Université Mohammed V - Agdal
Ecole Mohammadia d'Ingénieurs
Département Informatique
Rabat

Algorithmes et langage C

Plan du cours:

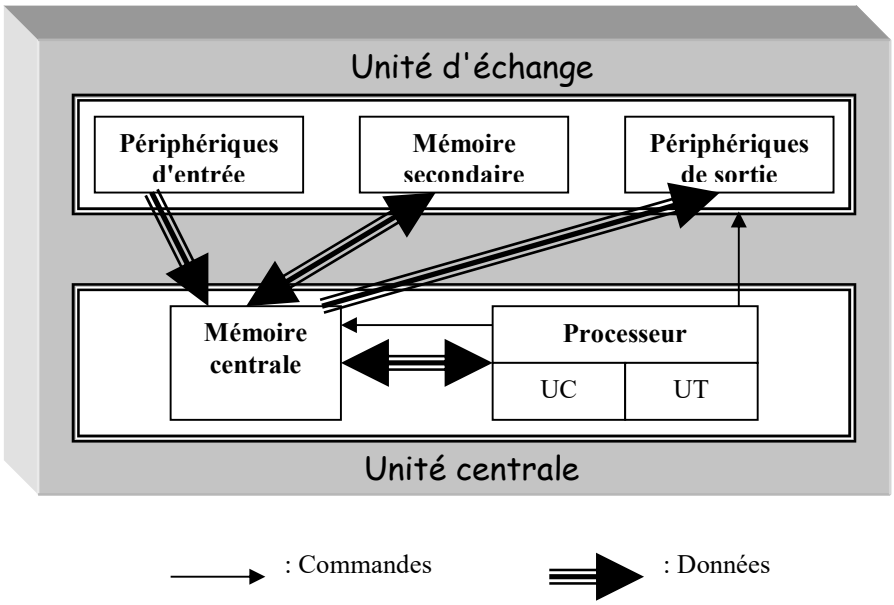
Madame Fatima-Zahra BELOUADHA **Première année, première période**

Sommaire

CHAPITRE 1 INTRODUCTION

L'informatique est la science du traitement automatique (moyennant l'ordinateur) de l'information. Elle a pour objet d'élaborer et de formuler l'ensemble de commandes, d'ordres ou d'instructions permettant de commander l'ordinateur et de l'orienter lors du traitement.

1 ARCHITECTURE ET COMPOSANTS MATERIELS DE L'ORDINATEUR



Un ordinateur est composé de deux unités :

1- L'unité centrale constituée de :

- ***L'unité de traitement*** (UT) qui commande tout traitement fait par l'ordinateur.
- ***L'unité de calcul*** (UC) qui effectue les opérations (arithmétiques, logiques…) commandées par l'UT. L'ensemble UT, UC est appelé **processeur**.
- ***La mémoire centrale*** qui sert de support de stockage de données. On signale ici qu'il s'agit d'une mémoire *volatile*.

2- L'unité d'échange constituée de **:**

- ***Les périphériques d'entrée/sortie*** comme le clavier, la souris, l'écran et l'imprimante.
- ***La mémoire secondaire*** qui sert également de support de stockage de données. Elle est permanente et se caractérise par une capacité supérieure à celle de la mémoire centrale.

Remarque :

Le composant mémoire est physiquement un ensemble de cellules mémoire (**octets**) contenant des données sous forme **binaire**. Un octet est constitué de 8 bits (digits contenant les chiffres 0 ou 1). Un kilooctet (KOctet) est composé de 1024 (2^{10}) octets.

2 COMPOSANTS LOGICIELS DE L'ORDINATEUR

Tout traitement automatique peut être réalisé au moyen d'un ou de plusieurs programmes. Un programme est une série d'instructions (opérations) que la machine peut exécuter pour effectuer des traitements donnés.
Un logiciel est en général un ensemble de programmes visant à effectuer automatiquement un traitement ou une tâche complexe.
Une machine peut héberger plusieurs logiciels au choix de son utilisateur. Cependant, un **système d'exploitation** dit aussi système opératoire est un **logiciel de base** qui doit faire l'objet de la première installation.
Un système d'exploitation fut un **ensemble de programmes assurant** d'une part, le **fonctionnement de toutes les composantes matérielles** d'un ordinateur et d'autre part, **la communication Homme/Machine**. Il a pour exemples de fonctions :

- La gestion de la mémoire.
- La gestion des périphériques.
- La gestion de partage de ressources entre plusieurs utilisateurs.
- Système de fichiers.
- Interface utilisateur.

