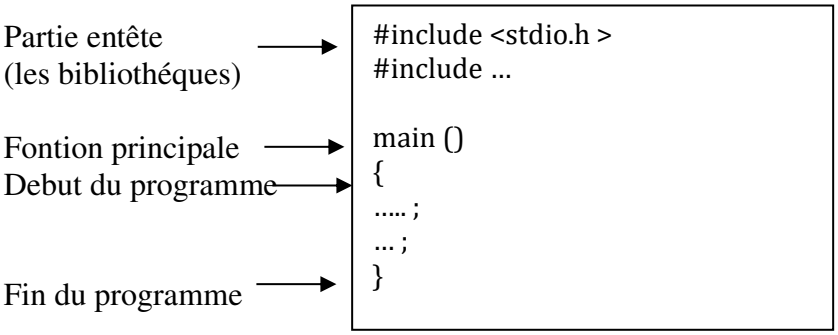
Les langages de programmation permettent aux utilisateurs de créer des instructions permettant à un ordinateur d'effectuer des tâches.
Il existe trois catégories de langages de programmation : les langages de programmation de haut niveau, le langage d'assemblage et le langage de machine.

- *Les langages de programmation de haut niveau* sont plus faciles à comprendre pour les humains.
- *Le langage d'assemblage* est une langue intermédiaire entre la langue de haut niveau et la langue de la machine.
  La différence clé entre le langage machine et le langage d'assemblage est que**,** le langage machine s'exécute directement par un ordinateur et le langage assembleur nécessite la conversion d'un assembleur en code machine ou en code objet à exécuter par la CPU.



Le langage C fait partie de la famille des langages de programmation fonctionnelle de haut Niveau. Il a été conçu pour l'écriture de systèmes, en particulier le système unix. Pour cette raison, ses concepteurs ont fait une séparation entre ce qui est purement algorithmique (déclarations, instructions, etc.) et tout ce qui est interaction avec le système (entrées sorties, allocation de mémoire, etc.) qui est réalisé par appel de fonctions se trouvant dans une bibliothèque dite bibliothèque standard.

2. Struture générale d'un programme C



3. Les bibliothèques de fonctions prédéfinies

La pratique en C exige l'utilisation de bibliothèques de fonctions. Ces bibliothèques sont disponibles dans leur forme précompilée (extension: .lib). Pour pouvoir les utiliser, il faut inclure des fichiers en-tête (*header files* - extension .h) dans nos programmes. Ces fichiers contiennent des *'prototypes'* des fonctions définies dans les bibliothèques et créent un lien entre les fonctions précompilées et nos programmes.

#include

L'instruction ***#include*** insère les fichiers en-tête indiqués comme arguments dans le texte du programme au moment de la compilation.

Par exemple :la bibliothéque #include <stdio.h> permet l'utilisation des fonctions d'entrée sorties.

4. Etapes de génération de fichier exécutable

L'objectif d'un programmeur est bien sur d'arriver à générer (puis exécuter) un fichier exécutable.

Ceci passe par plusieurs étapes :

- La première étape consiste à écrire le programme dans un fichier texte à l'aide d'un éditeur de texte. on parle de fichier source (ayant *l'extension .c*). Ce programme est compréhensible par la machine.

- La deuxième étape est l'étape de pré-compilation. Elle consiste à traiter les directives de compilation (comme l'inclusion de fichiers d'entête de bibliothèques). Elle génère un fichier texte qui est encore un fichier source en C.

- La troisième étape est la compilation. Elle consiste à transformer les instructions du programme source en langage compréhensible par le processeur (langage machine). Elle génère un fichier binaire dit fichier objet (*extension .obj*).

- La quatrième étape consiste à effectuer l'édition de liens. Le code généré à la compilation est complété par le code des fonctions des bibliothèques utilisées. C'est seulement après cette étape que l'on génère un fichier exécutable *(extension .exe)*.

5. Types et variables

5.1.Les principaux types

int : nombre entier
short : entier court
long : entier long
char : caractère
float : nombre réel simple précision
double : nombre réel double précision

Ces types peuvent être qualifiés par les qualificateurs suivants:
signed : nombre signé
unsigned : nombre non signé

Exemple `unsigned short a ;`

5.2. Les variables

Les variables contiennent les valeurs qui sont utilisées pendant l'exécution du programme. Les noms des variables sont des identificateurs quelconques.

Syntaxe générale de déclaration : *type nomVar ;*

Exemple 1 :

```
int x, y , z;
 float a, b;
unsigned short cpt = 1000;
```

5.3. Les commentaires

Un commentaire commence toujours par les deux symboles '**/***' et se termine par les symboles '***/**'. Il est interdit d'utiliser des commentaires imbriqués.
Ou bien **//**
/ Ceci est un commentaire correct */*
//ceci est un commentaire
//ceci est un commentaire

Exemple 2

```
int x, y , z; /*déclaration de 3 variables entières*/
 float a, b; //déclaration de 2 variables réelles
unsigned short cpt = 1000;  /*déclaration d'un entier court non
signé */
```

6. Expressions et opérateurs

Opérateurs arithmétiques de base: +, - , *, /
% (reste de division)

++ incrementation
-- décrémentation

Opérateurs relationnels : Les opérateurs renvoient une valeur entière de type int égale _a 0 (faux) ou a 1 (vrai) suivant le résultat de l'opération
La comparaison **: >, <, >=, <= ,**
test égalité **: ==**
*test de différence***: !=**
Opérateurs logiques : ET logique: &&
Ou logique : ||

7. Caractères et chaines de caractères

Une constante de type caractère se note en écrivant le caractère entre apostrophes. Une constante de type chaine de caractères se note en écrivant ses caractères entre guillemets.
Par exemple Caractére 'A' , '2' et chaine de caractére : "Bonjour a tous !"

\n nouvelle ligne (LF)
\t tabulation (HT)
\b espace-arrière (BS)
\r retour-chariot (CR)
\f saut de page (FF)
\a signal sonore (BELL)
Une constante de type caractère appartient au type char,
Le codage interne d'une chaine de caractères est celui d'un tableau de caractères (c'est-`a-dire char[]).

8. Lire et écrire des données

La bibliothèque standard <*stdio*> contient un ensemble de fonctions qui assurent la communication de la machine avec le monde extérieur.

8.1.La fonction printf()

La fonction **printf** est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expressions vers le fichier de sortie standard *stdout* (par défaut l'écran).
Tableau resumant les formats d'affichage

SYMBOLE	*TYPE*	*IMPRESSION COMME*
%d ou %i	**int**	entier relatif
%u	**int**	entier naturel (unsigned)
%o	**int**	entier exprimé en octal
%x	**int**	entier exprimé en hexadécimal
%c	**int**	caractère
%f	**double**	rationnel en notation décimale

%e	**double**	rationnel en notation scientifique
%s	**char***	chaîne de caractères

Exemple 3: La suite d'instructions:

```
int A = 1234;
int B = 567;
printf("%i fois %i est %li\n", A, B, (long)A*B);
```

Affichage écran : 1234 fois 567 est 699678

8.2.La Fonction scanf()

La fonction **scanf** est la fonction symétrique à **printf**; elle nous offre pratiquement les mêmes conversions que **printf**, mais en sens inverse.

scanf("<format>",<AdrVar1>,<AdrVar2>, ...)

Exemple 4:

```
int JOUR, MOIS, ANNEE;
scanf("%d%d%d", &JOUR, &MOIS, &ANNEE);
```

9. Les structures alternatives et iteratives

Syntaxe de la structure alternative

```
if (condition){
inst1 ;
inst2 ;
…
}
else {
inst3 ;
inst4 ;
…
}
```

Syntaxe de la boucle pour

```
for (compteur=valeurInitial ; condition ; pas){
   instr1 ;
   instr2 ;
   instr3 ;
   … ;
   }
```

Syntaxe de la boucle tant que

```
while (condition vrai)
   instr1 ;
   instr2 ;
   instr3 ;
   … ;
   }
```

Syntaxe de la boucle Répéter jusqu'à

```
do{
   instr1 ;
   instr2 ;
   instr3 ;
   … ;
   } while (condition vrai);
```

Exercice

Soient x et n deux nombres entiers saisis au clavier. Calculer :

$$S = x - \frac{x^3}{3!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

En utilisant La boucle **for**,la boucle **while** et la boucle **do – while**.

Solution avec boucle for

```
#include <stdio.h>
main()
{
int n;
int i,dt ;
double x,som,f,p,x2;
int s ;

do //Boucle de test de l'entier n
{
printf("Entrer un entier naturel : ");
scanf("%d", &n);
}
while (n<0);

printf("Entrer un réel : ");
scanf("%lf", &x);

//Initialisation
som=x;
f=1;
p=x;
s=1;
x2=x*x;
//Boucle de calcul de la somme
for (i=3; i<=2*n+1 ; i=i+2) {
p = p*x2;
f = f*(i)*(i-1) ;
s = s* (-1) ;
som=som+s*p/f;
}
printf ("La somme est %.4lf\n", som);
}
```

Translation de la boucle for en while:

```
#include <stdio.h>
main()
{
int n;
int i,dt ;
double x,som,f,p,x2;
int s ;

do //Boucle de test de l'entier n
{
printf("Entrer un entier naturel : ");
scanf("%d", &n);
}
while (n<0);

printf("Entrer un réel : ");
scanf("%lf", &x);

//Initialisation
som=x;
f=1;
p=x;
s=1;
x2=x*x;

//Boucle de calcul de la somme
i=3;
while (i<=2*n+1){
p = p*x2;
f = f*(i)*(i-1) ;
s = s* (-1) ;
som=som+s*p/f;
i=i+2;
}

printf ("La somme est %.4lf\n", som);
}
```

Translation de la boucle for en boucle do-while

```
#include <stdio.h>
main()
{
int n;
int i,dt ;
double x,som,f,p,x2;
int s ;


do //Boucle de test de l'entier n
{
```

```
printf("Entrer un entier naturel : ");
scanf("%d", &n);
}
while (n<0);

printf("Entrer un réel : ");
scanf("%lf", &x);

//Initialisation
som=x;
f=1;
p=x;
s=1;
x2=x*x;

//Boucle de calcul de la somme
i=3;
do {
p = p*x2;
f = f*(i)*(i-1) ;
s = s* (-1) ;
som=som+s*p/f;
i=i+2;
} while (i<=2*n+1);

printf ("La somme est %.4lf\n", som);
}
```

Chapitre 2

Les Fonctions en C

Introduction

En C, un programme peut être divisé en plusieurs fonctions. Une seule de ces fonctions est obligatoire : la fonction principale main. Cette fonction principale peut appeler une ou plusieurs fonctions secondaires. Chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même on parlera donc de fonction récursive (cours suivant).

1. Déclaration d'une fonction

La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main.
Le corps de la fonction contient des déclarations de variables locales à cette fonction.
Il se termine par l'instruction *return*.

Syntaxe :

```
Type NomFonction (listes des paramètres formels)
{
Inst1 ;
Inst2 ;
…
return(expression);
}
```

NB :

- La fonction de type *void* ne renvoie rien
- La valeur de l'expression est la valeur que renvoie la fonction.