Comme exemples de systèmes opératoires, nous citons *Windows, Unix, Linux, Ms Dos, MacOs...*

3 SCENARIO D'UN TRAITEMENT AUTOMATIQUE

Faire effectuer un traitement automatique par la machine nécessite de lui indiquer la source de données (sur quoi porte le traitement), les opérations ou actions élémentaires à effectuer pour atteindre l'objectif visé et la destination où elle doit renvoyer les résultats. L'ensemble de ces informations constitue ce qu'on appelle un algorithme que le programmeur doit encore traduire en **programme exécutable** par la machine.
L'exécution d'un programme par l'ordinateur passe, en général, par les étapes suivantes :

1- Le processeur **extrait les données** à traiter à partir de la source indiquée dans le programme (soit auprès de l'utilisateur qui devrait les introduire au moyen du clavier, soit en mémoire secondaire ou centrale).
2- Il **exécute**, ensuite, la série d'opérations élémentaires de manière séquentielle (dans l'ordre prévu par le programme) et **mémorise** tous les résultats intermédiaires.
3- Il **renvoie** enfin le ou **les résultats** attendus à la destination (périphérique de sortie) indiquée dans le programme.

Exemple :
Pour calculer le montant total d'une facture de téléphone pour des appels locaux effectués le soir, le programmeur doit préciser au processeur :

- les données (les entrées consommation, prix_unitaire, la TVA et le prix_d_abonnement) à demander à l'utilisateur.
- l'ordre des opérations à faire et les résultats intermédiaires à mémoriser (dans cet exemple, il s'agit de calculer respectivement le prix hors taxe (PHT) et le prix total (PT).
- Le résultat final (PT) à afficher.

Algorithme :
Lire (consommation, prix_unitaire,TVA, prix_d_abonnement)
PHT← (consommation * prix_unitaire) + prix_d_abonnement
PT ← PHT * (1+TVA)
Ecrire PT

Dans le cas où on donne une consommation de 100 unités avec 0.50 Dh comme prix unitaire, 0.2 comme taux de la TVA et 70 Dh comme prix d'abonnement, le processeur, après avoir demandé les données (100, 0.50, 0.2 et 70) à l'utilisateur, calcule dans l'ordre indiqué chaque expression élémentaire et mémorise son résultat :

PHT=(100*0.5)+70=120

PT=120*(1+0.2)=144

Il affiche enfin le résultat final PT (144).

4 LANGAGE DE PROGRAMMATION

On vient de voir que pour pouvoir effectuer un traitement donné, la machine doit disposer du programme exécutable correspondant. Ce programme doit se trouver en mémoire et doit alors être codé en binaire (langage machine).

Un langage de programmation permet au programmeur d'écrire son programme suivant une grammaire qui peut être, soit celle du langage machine même, soit une grammaire facilement interprétable par la machine ou pouvant être traduite en langage machine au moyen d'un outil logiciel dit compilateur du langage.

Il existe, en fait, **trois catégories de langages** :

- **Le langage binaire :** il s'agit du langage machine exprimé par des chiffres (**0 ou 1**). Ce langage produit, en effet, des programmes ***automatiquement consommables*** (compréhensibles) par la machine mais qui sont illisibles et ***non portables***.
- **Les langages de bas niveau (comme l'assembleur) :** ces langages produisent des programmes ***facilement interprétables*** par la machine mais ***d'utilisation lourde*** pour les programmeurs.
- **Les langages évolués :** ils sont, d'utilisation, souples et produisent des programmes clairs et ***lisibles*** mais ils ***doivent encore être compilés*** (traduits en langage machine par un compilateur du langage) pour générer des programmes exécutables. Nous en citons: *Fortran, Basic, Pascal, C, C++, Visual Basic, Visual C++, Java...*

Exemple :

Ce programme écrit en C calcule le montant de la facture de téléphone

```
#include <stdio.h>
main( )
{
        int consommation, prix_d_abonnement ;
        float prix_unitaire, TVA, PT ;
        printf("Entrer la valeur de la consommation :");
        scanf("%d",&consommation) ;
        printf("Entrer la valeur du prix unitaire :");
        scanf("%f",&prix_unitaire) ;
        printf("Entrer la valeur de la TVA :");
        scanf("%f",&TVA) ;
        printf("Entrer la valeur du prix abonnement :");
        scanf("%d",&prix_d_abonnement) ;
        PHT=(consommation * prix_unitaire) + prix_d_abonnement ;
        PT=PHT * (1+TVA) ;
        printf("Le prix total est de %f  DH\n",PT);
}
```

5 COMPILATION ET EXECUTION D'UN PROGRAMME EN C

Générer un programme exécutable à partir d'un programme source (écrit en C et dont le nom de fichier se termine nécessairement par l'extension .c) consiste à faire appel au compilateur et éditeur de lien du langage moyennant la **commande Unix cc**.

- Le **compilateur** traduit le programme source en un **fichier objet** (qui porte l'extension.o).
- **L'éditeur de liens** génère pour les différents fichiers objet composant le programme un fichier **exécutable**.