Exemple 1 : Déclaration d'une fonction qui retourne la plus grande valeur entre 2 valeurs.

```
 float plusGrand(float x, float y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

Exemple 2 :

- Ecrire une fonction qui retourne le factoriel d'un nombre n.
- En vous basant sur la fonction factoriel, écrire un programme en C qui calcule :

$$C(N,P) = \frac{N!}{P!\,(N-P)!}$$

```
#include <stdio.h>
         /*Déclaration de la fonction*/
long factoriel(int n)
{
int i,fact;
for (i=1,fact=1 ;i<=n ;i++)
fact=fact*i;
return fact;
}
               /*Programme principal*/
main()
{
int N,P;
long Res;
printf("N = ") ;
scanf("%d",&N) ;
printf("P= ") ;
scanf("%d",&P) ;
Res=factoriel(N)/(factoriel(P)*factoriel(N-P));
printf("C (%d, %d)= %ld \n",N,P,Res);
}
```

Exemple 3 :

- Ecrire une fonction qui extrait le minimum de deux nombres de type double.
- Ecrire une fonction qui extrait le maximum de deux nombres de type double.
- Ecrire une fonction *main* qui permet de saisir 4 nombres réels et affiche la valeur minimale et la valeur maximale.

```
#include <stdio.h>
         /*Déclaration de la fonction min*/
double min (double a,double b)
{
if (a<b) return a;
else return b;
}
```

```
         /*Déclaration de la fonction max*/
double max (double a,double b)
{
if (a<b) return b;
else return a;
}
               /*Programme principal*/
main()
{
double x,y,z,t, res1,res2;
printf ("Saisissez 4 nombres");
scanf("%lf %lf%lf%lf",&x,&y,&z,&t);
res1=min(min(x,y),min(z,t));
res2=max(max(x,y),max(z,t));
printf("Minimum(%.2lf,%.2lf,%.2lf,%.2lf)=%.2lf,\n",x,y,z,t,res1);
printf("Maximum (%.2lf,%.2lf,%.2lf,%.2lf)=%.2lf\n",x,y,z,t,res2);
}
```

2. Variables globales

On appelle ***variable globale*** une variable déclarée en dehors de toute fonction. Les variables globales sont permanentes (connues dans tous le programme).

Exemple 4: Dérouler ce code :

```
#include <stdio.h>
int n=0; /*n est globale*/

void incrementer ()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}
main()
{
  int i;
  for (i = 0; i < 5; i++)
    incrementer ();
}
```

Exécution du programme

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

3. Variables locales

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction, Les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Exemple 5 : Dérouler le programme :

```
#include <stdio.h>
int n = 10; /*variable globale*/

          /*La fonction incrementer*/
void incrementer()
{
  int n = 0;  /*variable locale*/
  n++;

  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    incrementer ();
}
```

Exécution du programme
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1

4. Passage des paramètres d'une fonction

Pendant l'appel de la fonction, les paramètres effectifs *sont copiés* dans une pile. La fonction travaille alors uniquement sur cette copie.

Si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme principale le programme qui appelle ne sera pas modifiée. On dit que les paramètres d'une fonction sont ***transmis par valeurs***.

Exemple 6 : Dérouler ce programme :

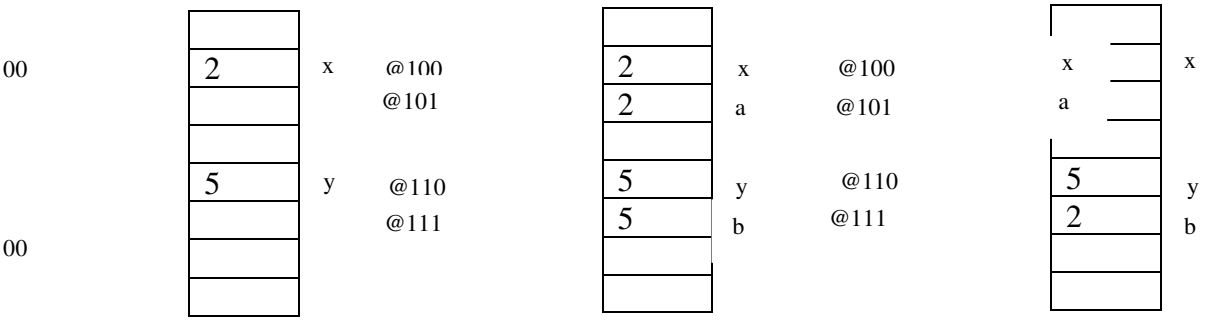
```
#include <stdio.h>

void echange (int a, int b)
{
  int t;
  t = a;
  a = b;          } //Etape 3. Permutation des copies a et b
  b = t;
  return;
}

main()
{
int x,y;
printf("donnez deux entiers");
scanf ("%d%d", &x,&y);  //Etape 1 la saisi des deux valeurs
echange(x,y) ;  //Etape 2 Appel de la fonction⇨passage par valeur
printf("fin programme principal :\n x = %d \t y = %d\n",x,y);  //Etape
4  Affichage des valeurs de x et y après permutation
}
```

Déroulement du programme

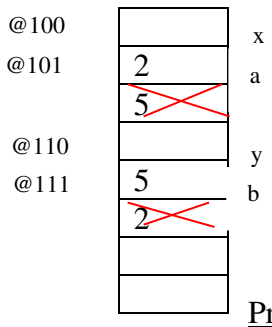
Etape 1.
La saisi des valeurs x et y donc réservation de deux emplacements mémoires

Etape 2.
Appel de la fonction echange (x,y) impliquera automatiquement la copie des valeurs et échange des valeurs a et b

Etape 3.
Affichage des nouvelles valeurs de x et y a et b seront supprimées dés qu'on sort de la fonction



Etape 4.
Affichage des nouvelles valeurs de x et y a et b seront supprimées dés qu'on sort de la fonction



⇨A la fin du programme a=2 et b=5 !

Problème

Passage par valeur : la fonction travaille uniquement sur la copie.
La copie est supprimée dès la fin de la fonction.

⇨ Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur.

Par exemple, pour échanger les valeurs de deux variables, il faut utiliser :

Solution 1 : Déclaration globale comme une première solution donc nous aurons le programme modifié :

```
#include <stdio.h>
int a,b; /*declaration globale*/

/*fonction d'echange*/
void echange()
{
```

```
  int t;
  t = a;
  a = b;
  b = t;
  return;
}

/*Fonction principale*/
main()
{
printf("donnez deux entiers");
scanf ("%d%d", &a,&b);
echange( ) ;
printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Solution 2 : Meilleure solution : utiliser le passage par référence (par adresse).

COURS PROCHAIN !!

Chapitre 3

Pointeurs & allocation dynamique de la mémoire

PARTIE I : LES POINTEURS

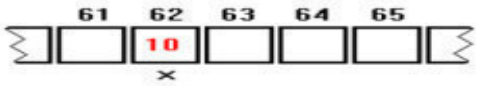
1. Notions de base

1.1. Rappel

Rappelons qu'une variable contient une valeur.

Exemple 1:

```
int x ;      /*   Réserve un emplacement pour un entier en mémoire */
x=10 ;       /*  Ecrit la valeur 10 dans l'emplamcement réservé.*/
```



- Pour afficher la valeur 10, on ecrit : `printf("x = %d \n",x) ;`
- Pour afficher l'emplacement (adresse=62) de x, on ecrit :
  `printf("adresse de x = %d \n",&x) ;`

1.2. Notion de pointeur

Un *pointeur* est un objet dont la valeur ou le contenu est égale à l'adresse d'un autre variable objet.
On déclare un pointeur par l'instruction :

type **nom-du-pointeur*; où *type* est le type de l'objet pointé.

Reprenant l'exemple 1 et déclarons un pointeur px qui contient l'adresse de la variable x:

```
int *px ; /*Réservation d'un emplamcement pour stocker une adresse mémoire*/
px=&x ;   /*Ecrire l'adresse de x dans le pointeur px*/
```

Lorsqu'on écrit `int *px ;`

Lorsqu'on écrit `px = &x ;`

Nous déduisons :

- 2 façons pour afficher la valeur de la variable x= 10

En utilisant la variable : `printf("la valeur de x = %d \n",x) ;`
En utilisant le pointeur : `printf("la valeur de x = %d \n",*px) ;`

- 2 façons pour afficher l'adresse de x = 62 :

En utilisant la variable : `printf("l'adresse de la variable x = %p \n",&x) ;`
En utilisant le pointeur : `printf("l'adresse de la variable x %p \n",px) ;`

Explications

- L'*opérateur unaire d'indirection* * permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, alors *p désigne la valeur de i.
- *L'opérateur &* permet d'accéder à l'adresse d'une variable.
- Le symbole %p est uutilisé pour afficher l'adresse

Exemple 2

```
#include <stdio.h>
main()
{
int a;
int *x,*y;
a = 10;
x = &a;
printf("%d\n", *x);
*x = 2*a;
printf("a vaut : %d\n", a);
y = x;
*y = 30;
printf("a = %d\n", a);
}
```

Que contient la variable a ?

2. Paramétres de fonction avec les pointeurs

Les pointeurs permettent aux fonctions de modifier les données elles mêmes et non leurs copies (probléme du passage par valeur)=> Nous parlerons du passage par référence.

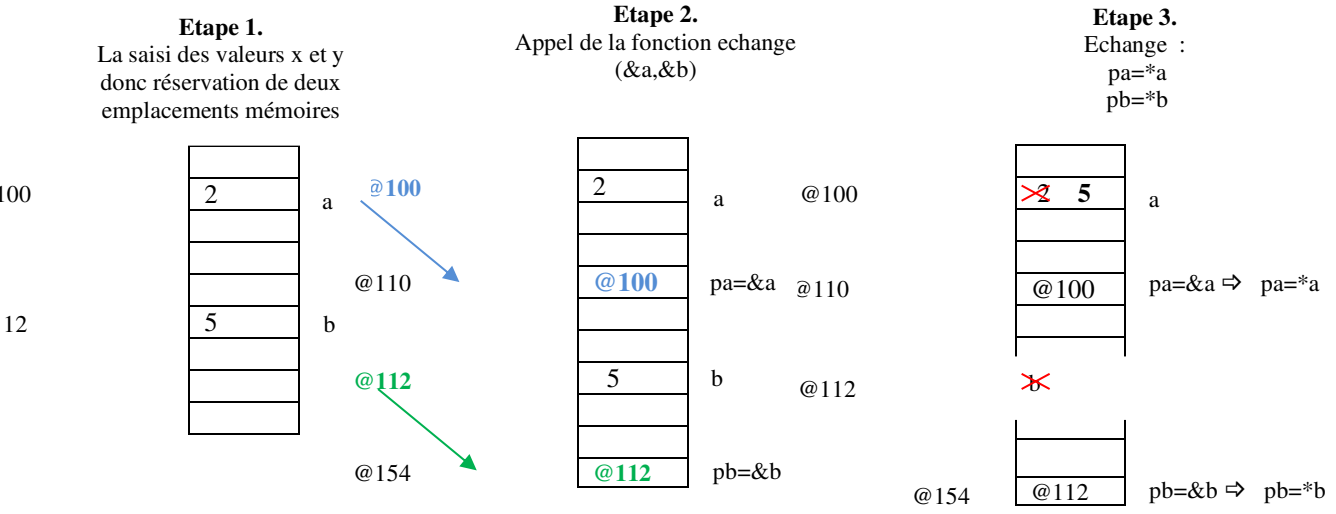
Reprenons le même exemple vu dans le cours précédent (Exemple 6 Echanger deux valeurs) et proposons une solution en se basant sur les pointeurs.

Nous aurons :

```
#include <stdio.h>
void echange (int *pa, int *pb)
{
  int t;
  t = *pa;
  *pa = *pb;
  *pb = t;
  return;
}

main(){
int a,b ;
printf("donnez deux valeurs");
scanf("%d %d", &a, &b); /*Etape 1:réservation de deux cases mémoires
                                et saisi de valeurs*/
echange(&a,&b); /*Etape2 : Appel de la fonction : echange par référence*/
printf("Après echange a = %d \t b = %d\n",a,b);
}
```

Etape 3 : Etapes d'echange

Déroulement du programme



Nous aurons donc à la fin de notre programme a= 5 et b =2.

Exercice 1
Ecrire une fonction qui determine le min et le max de 2 entiers.

```
#include <stdio.h>

void minmax(int i, int j, int* min, int* max)
{
if(i<j) { *min=i; *max=j; }
else { *min=j; *max=i; }
}

main()
{
int a, b, w, x;
printf("Tapez la valeur de a : "); scanf("%d", &a);
printf("Tapez la valeur de b : "); scanf("%d", &b);
minmax(a, b, &w, &x);
printf("Le plus petit vaut : %d\n", w);
printf("Le plus grand vaut : %d\n", x);
}
```

3. Pointeurs et tableaux

Le nom d'un tableau contient l'adresse du premier élément du tableau donc le nom du tableau se n'est rien d'autre qu'un pointeur.

Exemple 3

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
printf("premiére valeur = %d\n", A[1]);
printf("première valeur = %d\n", *A);
printf("l'adresse du 1er element = %p\n", &A[1]);
printf("premiére valeur = %p\n", A);
```

Exemple 4
Soit P un pointeur qui pointe sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;    /*Initialisation du pointeur P avec le nom du tableau*/
```

Quelles valeurs ou adresses fournissent ces expressions:

`*P+2`	=>	la valeur 14
`*(P+2)`	=>	la valeur 34
`&P+1`	=>	&P[1]
`&A[4]-3`	=>	& A[1]
`A+3`	=>	& A[3]
`P+(*P-10)`	=>	& A[2]
`*(P+*(P+8)-A[7])`	=>	la valeur 23

Exercice 2

1. Ecrire une fonction lectureTab pour la saisi des valeurs d'un tableau réel.
2. Ecrire une fonction affichageTab pour afficher le tableau.
3. Ecrire une fonction plusgrand pour determiner la valeur maximale du tableau avec son indice.
4. Ecrire une fonction somme pour determiner la somme des valeurs inferieures à une valeure donnée.
5. Testez vos fonctions dans le main.

```
#include <stdio.h>
#define TAILLE 100

void LIRE_TAB (int *TAB, int *N, int NMAX)
{
/* Variables locales */
int i;
/* Saisie de la dimension du tableau */
do{
printf("Dimension du tableau (max.%d) : ", NMAX);
scanf("%d", N);
}
while (*N<=0 || *N>NMAX);
/* Saisie des composantes du tableau */
for (i=0; i<*N; i++) {
printf("Elément[%d] : ", i);
scanf("%d", &TAB[i]);
}
}
/*-----------------------------------------------------------------*/
void ECRIRE_TAB (int *TAB, int N) {
int i ;
/* Affichage des composantes du tableau */
for (i=0; i<N; i++)
printf("%d ", *(TAB+i));
printf("\n");
}

/*-----------------------------------------------------------------*/
void plusgrand(int *TAB,int N,int *max,int *posmax) {
int i;
*max=*TAB;*posmax=0 ;
for (i=1 ;i<N; i++) {
if (*(TAB+i)>*max){ *max=*(TAB+i);*posmax=i ;}
}
}

/*-----------------------------------------------------------------*/
int somme (int *TAB, int N, int val){
int i,som=0;
for (i=0;i<N;i++){
    if (*(TAB+i)<val) som=som+*(TAB+i);
```

```
}
return som;
}

/*-------------------------------------------------------------*/
main()
{
int T[TAILLE]; /* Tableau d'entiers */
int DIM, valeur; /* Dimension du tableau */
/* Traitements */
LIRE_TAB (T, &DIM, TAILLE);
int max,posmax ;
ECRIRE_TAB (T, DIM) ;
plusgrand(T, DIM,&max,&posmax) ;
printf("maximum=%d, sa position est %d ",max,posmax ) ;
printf ("saisi une valeur   ");
scanf ("%d",&valeur);
printf   ("la   somme   des   valeurs   inf   à   %d=   %d",valeur,
somme(T,DIM,valeur));
}
```

PARTIE II. ALLOCATION DYNAMIQUE DE LA MEMOIRE

Un des principaux intérêts de l'allocation dynamique est de permettre à un programme de réserver la place nécessaire au stockage d'un tableau en mémoire dont il ne connaissait pas la taille avant la compilation. En effet, jusqu'ici, la taille de nos tableaux était fixée dans le code source.
Avec l'allocation on peut créer des tableaux de façon plus flexible .
Quand on déclare une variable, on dit qu'on **demande à allouer de la mémoire.**

1 Taille mémoire des variables

Chaque type de variable occupe un espace mémoire. Par exemple, un int occupe généralement 4 octets en mémoire et un double 8 octets.
Pour connaitre la taille qu'auccupe chaque type, nous nous basons sur l'opérateur **sizeof()**.

Exemple 5

Ecrire un programme C qui affiche la taille mémoire qu'occupe un caractére, un entier, un entier long et un double.

```
#include <stdio.h>
main(){
printf("char : %d octets\n", sizeof(char));
printf("int : %d octets\n", sizeof(int));
printf("long : %d octets\n", sizeof(long));
printf("double : %d octets\n", sizeof(double));
}
```

Aprés exécution
Char : 1 octet
Int : 4 octets
Long : 4 octets
double : 8 octets

2 Allocation de mémoire dynamique

La bibliothèque <stdlib.h> permet l'allocation mémoire dynamque .
Cette bibliothéque contient deux fonctions :

- *malloc* « Memory ALLOCation » permet la demande système d'exploitation la permission d'utiliser de la mémoire ;

```
void* malloc(size_t nombreOctetsNecessaires);
```

- *free* permet de libérer la place en mémoire

```
void free(void* pointeur);
```

L'allocation passe donc par :

a. Appel de **malloc** pour demander de la mémoire,
b. Vérification de la valeur retournée par **malloc,**
c. Libération de l'espace avec **free** une fois terminé.

Exemple 6

Allouer dynamiquement un espace int, tester l'allocation et liberer l'espace.

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

main()
{
    int* MemAllouee = NULL;  /*On crée un pointeur sur un entier */
MemAllouee = malloc(sizeof(int)); /* La fonction malloc inscrit dans notre pointeur
l'adresse qui a été reservée.*/
if (MemAllouee == NULL) //  Si l'allocation a échoué
   exit(0); // On arrête immédiatement le programme

  // On peut continuer le programme normalement sinon

free(MemAllouee); // On n'a plus besoin de la mémoire, on la libère
}
```

3 Allocation dynamique d'un tableau

Pour le moment, nous avons utilisé l'allocation dynamique uniquement pour créer une petite variable. on a besoin de l'allocation dynamique, pour créer un tableau dont on ne connaît pas la taille avant l'exécution du programme.

Exemple 7

Ecrire un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau en demandant le nombre d'amis à l'utilisateur.
Lors de la saisi du code source, on ne connait pas la taille de notre tableau. Celle-ci sera connue qu'à l'execution lorsqu'on demande le nombre d'amis de l'utilisateur.
Donc nous allons crée un tableau dont ses cases sont égales au nombred'amis d'où l'intrêt de l'allocation dynamique.

Etapes

1. Demander à l'utilisateur combien il a d'amis ;
2. Créer un tableau de int ayant une taille égale à son nombre d'amis ;
3. Demander l'âge de chacun de ses amis un à un, qu'on stocke dans le tableau ;
4. Afficher l'âge des amis pour montrer qu'on a bien mémorisé tout cela ;
5. Puisqu'on n'a plus besoin du tableau contenant l'âge des amis, le libérer avec la fonction free.

Programme en C

```
#include <stdio.h>
#include <stdlib.h>
main ()
{
int nombreAmis = 0, i = 0;
int* ageAmis = NULL; /*Ce pointeur va servir de tableau après
                     l'appel du malloc*/

// On demande le nombre d'amis à l'utilisateur
printf("Combien d'amis avez-vous ? ");
scanf("%d", &nombreAmis);

if (nombreAmis > 0) // au moins un ami
{
 ageAmis = malloc(nombreAmis * sizeof(int)); /*On alloue de la
                                 mémoire pour le tableau*/
 if (ageAmis == NULL) // On vérifie si l'allocation a marché ou non
   exit(0); // On arrête tout
// On demande l'âge des amis un à un boucle for
for (i = 0 ; i < nombreAmis ; i++)
{
 printf("Quel age a l'ami numero %d ? ", i + 1);
 scanf("%d", &ageAmis[i]);
```

```
}

//Affichage des âges stockés un à un
printf("\n\nVos amis ont les ages suivants :\n");
for (i = 0 ; i < nombreAmis ; i++)
 printf("%d ans\n", ageAmis[i]);

free(ageAmis);  // On libère la mémoire allouée
}
}
```

Exécution du programme

Combien d'amis avez-vous ? 3
Quel age a l'ami numero 1 ? 21
Quel age a l'ami numero 2 ? 18
Quel age a l'ami numero 3 ? 20

Vos amis ont les ages suivants :
21 ans
18 ans
20 ans

Chapitre 4

La récursivité

1. Définition

Une fonction récursive est une fonction qui s'appelle elle - même.
L'intérêt d'utiliser les fonctions récursives est l'immense gain de temps.

2. Résolution récursive d'un problème

Pour créer une fonction récursive, il faut :

1. Décomposer un problème en un ou plusieurs sous-problèmes du même type.
2. Les sous-problèmes doivent être de taille plus petite que le problème initial.
3. La décomposition doit conduire à un cas élémentaire, qui, lui, n'est pas décomposé en sous-problème.
4. On résout les sous-problèmes par des appels récursifs.
5. L'arrêt de l'algorithme est obtenu quand il n y aura plus d'appels.

3. Structure d'une fonction récursive

```
Type FonctionRécursive( type1 p1,type2 p2,...,  typek pk)
{
if (condition)       /* condition d'arret */
     return calcul; /* cas élémentaire */
else
Fonction Récursive(...); /* appel recursif */
return résultat;
}
```

Exemple 1: Calcul du factoriel d'un nombre.
Le factoriel est une fonction mathématique qui pour une valeur entière positive, retourne le produit de tous les entiers entre 1 et cette valeur. Pour une valeur nulle, la fonction retourne 1.

Par exemple, 5!= 5 . 4 . 3 . 2 .1 = 120.
On peut écrire la fonction factorielle sous la forme d'une simple boucle (*implémentation itérative)* ou de manière récursive.

Implémentation récursive

Sous-problème : $n!=n\times(n-1)!$

$(n-1)!=(n-1)\times(n-2)!$

$(n-2)!=(n-2)\times(n-3)!$

.

.

.

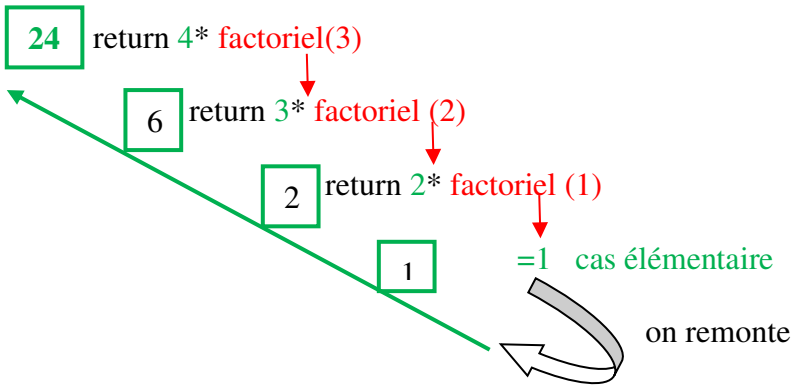
$1!=1\times 0!$

$0!=1$

Donc pour n=1 ou n=0, le résultat =1 (cas élémentaire)
Sinon on refait le calcul.

```
#include <stdio.h>
unsigned long int n;

unsigned long int factoriel(unsigned long int n)
{
if(n <= 1)
return 1;  //cas élémentaire
else
return n * factoriel(n-1);  // appel récursif
}

main()
{
printf("Entrer un entier positif \n");
scanf("%ld", &n);
//Appel récursif et affichage du résultat final
printf("%ld != %ld\n", n, factoriel(n)) ;
}
```

Déroulement pour n=4



La récursivité nécessite l'emploi d'une pile pour stocker les résultats intermédiaires.
Le factoriel peut se résoudre sans récursivité, on parle alors d'implémentation itérative.

L'avantage de l'implémentation récursive est le gain de temps. Mais, son inconvénient réside dans l'utilisation d'une grande quantité de mémoire.

4. Types de récursivité

Il existe deux types de fonctions récursives :
- Les fonctions récursives terminales
- Les fonctions récursives non terminales.

4.1. Fonction récursive terminale

Une fonction récursive est dite terminale si aucun traitement n'est effectué à la remontée d'un appel récursif sauf le retour d'une valeur. Il n'y a pas de calcul entre l'appel récursif et l'instruction ***return***.
Les appels récursifs n'ont pas besoin d'être empilés dans la pile d'exécution car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

4.2. Fonction récursive non terminale

Une fonction récursive est dite non terminale si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).

Le factoriel de l'exemple 1 est basé sur une récursivité non terminale puisqu'il y a multiplication par n avant return.

Exemple 2 : Ecriture du factoriel sous forme récursivité terminale.

L'idée est de supprimer le calcul qui se fait dans l'appel récursif `return n*fact(n-1)`. Il faudra donc injecter une variable dans la définition de la fonction qui va accumuler les calculs au fur et a mesure. Nous aurons donc :

```
//Programme C de résolution récursive terminale du factoriel
#include <stdio.h>
unsigned long int n,resultat;

int factoriel(int n, int accu)
{
    if (n == 0) return accu;
    else return factoriel(n - 1, n * accu);
}

main()
{
printf("Entrer un entier positif \n");
scanf("%ld", &n);
resultat=1 ;
//Appel récursive et affichage du résultat final
printf("%ld != %ld\n", n, factoriel(n,resultat));
}
```

L'appel récursif `return factoriel(n - 1, n * accu)` est donc terminal.

5. Passage du récursif à l'itératif

Un programme itératif se base sur des boucles pour traiter un certain nombre d'éléments. Le passage du récursif à l'itératif reviendra à faire de la **dé-récursivité.**

On peut transformer une fonction récursive terminale en itération pour optimiser l'exécution.

Une fonction récursive terminale a pour forme générale :

```
Type recursive(P)
{
I0;
if (Condition) return element
else recursive(P'));
}
```

Forme générale pour passage à la forme itérative

```
Type Iteratif(P)
{
I0;
while (non Condition)
P'=f(P);
return (resultat) ;
}
```

Avec **f**: la fonction de transformation des paramètres P

Exemple 3

```
unsigned long int factoriel(unsigned long
                            int n)
{
if(n <= 1)
return 1;
else
return n * factoriel(n-1);  }
```

```
unsigned long int factoriel(unsigned long
                                int n)
{
  int accu = 1;
  while (n>1)
  {
    accu = n*accu;
    n = n-1;
  }
  return accu;
}
```

Exemple 4

Proposer une fonction récursive et une autre itérative pour le calcul de la somme

$$u_n = 1 + 2^4 + 3^4 + \cdots + n^4$$

```
int RecursiveSomme(int n)
{
if (n <= 0) return 0;
else return (n * n * n * n + Somme(n - 1));
}

int IterativeSomme(int n)
{
int  i,som;
for (i=2,som=1;i<=n;i++)
som=som+ i*i*i*i;
return som;
}
```

Chapitre 5

Structures complexes : listes chainées et piles

1. Définition d'une structure

Nous avons déjà vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.
La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nommé champ) se fera, cette fois, non plus par une indication de position, mais par son nom au sein de la structure.

Exemple 1
Décrire une structure produit servant à stocker le numéro, la quantité et le prix unitaire d'un produit.

En C, nous aurons :

```
struct produit {
int numero;
int qte ;
float prix ;
} ;
```

2. Les listes chainées

Les listes sont des structures de données informatiques qui permettent, au même titre que les tableaux par exemple, de garder en mémoire des données en respectant un certain ordre : on peut ajouter, supprimer ou consulter un élément en début ou en fin de liste, vider une liste ou savoir si elle contient un ou plusieurs éléments.

Les langages de programmation gèrent les structures de données, permettant l'accès à tous les éléments de la liste.
Une c*ellule,* plus connue sous le nom de *maillon* possède un ou plusieurs *champs*.
Les cellules permettent de stocker des données.

Une **liste chaînée** désigne une liste ordonnée, dont la représentation en mémoire est une succession de maillons.
Chaque maillon (cellule) contient une ou plusieurs données et un pointeur vers le maillon suivant.
Il existe deux plusieurs types de listes chaînées : Liste simplement chaînée et liste doublement chaînée.

Nous nous intéressons aux listes simplement chainées.

Les listes simplement chaînées sont plus flexibles que les tableaux car on peut ajouter et supprimer des cases à n'importe quel moment.

Dans une liste simplement chaînée, deux informations composent chaque élément de la liste chaînée :

- La valeur associée à l'élément
- Un pointeur vers l'élément suivant (successeur).

Comme un seul élément de la liste est pointé, l'accès se fait dans un seul sens, chaque élément est une structure qui contient l'adresse de l'élément suivant.

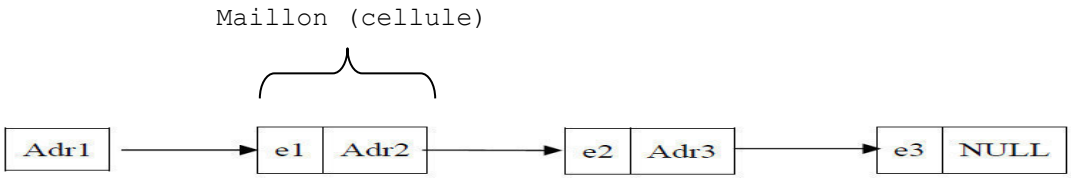


Figure.1. Représentation d'une liste simplement chaînée en mémoire.

Dans une liste, nous pouvons insérer des éléments ou les supprimer. Mais avons ca, il faut au préalable créer une liste.

2.1. Création d'une liste vide

La toute première étape consiste à créer une structure ***Liste*** contenant par exemple une donnée *val* et un pointeur pour pointer vers l'élément suivant **suivant* .

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

struct Liste
{
int val;
struct Liste *suivant;
};
typedef struct Liste maliste;
maliste tete=NULL; /*maliste de type Liste est vide
```

Nous aurons :

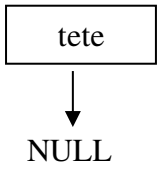


Figure. 2. Création d'une liste vide

2.2. Insertion dans une liste

2.2.1. Insertion en début de liste

La fonction implémentée en C ***insertion_debut*** a comme paramètre le pointeur *tete* de*maliste* et la *valeur* à insérer.

L'insertion passe par :

1. Allocation mémoire pour l'élément *(*elem)* et le maillon à insérer (**debut),*
2. Insertion de l'élément nouveau en testant si la *tete* est null (liste vide) ou non.

Implémentation en C

```
maliste * insertion_debut(maliste *tete,int valeur)
{
maliste *debut = malloc(sizeof(maliste));
if ((debut==NULL)||(elem==NULL)) exit(0);
debut->val = valeur;

if(tete==NULL)
{
debut->suivant=NULL;
tete=debut;
}
else
{
debut->suivant=tete;
tete=debut;
}
return tete;
}
```

if(tete==NULL)

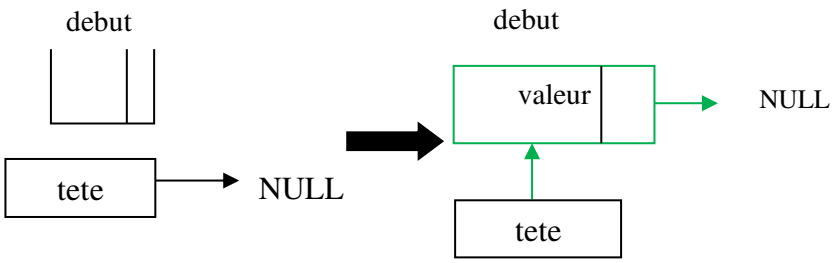


Figure .3: Insertion d'un élément dans une liste vide

if (tete !=NULL)

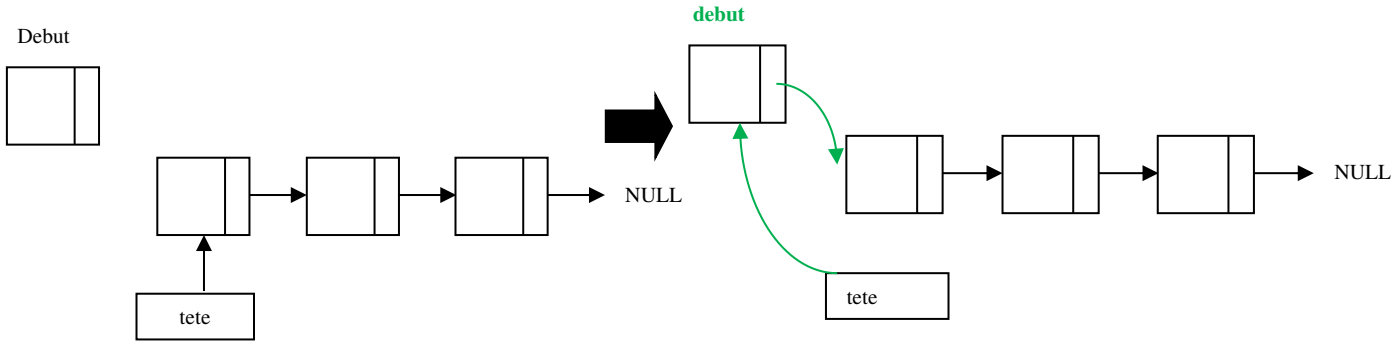


Figure .4 : Insertion d'un élément en début d'une liste

2.2.2. Insertion en fin de liste

La fonction implémentée en C **insertion_fin** a comme paramètre le pointeur *tete* de *maliste* et la *valeur* à insérer.

L'insertion passe par plusieurs étapes :

1. On alloue de la mémoire pour le maillon à insérer (**dernier*) et (**elem*) pour parcourir *maliste*.
2. On parcourt avec une boucle while chaque élément de la liste jusqu'a ce que `elem->suivant=NULL` (Fin de liste).
3. On relie les deux éléments : *elem* et *dernier*.

Implémentation en C

```
maliste * insertion_fin(maliste *tete,int valeur)
{
maliste *dernier = malloc(sizeof(maliste));
maliste *elem = malloc(sizeof(maliste));
if ((dernier==NULL)||(elem==NULL))
        exit(0);

dernier->val=valeur;
dernier->suivant=NULL;

elem=tete;
while(elem->suivant!=NULL)
elem=elem->suivant;

elem->suivant=dernier;

return tete;
}
```

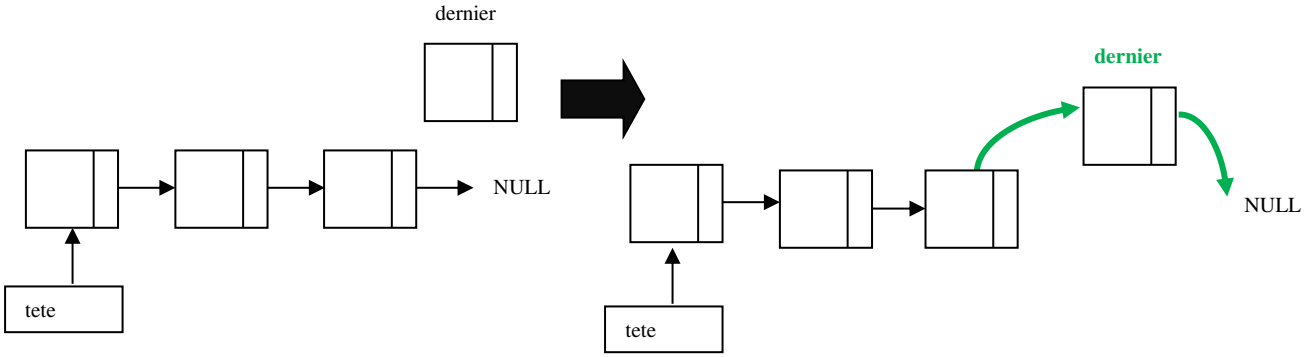


Figure .5. Insertion d'un élément en queue de liste

2.2.3. Insertion au milieu d'une liste

La fonction implémentée en C ***insertion_milieu*** a comme paramètre le pointeur *tete* de *maliste*, la *valeur* à insérer et la *position* voulue.

L'insertion passe par plusieurs étapes :

1. On alloue de la mémoire pour le maillon à inserer (**milieu*) et (**elem*) pour parcourir *maliste*.
2. On parcourt avec une boucle « *pour* » chaque élément de la liste jusqu'à atteindre la *position* souhaitée.
3. On relie les deux éléments : *elem* et *milieu.*

Implémentation C

```
maliste * insertion_milieu(maliste *tete,int valeur,int position)
{
int i;
maliste *milieu = malloc(sizeof(maliste));
maliste *elem = malloc(sizeof(maliste));

if ((milieu==NULL)||(elem==NULL))
        exit(0);

milieu->val=valeur;
elem=tete;
for(i=0;i<position-2;i++)
elem=elem->suivant;

/*ordre important des 2 instructions suivantes sinon on perd */
/* le reste de la liste*/
milieu->suivant=elem->suivant;
elem->suivant=milieu;

return tete;
}
```

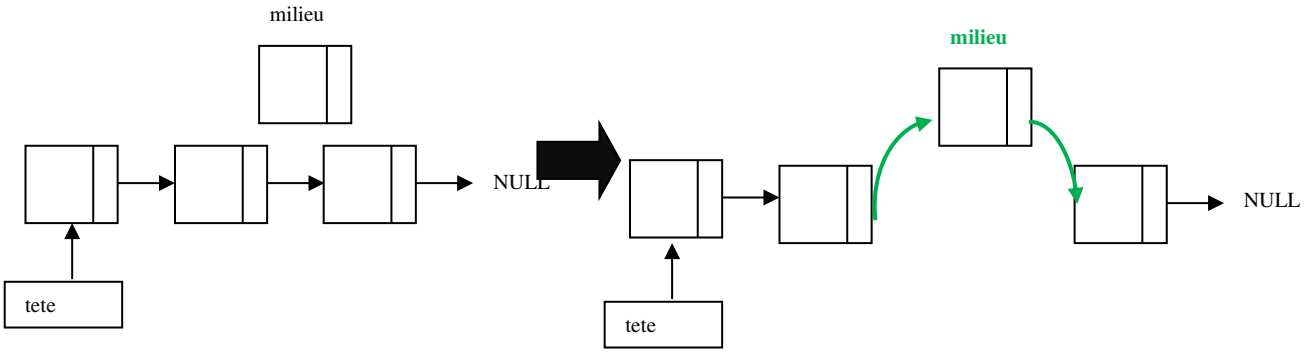


Figure .6. Ajout au milieu de liste

2.3. Suppression dans une liste

2.3.1. Suppression en début de liste

La fonction implémentée en C ***supp_debut*** a comme paramètre le pointeur *tete* de *maliste*. La suppression début consiste directement à affecter au pointeur *tete* l'adresse suivante. On n'oublie pas de libérer l'espace avec la fonction *free*.

Implémentation en C

```
maliste * supp_debut(maliste *tete)
{
int i;
if (tete == NULL)
        exit(0);

maliste *supp_debut=tete ;
tete=tete->suivant;
free (supp_debut);
return tete;
}
```

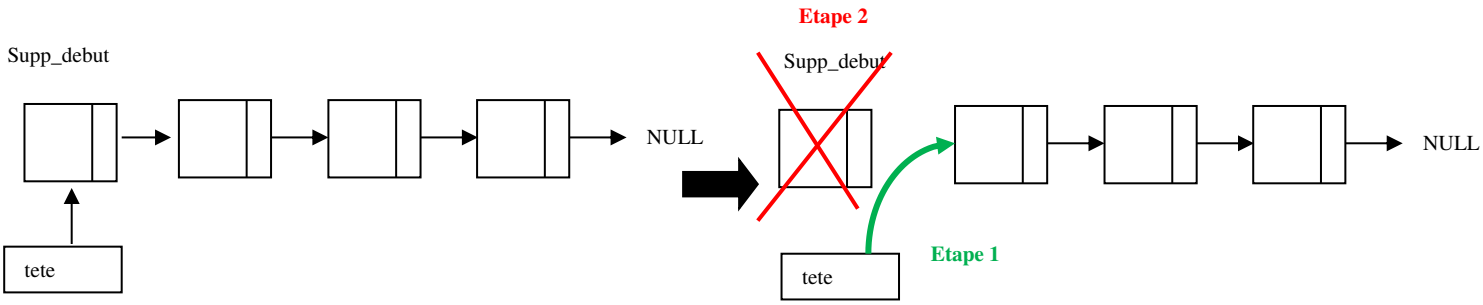


Figure .7. Suppression en tête de liste

2.3.2. Suppression en fin de liste

La fonction implémentée en C ***supp_fin*** a comme paramètre le pointeur *tete* de *maliste*.
La suppression en queue de liste consiste à parcourir la liste jusqu'au NULL tout en sauvegardant à l'avance l'adresse de l'élément précédent (**figure 8**).
La dernière étape consiste en la libération de l'espace.

Implémentation en C

```
maliste * supp_dernier(maliste *tete)
{
int i;
maliste *elem_precedent;
maliste *elem_supp=tete;
while (elem!=NULL)
 {
  elem_precedent=elem_supp; /*sauvegarder le precedent*/
  elem_supp=elem_supp->suivant;
 }
elem_precedent=NULL;
free(elem_supp);
return tete;
}
```

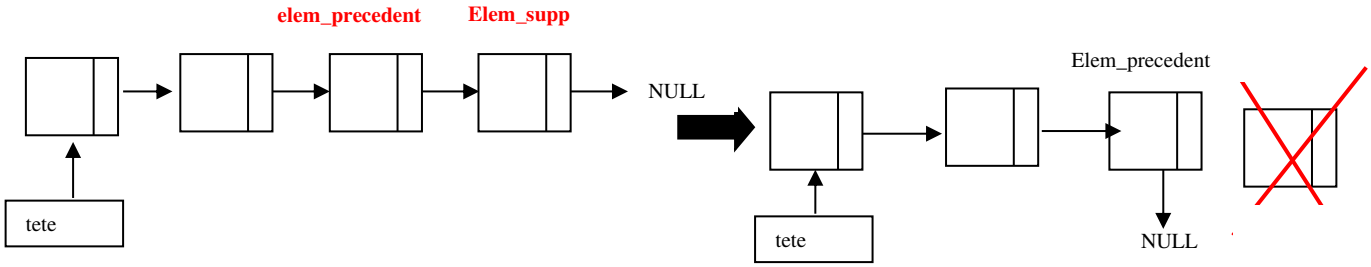


Figure .8. Suppression en queue de liste

2.3.3. Suppression au milieu de liste

La fonction implémentée en C ***supp_milieu*** a comme paramètre le pointeur *tete* de *maliste* et la *position* de suppression.
La suppression consiste en :

1. Initialiser *elem* avec la *tete* de liste.
2. Parcourir la liste élément par élément jusqu'à la position.
   Sauvegarder le maillon à supprimer dans *elem_supp* (pour pouvoir le libérer après).
   Etablir le lien avec `elem->suivant=elem->suivant->suivant`
3. Libérer *elem_supp*.

Implémentation en C

```
maliste * supp_milieu(maliste *tete,int position)
{
int i;
maliste *elem_supp;
maliste *elem=tete;

for(i=0;i<position-2;i++)
elem=elem->suivant;

elem_supp=elem->suivant;
elem->suivant=elem->suivant->suivant;
free(elem_supp);

return tete;
}
```

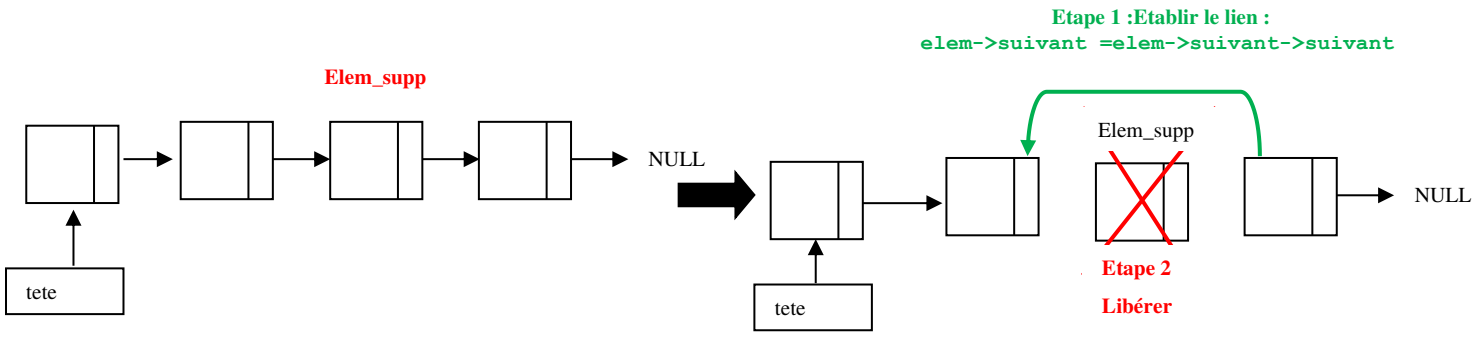


Figure .9. Suppression au milieu de liste

3. Les piles

3.1. Définition

Une pile est une séquence d'éléments accessibles par une seule extrémité appelée sommet. Toutes les opérations définies sur les piles s'appliquent à cette extrémité. L'élément situé au sommet s'appelle le sommet de pile.
Une pile est gérée suivant la politique **LIFO** (**L**ast **I**n **F**irst **O**ut) (dernier arrivé premier servi), ce qui signifie en clair que les derniers éléments à être ajoutés à la pile seront les premiers à être récupérés.

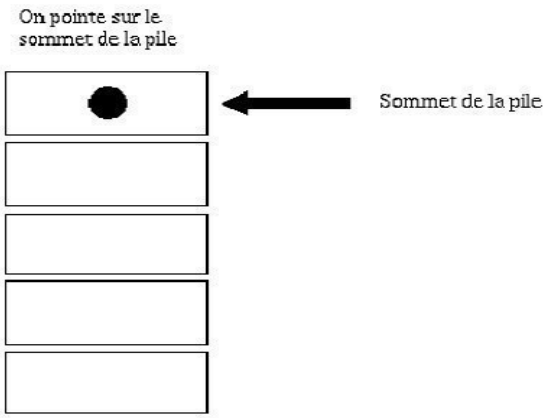


Figure 10. Représentation d'une pile

3.2. Opérations sur les piles

Plusieurs opérations peuvent être effectuées sur les piles dont les plus importantes sont :
- Créer une pile vide ;
- Tester si une pile est vide ;
- Accéder à l'information contenue dans le sommet de la pile ;
- Ajouter un élément au sommet de la pile (empiler) ;
- Supprimer l'élément qui se trouve au sommet de la pile (dépiler).

Opérations
pile_vide : → Pile
est vide : Pile → booléen
empiler : Pile * élément Pile → Pile
dépiler : Pile →Pile
sommet : Pile → élément

Les opérations ci-dessus ne sont pas définies partout, on a les pré-conditions suivantes où P est de sorte Pile et e est de sorte élément:

dépiler(P) **est définie ssi** est vide(P) = faux
sommet(P) **est définie ssi** est vide(P) = faux

En supposant les pré-conditions vérifiées, ces opérations vérifient les axiomes suivants :
dépiler(empiler(P, e)) = P
sommet(empiler(P, e)) = e
est vide(pile_vide) = vrai
est vide (empiler(P, e))= faux

3. 3. Représentation des piles

3.3.1. Représentation contiguë

Dans cette représentation, les éléments de la pile sont rangés dans un tableau. De plus, il faut conserver l'indice du sommet de la pile et la taille maximale du tableau utilisé.

3.3.2 Représentation chaînée

Chaque élément de la pile pointera vers l'élément précédent. La liste pointera toujours vers le sommet de la pile. Voici donc la structure qui constituera notre pile :

```
struct Pile{
int val;
struct Pile *precedent;
};
typedef struct Pile mapile;
mapile *tete=NULL;
```

Chaque case d'une pile représente un élément. Les cases sont en quelque sorte emboitées les unes sur les autres. Le pointeur est représenté par le jeton noir (voir **figure 10**).
Enfin, on peut dire que les piles sont un cas particulier des listes chaînées.

a. Ajout d'un nouvel élément (Empilement)

Lors de l'ajout d'un élément dans une pile veut ajouter, puis nous devrons nouvel élément. Ceci dit que le nouvel élément est devenu le sommet de la pile.

Fonction en C d'empilement

```
mapile * empiler(mapile *tete,int valeur)
{
mapile *p_nouveau = malloc(sizeof(mapile));
if (NULL != p_nouveau)
{
p_nouveau->val = valeur;
p_nouveau->precedent=tete;
tete=p_nouveau;

return tete;
}
}
```

Explication

- On crée un nouvel élément de type Pile ;
- On vérifie que l'élément a bien été créé ;
- On assigne à la valeur de cet élément la donnée que l'on veut ajouter ;
- On fait pointer cet élément sur le sommet de la pile ;
- On fait pointer le sommet de pile sur l'élément ajouté ;
- On retourne le nouveau sommet de la pile ;

b. Suppression d'un élément (dépilement)

Dans une pile, nous supprimons toujours l'élément qui se trouve en sommet de pile, pour ce faire, il nous faudra utiliser la fonction free.

Si la liste n'est pas vide, on stocke l'adresse du sommet de pile après suppression

Fonction en C de dépilement

```
mapile *depiler(mapile *tete)
{
mapile * temp;
int elem;
if(tete != NULL)
{
temp = tete->precedent;
free(tete);
tete = temp;
return tete;
}
else
printf("La pile est vide\n");
}
```

Explication

- Vérifier si la pile n'est pas vide ;
- Si elle ne l'est pas, stockez dans un élément temporaire l'avant-dernier élément de la pile ;
- Supprimer le dernier élément dans la tète ;
- Faire pointer la pile vers notre élément temporaire ;
- On retourne le nouveau sommet de la pile ;

Code C :

```
#include<stdio.h>
#include<stdlib.h>
/*-------------------------------------------------------*/
/*DECLARATION DE LA STRUCTURE PILE*/
struct Pile {
int val;
struct Pile *precedent;
};
typedef struct Pile mapile;
mapile *tete=NULL;

/*-------------------------------------------------------*/
/*FONCTION EMPLILEMENT*/

mapile * empiler(mapile *tete,int valeur)
{
mapile *p_nouveau = malloc(sizeof(mapile));
if (NULL != p_nouveau)
{
p_nouveau->val = valeur;
p_nouveau->precedent=tete;
tete=p_nouveau;
return tete;
}
}
/*-------------------------------------------------------*/
/*FONCTION DEPILEMENT*/
mapile *depiler(mapile *tete)
{
mapile * temp;
int elem;
if(tete != NULL)
{
elem = tete->val;
temp = tete->precedent;
free(tete);
tete = temp;
return tete;
}
else printf("La pile est vide\n");
}
/*-------------------------------------------------------*/
main()
{
int i,a;
mapile *pp;
printf ("la tete de la pile initialement pointe vers %p ",tete);
printf ("\n Insertion du 1er élément dans la pile \t");
scanf ("%d",&a);
tete=empiler (tete, a);
```

```
printf ("\n -----Insertion des éléments------- \n");
do
{
 printf ("donnez un entier \t");
 scanf ("%d",&a);
 if (a!=0) tete=empiler (tete, a);  /* la condition est
nécessaire pour que le 0 ne s'empile pas*/
}
while (a!=0);

printf ("\n ------Affichage des éléments insérés-------\n");
for (pp=tete;pp!=NULL;pp=pp->precedent)
  printf ("|%d|\n",pp->val);


/*La procédure de Dépilement*/
int choix;

printf ("\n Taper 1 si vous voulez dépiler sinon un autre nombre
pour arrêter\t");
scanf ("%d",&choix);

while (choix==1)
{
tete=depiler(tete);
printf ("Taper 1 si vous voulez depiler sinon un autre nombre
pour arrêter\t");
scanf ("%d",&choix);
}

printf ("---------------\n");
printf ("\n-----------Etat de la pile après dépilement----------
\n");

for (pp=tete;pp!=NULL;pp=pp->precedent)
  printf ("|%d|\n",pp->val);

}
/*FIN DU PROGRAMME C*/
```

TP 1

Présentation de l'environnement de travail

Un langage de programmation est un moyen formel permettant de décrire des traitements (i.e. des tâches à effectuer) sous la forme de programmes (i.e. de séquences d'instructions et de données de haut niveau, c'est-à-dire compréhensibles par le programmeur) et pour lequel il existe un compilateur permettant l'exécution effective des programmes par un ordinateur.
Les aspects syntaxiques (règles d'écriture des programmes) et sémantiques (définition des instructions) d'un langage de programmation doivent être spécifiés de manière précise.

L'écriture, la compilation, la mise au point et l'exécution d'un programme C fait appel à différents outils.
Les instructions du programme doivent être rentrées comme du texte normal dans un fichier à l'aide d'un éditeur de texte. On parle de *programme source*. Le fichier contenant un programme source doit comporter *l'extension .c*(Ex: programme.c). Ce fichier source doit ensuite être compilé à l'aide d'un compilateur afin de créer un exécutable (fichier binaire) ou un fichier objet (d'extension .o). Ceci est fait en utilisant un compilateur. Une fois la compilation, nous passons à l'exécution du programme.
Il existe différents éditeurs, nous travaillons avec le Code Blocks.

Partie 1. Structures alternatives et itératives

Exercice 1 : Ecrire un programme affichant à l'écran le message : *C'est mon premier programme*. Le compiler puis l'exécuter.

Exercice 2 : Ecrire un programme qui affiche le maximum et le minimum entre 3 nombres réels saisis au clavier.

Exercice 3 : Ecrire un programme qui dit si un nombre entier est pair ou impair.

Exercice 4 : Ecrire un programme en C qui permet de calculer la moyenne d'une suite des nombres positifs ou nuls. La fin de cette suite est déterminée par la lecture d'un nombre négatif.

Exercice 5 : En utilisant La boucle **for**,la boucle **while** et la boucle **do – while**, écrire un programme qui calcule

$$S= -1+\frac{1}{2!}-\frac{1}{3!}+\frac{1}{4!}-\frac{1}{5!}+\frac{1}{6!}-\frac{1}{7!}+\frac{1}{8!}- \ \ldots\frac{1}{n!}$$

Partie 2. Tableaux et matrices

Exercice 1

Ecrire un programme qui :

- Lire la dimension N d'un tableau T du type **int** (dimension maximale: 50 composantes),
- Remplir le tableau par des valeurs entrées au clavier et afficher le tableau.
- Calculer et afficher la somme des éléments du tableau.

Exercice 2

Soit une matrice réelle, écrire un programme permettant de mettre à zéro les valeurs de la deuxième diagonale et de sommer les valeurs de la première diagonale.
Afficher la nouvelle matrice et le résultat de la somme trouvé.

Exercices supplémentaires

Exercice 1

Ecrire un programme qui affiche la résistance équivalente à trois résistances R1, R2, R3 (type **double**),
- si les résistances sont branchées en série:
$R_{série} = R1+R2+R3$
- si les résistances sont branchées en parallèle:

$$R_{par} = \frac{R1 \cdot R2 \cdot R3}{R1 \cdot R2 + R1 \cdot R3 + R2 \cdot R3}$$

Exercice 2

Ecrire un programme qui lit deux nombres entiers a et b et donne le choix à l'utilisateur :
1. de savoir si la somme a + b est paire ;
2. de savoir si le produit ab est pair ;
3. de connaître le signe de la somme a + b ;
4. de connaître le signe du produit ab.

Exercice 3

Soient x et n deux nombres entiers saisis au clavier. Calculer :

$$S = x - \frac{x^3}{3!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

En utilisant La boucle **for**,la boucle **while** et la boucle **do – while**.

Corrigé

Exercice 2

```
#include <stdio.h>
main()
{
int a, b,c,min,max;
printf ("Saisi de 3 nombres:\n");
scanf ("%d",&a);
scanf ("%d",&b);
scanf ("%d",&c);
if (a<b)
 {
   min=a;
   max=b;
 }
else
 {
   min=b;
   max=a;
}

if (c<min) min=c;
else if (c>max) max=c;

printf ("le minimum entre (%d %d %d) est %d \n \n",a,b,c,min);
printf ("le maximumn entre (%d %d %d) est %d \n \n",a,b,c,max);
}
```

Exercice 3

```
#include<stdio.h>
main()
{
int a;
printf("Donnez un nombre:");
scanf ("%d",&a);
if (a%2==0)printf ("%d est paire", a);
else printf ("%d est impaire", a);
}
```

Exercice 4

```
#include <stdio.h>
main()
{
int nbr,s;
int comp;
s=0,comp=0;
do
{
printf ("Saisissez un nombre:  ");
scanf ("%d",&nbr);
if (nbr>=0) {
             s=s+nbr;
             comp++;
```

```
            }
} while (nbr>=0);

printf ("La moyenne de la suite saisie es %.2f", (float) s/comp);
}
```

Exercice 5

```
#include <stdio.h>
main()
{
int n,i;
double som,f;
int signe;

/*Tester le degré n*/
do
{
printf("Entrer un entier naturel : ");
scanf("%d", &n);
}
while (n<0);


/*Initialisation*/
som=0;
signe=1;
f=1;
/*Resolution avec la boucle pour */
for (i=1; i<=n ;i++)
{
f = f*i ;  //calcul du factoriel
signe= signe* (-1) ;
som=som+signe*1/f;
}

printf ("La somme est %.4lf\n", som);  /*affichage de resultat
avec 4 chiffres aprés la virgule*/
}
```

Partie 2. Tableaux et matrices

Exercice 1

```
#include <stdio.h>
# define taille 50
main ()
{
int tab[taille]; /*déclaration statique d'un tableau d'entier de
dimension maximale =50*/
int i,n;

/*ETAPE 1:   Saisi de la dimension du tableau*/
do{
```

```
printf ("donnez la dimension de votre tableau \t");
scanf ("%d",&n);
}
while ((n>taille)||(n<1));
/*ETAPE 2:    Boucle de Saisi des valeurs du tableau*/
for (i=0;i<n;i++){
printf ("donnez la valeur de la case %d \t",i+1);
scanf ("%d",&tab[i]);
}
/*ETAPE 3: Boucle d'affichage  des valeurs du tableau*/
printf ("\n Affichage du tableau \n");
for (i=0;i<n;i++)
     printf ("tab [%d]=%d \n ",i,tab [i]);
/*ETAPE 4: Boucle pour le calcul de la somme des valeurs du
tableau*/
 int S=0;
for (i=0;i<n;i++)
     S=S+tab [i];
printf ("La somme des valeurs du tableau est %d \n",S);
}
```

Exercice 2

```
#include <stdio.h>
# define taille 50
main ()
{
int  mat[taille][taille]; /*déclaration statique d'une matrice
carrée de dimension maximale =50*/
int i,j,n;

/*ETAPE 1:   Saisi de la dimension du tableau*/
do{
printf ("donnez la dimension de votre matrice \t");
scanf ("%d",&n);
}
while ((n<1)||(n>taille));

/*ETAPE 2:    Boucle de Saisi des valeurs de la matrice*/
for (i=0;i<n;i++)
for (j=0;j<n;j++)
{
    printf ("Donnez la valeur de la case %d,%d  \t",i+1,j+1);
    scanf ("%d",&mat[i][j]);
}

/*ETAPE 3: Boucle pour le calcul de la somme des valeurs de la
1ere diagonale*/
 int S=0;
for (i=0;i<n;i++)
        S=S+mat [i][i];
printf ("\n");
```

```
printf ("La somme des valeurs de la 1ere diagonale de la matrice
est  %d \n",S);

/*ETAPE 4: Mettre à zéro les valeurs de la 2eme diagonale */
for (i=0;i<n;i++)
 mat [i][n-i-1]=0;
printf ("Affichage de la matrice aprés la mise à 0 des valeurs
de la 2éme diagonale\n");
 /*ETAPE 5: Boucle d'affichage  des valeurs de la matrice*/
for (i=0;i<n;i++)
{
    for (j=0;j<n;j++)
    printf ("mat [%d][%d]=%d \n ",i+1,j+1,mat [i][j]) ;
printf ("\n");
}
}
```

TP2

But
Implémenter sous C les fonctions et les fonctions récursives.
Initier l'étudiant aux pointeurs.

Partie 1. Les Fonctions

Exercice 1

1. **Variante 1**

- Déclarer 2 variables entières *a,b* dans la classe globale.
- Ecrire une fonction *add* qui retourne la somme de deux nombres entiers *a,b.*
- Ecrire la fonction *main* qui réalise la saisi des deux variables *a,b* et fait appel à la fonction *add.*
- Compiler et exécuter.

2. **Variante 2**

- Déclarer 2 variables entières *a,b* dans la classe main (elles sont locales).
- Ecrire une fonction d'addition *add* retourne la somme des deux variables données en paramètres. (Cette fonction accepte doc 2 paramètres x, y et retourne un entier).
- Ecrire la fonction main qui saisi les deux variables locales *a,b* et fait appel à la fonction *add.*
- Compiler et exécuter.

Que retenez vous des deux variantes?

Exercice 2

- Ecrire une fonction qui affiche tous les diviseurs d'un nombre et retourne la somme de ses diviseurs.
- Tester dans la fonction principale *main*.

Exercice 3 (La récursivité)
On rappelle que les nombres de *Fibonacci* sont définis de la façon suivante :

$$\begin{cases} F1=F2=1 \\ Fn=Fn-1+Fn-2 \ pour \ (n \geqslant 3) \end{cases}$$

- Ecrire une fonction récursive qui calcule le nième nombre de Fibonacci.
- Tester la fonction dans un programme principal.

Partie 2. Les pointeurs

Exercice 1
Déclarer un entier *age* et un pointeur *p* pointant vers cet entier.
Initialiser l'entier à la valeur 18.
Afficher :

- Le contenu de la variable sans le pointeur.
- Le contenu de la variable en utilisant le pointeur.
- L'adresse de la variable sans le pointeur.
- L'adresse de la variable en utilisant le pointeur.

Exercice 2
Soit le progamme C :

```
#include <stdio.h>

void calcul (int a, int b){
a=2* a; int c = a+b;
}

main ()
{
int x,y;
printf ("donnez une valeur x:  " ); scanf ("%d",&x);
printf ("donnez une valeur y:  " ); scanf ("%d",&y);
calcul(x,y);
printf ("les nouvelles valeurs aprés appel: x=%d, y=%d",x,y);}
```

- Lisez les lignes du code, compiler et exécuter.
- Détectez l'erreur et corrigez ce programme

Exercice 3
Soit un tableau d'entiers.

1. La fonction *LIRE_TAB c*omprenant trois paramètres TAB, N et NMAX, lit la dimension N et les composantes d'un tableau TAB. La dimension N doit être inférieure à NMAX.
Implémenter la fonction *LIRE_TAB*.

2. La fonction *ECRIRE_TAB* à deux paramètres TAB et N qui affiche N composantes du tableau TAB.
Implémenter cette fonction.

3. Ecrire la fonction *ECHANGE* qui permute le contenue de deux variables entières (Faites dans l'exercice 2).

4. Ecrire la fonction *INVERSE_TAB (T,N)* qui range les éléments d'un tableau T dans l'ordre inverse sans utiliser de tableau d'aide.

5. A l'aide des fonctions précédentes, Ecrire un programme qui lit un tableau **T** d'une dimension DIM (DIM<=100) et qui affiche le tableau **T** avant et après l'appel de *INVERSE_TAB(T, DIM*).

Exercices supplémentaires
Exercice 1

- Ecrire une fonction qui extrait le minimum de deux nombres de type double.
- Ecrire une fonction qui extrait le maximum de deux nombres de type double.
- Ecrire une fonction *main* qui permet de saisir 4 nombres réels et affiche la valeur minimale et la valeur maximale.

Exercice 2
Ecrire la fonction NCHIFFRES du type **int** qui obtient une valeur entière N (positive ou négative) du type **long** comme paramètre et qui fournit le nombre de chiffres de N comme résultat.
Ecrire un programme qui teste la fonction NCHIFFRES:
Exemple : Introduire un nombre entier : 6457392
Le nombre 6457392 a 7 chiffres.

Exercice 3
Écrire un programme en utilisant une fonction récursive permettant de multiplier deux entiers positifs a et b selon le principe récursif suivant :
a*b = a*(b-1) +a si b est impair
a*b = (2*a)*(b/2) si b est pair et différent de 0
Exemple : 36*7=36*6+36
= 72*3+36
= 72*2*108
=144*1+108
=144*0+252
=252

Exercice 4
- Proposer une fonction itérative pour la résolution du problème de *Fibonacci*.
- Tester la fonction dans un programme principal.

Exercice 5
Ecrire un programme en utilisant une fonction qui détermine la plus grande et la plus petite valeur dans un tableau d'entiers **T**. Afficher ensuite la valeur et la position du maximum et du minimum. Si le tableau contient plusieurs maxima ou minima, le programme retiendra la position du premier maximum ou minimum rencontré.

Corrigé
Partie I. Les fonctions

Exercice 1
```
/*PROGRAMME AVEC VARIABLES LOCALES DANS LA FONCTION main*/
#include <stdio.h>
int add (int a, int b)
{
return a+b;
}
main ()
{
int x,y;
scanf ("%d %d", &x,&y);
printf ("la somme de %d %d est %d", x,y,add (x,y));
}

/*PROGRAMME AVEC VARIABLES GLOBALES */
#include <stdio.h>
int a,b;
int add (  )
{
return a+b;
}
main ()
{
scanf ("%d %d", &a,&b);
printf ("la somme de %d %d est %d", a,b,add ( ));
```

```
}
```

Exercice 2

```
#include <stdio.h>
int N;
int diviseur(int nbr)
{
int i,compt,som;
for (som=0,compt=0,i=1;i<=N;i++)
if (N % (i) == 0) {
                compt++;
                printf ("diviseur %d est %d\n",compt,i);
                som=som+i;
                }
return (som);
}

main()
{
printf ( "Entrer un nombre: " );
scanf("%d",&N);
printf ("la somme des diviseurs est %d\n",diviseur(N));
}
```

Exercice 3

```
#include <stdio.h>
int fibonacci_rec(int n)
{
if (n==1 || n==2) return 1;
else return fibonacci_rec(n-1)+fibonacci_rec(n-2);
}

main()
{int n;
printf ("donnez le nombre de fibo");
scanf ("%d",&n);
printf ("fibonnaci (%d)=%d \n",n,fibonacci_rec (n));
}
```

Partie 2. Les pointeurs

Exercice 1

```
#include <stdio.h>
main ()
{
int age= 18;
int *p=NULL; // créer un pointeur entier
p=&age; // le pointeur contient l'adresse de la variable
printf ("La variable val contient la valeur  %d \n ", age);
printf ("Adresse de la variable  = %p  \n", &age) ;
printf ("Le pointeur contient l'adresse %p \n", p) ;
printf ("Le pointeur contient la valeur %d \n", *p) ;
}
```

Exercice 2

Correction du programme:

```
#include <stdio.h>

void calcul (int *a, int b){
*a=2* *a; int c = *a+b;
}

main ()
{
int x,y;
printf ("donnez une valeur x:  " ); scanf ("%d",&x);
printf ("donnez une valeur y:  " ); scanf ("%d",&y);
calcul(&x,y);
printf ("les nouvelles valeurs aprés appel: x=%d, y=%d",x,y);}
```

Exercice 3

```
#include <stdio.h>
#define TAILLE 100


void LIRE_TAB (int *TAB, int *N, int NMAX)
{
/* Variables locales */
int i;
/* Saisie de la dimension du tableau */
do{
printf("Dimension du tableau (max.%d) : ", NMAX);
scanf("%d", N);
}
while (*N<=0 || *N>NMAX);

/* Saisie des composantes du tableau */
for (i=0; i<*N; i++)
{
printf("Elément[%d] : ", i);
scanf("%d", &TAB[i]);
}
}

void ECRIRE_TAB (int *TAB, int N)
{
int i ;
/* Affichage des composantes du tableau */
for (i=0; i<N; i++)
printf("%d ", *(TAB+i));
printf("\n");
}

void ECHANGE(int *a, int *b)
```

```
{
int tmp;
tmp=*a;
*a=*b;
*b=tmp;
}

void INVERSE_TAB(int *TAB, int N)
{
/* Variables locales */
int i,j;
for (i=0, j=N-1 ; i<(N)/2 ; i++){
/* Echange de TAB[i] et TAB[j] */
ECHANGE(TAB+i,TAB+j);j--;
}
}

main()
{
/* Variables locales */
int T[TAILLE]; /* Tableau d'entiers */
int DIM; /* Dimension du tableau */
/* Traitements */
LIRE_TAB (T, &DIM, TAILLE);
printf("Tableau donné : \n");
ECRIRE_TAB (T, DIM);
INVERSE_TAB(T, DIM);
printf("Tableau inversé : \n");
ECRIRE_TAB (T, DIM);
}
```

TP3

But
Initier l'étudiant au concept de l'allocation dynamique et des listes chainées.

Exercice 1

- Ecrire une fonction qui alloue dynamiquement un tableau d'entiers.
- Ecrire une fonction qui saisi les valeurs du tableau.
- Ecrire une fonction qui affiche les valeurs du tableau.
- Ecrire une fonction qui recherche le nombre d'occurrence d'une valeur dans le tableau.
- Tester vos fonctions dans la fonction main.

Exercice 2
Donner les déclarations nécessaires qui permettent de créer une liste chaînée comprenant :

- Deux champs : Le champ ***id*** de type entier et le champ ***moyenne*** de type réel.
- Pointeur ***next*** pour pointer sur le maillon suivant.

Ecrire :

- Une fonction qui permet d'ajouter un maillon à la tête d'une liste.
- Ecrire une fonction qui affiche les éléments d'une liste passée comme paramètre.
- Ecrire une fonction qui retourne le nombre des moyennes supérieures ou égale à 10.
- Ecrire une fonction qui cherche un identificateur *(id)* dans une liste et retourne son pointeur.
- Ecrire une fonction qui supprime un identificateur *id* de la liste.
- Ecrire une fonction *menu ()*, afficher un menu qui présente les traitements à réaliser sous forme d'options de choix.

  MENU PRINCIPAL
  1 - Ajout un élément à la liste.
  2 - Afficher la liste.
  3 - Supprimer un identificateur de la liste
  4 – Affiche le nombre des moyennes supérieures ou égales à 10.
  5 - Quitter.

  Taper votre choix :

- Tester vos fonctions.

Exercice supplémentaire

- Ecrire une fonction qui alloue dynamiquement une matrice réelle.
- Ecrire une fonction qui saisi les valeurs de la matrice.
- Ecrire une fonction qui affiche la matrice.
- Ecrire une fonction qui retourne la somme des valeurs de la matrice.

Corrigé
Exercice 1

```
#include<stdio.h>
#include<stdlib.h>
int i;
/*Allocation dynamique*/
int *allouevecteur(int dimension)
```

```
{
int *vecteur = malloc(dimension*sizeof(int));
return vecteur;
}
/*Affichage du vecteur*/
void affiche(int *vecteur,int dimension)
{
for (i=0;i<dimension;i++)
    printf ("%d \t",*(vecteur+i));
}
/*Saisi du vecteur*/
void saisi(int *vecteur,int dimension)
{
for (i=0;i<dimension;i++){
    printf ("case %d= ",i);
    scanf ("%d= ",vecteur+i);
    }
}
/*Rechercher*/
int occurence (int *vecteur,int dimension, int valeur)
{
int occ;
for (i=0,occ=0;i<dimension;i++)
    if (valeur==*(vecteur+i)) occ++;

return occ;

}
/*Liberer*/
void liberevecteur(int *vecteur)
{
free(vecteur);
}

main(){
int *vecteur=NULL;
int dim;
int valeur;
printf ("La dimension de mon tableau est: ");
scanf("%d",&dim);
vecteur=allouevecteur(dim);
printf ("La saisi du tableau \n");
saisi(vecteur,dim);
printf ("Affichage du tableau saisi \n");
affiche(vecteur,dim);
printf ("\nLa valeur à rechercher est: ");
scanf("%d",&valeur);
printf ("Le nombre d'occurence de la valeur %d est %d
\n",valeur, occurence(vecteur, dim,valeur));
liberevecteur(vecteur);
}
```

Exercice 2

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

struct liste {
int id;
float moyenne;
struct liste *next;
};
typedef struct liste maliste;
maliste *tete=NULL;

/* fonction qui ajoute un élément à la liste */
maliste * ajouter(maliste *tete,int valeur, float moy)
{
maliste *elem = malloc(sizeof(maliste));
if (elem == NULL)
exit(0);
elem->id =valeur;
elem->moyenne= moy;
if(tete==NULL) elem->next=NULL;
else elem->next=tete;
tete=elem;
return tete;
}

/* fonction qui affiche les elements de la liste */
void afficher(maliste *tete)
{
maliste *pp;
printf("Affichage de la liste\n");
for(pp=tete;pp!=NULL;pp=pp->next)
{
    printf("Identificateur %d\t",pp->id);
    printf("Moyenne %f\n",pp->moyenne);
}
}
/* fonction qui retourne le nombre de moyenne >=10 */
int Nombre(maliste *tete)
{
int nb;
maliste *pp;
for(pp=tete,nb=0;pp!=NULL;pp=pp->next){
if (pp->moyenne>=10) nb++; else pp->next;}
return nb;
}

/* fonction qui retourne un pointeur sur un valeur recherchée
(id) */
maliste* rechercher(maliste* tete,int valeur)
{
maliste *pp;
for(pp=tete;pp!=NULL;pp=pp->next)
```

```
if(pp->id==valeur) return pp;
return NULL;
}

/*fonction qui supprime une valeur de la liste (id)*/
maliste* supprimer(maliste *tete, int valeur)
{
maliste* pp,*pp1;
pp=rechercher(tete,valeur);
if (pp!=NULL){
if (pp==tete){ tete=tete->next;
               free(pp);
            }
else{ pp1=tete;
       while(pp1->next!=pp) pp1=pp1->next;
       pp1->next=pp->next;
       free(pp);
}
}
else printf("Identificateur introuvable\n");
return tete;
}


/*fonction menu principal*/
void menu()
{
printf("-----------------------------------------\n");
printf("\tMENU PRINCIPAL\n");
printf("1 - Ajout un élément à la liste.\n");
printf("2 - Afficher la liste.\n");
printf("3 - Supprimer un identificateur de la liste.\n");
printf("4  -  Affiche  le  nombre  des  moyennes  >10  de  la
liste\n");
printf("5 - Quitter.\n");
printf("Taper votre choix: ");
}


main()
{
char choix;
int code;
float moyenne;
maliste* pp,*pp1;

do{
menu();
choix=getchar();
switch(choix){
case       '1':{printf("Donner       un       identificateur:
");scanf("%d",&code);
printf("\nDonner la moyenne ");scanf("%f",&moyenne);
```

```
tete=ajouter(tete,code,moyenne);break;}
case '2':{afficher(tete);break;}
case   '3':{printf("Taper   l'identificateur   à   supprimer=
");scanf("%d",&code);
tete=supprimer(tete,code);break;}
case      '4':{printf("le      nombre      des      moyennes
>=10=%d\n",Nombre(tete));break;}
case '5':break;
default: printf("Choix erroné\n");
}
getchar(); /* pour lire le saut de ligne du premier getchar
*/
}while(choix!='5');

}
```

REFERENCES BIBLIOGRAPHIQUES

P. Zanella , Y. Ligier, *« Architecture et technologie des ordinateurs »*, DUNOD, ISBN 2-04-018795-2, Paris 1989.

R. Malgouyres, R. Zrour, F. Feschet, *« Initiatiation à l'algortihmique et à la programmation C »*, 2eme edition, DUNOD, ISBN 978-2-10-055903-9 , 2011.

N. Flasque, H. Kassel, F. Lepoivre, B. Velikson, *« Exercices et problémes d'algorithmique »*, DUNOD, ISBN 978-2-10-055072-2,2010.

B.W. Kernighan, D. Ritchie, *« Le langage C, Norme ANSI »,* DUNOD, ISBN 2 100051164, Paris 2000.

L. Baba-Hamed, S. Hocine, *« Algorithme et structures de données statiques»,* Cours et exercices avec solutions, Edition numéro 4494.