1- Utilisation de la commande sans option

Syntaxe :

cc *nom-fichier-source*

l'exécutable porte, par défaut, le nom **a.out**

Exemples :

Compilation par cc

cc tp1.c *compile le fichier source tp1.c*
cc pg1.c pg2.c *génère l'exécutable a.out pour les fichiers source pg1.c et pg2.c*

2- Utilisation de la commande avec l'option -o

Syntaxe :

cc -o *nom_fichier_exécutable nom_fichier_source*

l'exécutable peut porter un nom différent du nom de fichier source au choix de l'utilisateur.

Exemples :

Compilation par cc -o

cc -o somme somme.c *crée l'exécutable somme à partir de somme.c*
cc -o calculette somme.c division.c soustraction.c multiplication.c
crée l'exécutable calculette à partir de plusieurs fichiers .c

Remarque :

Pour **exécuter** un programme, il suffit de **taper le nom** de son exécutable.

CHAPITRE 2 NOTIONS D'ALGORITHMES

1 QU'EST CE QU'UN ALGORITHME ?

Le terme algorithme est employé en informatique pour **décrire une méthode de résolution de problème programmable sur machine.**
Un algorithme est une **suite finie et ordonnée d'opérations** (actions) élémentaires finies (en temps et moyens). Elle est régie par un ensemble de **règles ou d'instructions de contrôle** (***séquencement***, ***sélection*** et ***itération***) permettant d'aboutir à un résultat déterminé d'un problème donné.

Exemple 1:

Creuser un trou, reboucher un trou et placer un arbre dans un trou sont des opérations élémentaires (des actions simples) que toute personne (machine dans le cas de l'informatique) est censée savoir exécuter. Néanmoins, si un jardinier (programmeur) veut faire planter un arbre par une personne qui ne sait pas le faire, il doit lui fournir "un descriptif" (un algorithme) qui lui indique les opérations à faire ainsi que leur ordre d'exécution (séquencement).

Algorithme de plantation d'un arbre

1- Creuser un trou.
2- Placer l'arbre dans le trou.
3- Reboucher le trou.

Exemple 2:

Pour planter et arroser un ensemble d'arbres, on peut procéder de la manière suivante: planter l'ensemble d'arbres et les arroser tous à la fin.

Algorithme de plantation et d'arrosage de plusieurs arbres

1- Creuser un trou.
2- Placer un arbre dans le trou.
3- Reboucher le trou.
4- S'il existe encore des arbres exécuter les actions 1, 2, 3 et 4.
   Sinon exécuter les actions suivantes.
5- Arroser les arbres.

L'algorithme correspondant indique un ensemble d'opérations (1, 2, 3 et 4) à répéter un certain nombre de fois (*règle d'itération*) et l'opération ou les opérations (5 ou 1, 2, 3 et 4) à exécuter selon qu'il reste ou non des arbres à planter (*règle de sélection*).

Remarque:

Pour résoudre le problème précédent, on peut procéder autrement : planter et arroser arbre par arbre. On conclut alors qu'**à un problème donné pourraient correspondre plusieurs algorithmes**.

2 OPERATIONS DE BASE

Ce paragraphe décrit la liste des opérations de base pouvant composer un algorithme. Elles sont décrites en ***pseudocode (pseudolangage)***. Il s'agit d'un langage informel proche du langage naturel et indépendant de tout langage de programmation.
Les données manipulées dans un algorithme sont appelées des ***variables***.

2.1 L'AFFECTATION

Variable ← *expression*

L'affectation, notée par le symbole ←, est l'opération qui évalue une expression (constante ou une expression arithmétique ou logique) et attribue la valeur obtenue à une variable.

Exemples d'affectation

a ← 10	a reçoit la constante 10
a ← (a*b)+c	a reçoit le résultat de (a*b)+c
d ← 'm'	d reçoit la lettre m

2.2 LA LECTURE

Lire *variable*

Cette opération permet d'attribuer à une variable une valeur introduite au moyen d'un organe d'entrée (généralement le clavier).

Exemples de lecture

Lire *a*	On demande à l'utilisateur d'introduire une valeur pour a
Lire *(a,b,c)*	On demande à l'utilisateur d'introduire 3 valeurs pour a, b et c respectivement

Remarque :

L'organe d'entrée est assimilé à un ruban composé d'une suite de cases chacune peut contenir un caractère ou un chiffre.
C'est la valeur qui se trouve à la tête du ruban qui sera attribuée à la variable à lire. Une fois, la lecture est terminée, elle est supprimée du ruban.

2.3 L'ECRITURE

Ecrire *expression*

Elle communique une valeur donnée ou un résultat d'une expression à l'organe de sortie.

Exemples d'écriture

Ecrire *'bonjour'*	Affiche le message bonjour (constante)
Ecrire *12*	Affiche la valeur 12
Ecrire *a,b,c*	Affiche les valeurs de a, b et c
Ecrire *a+b*	Affiche la valeur de a+b

2.4 INSTRUCTIONS DE CONTROLE

2.4.1 INSTRUCTIONS SELECTIVES

a) Instruction Si

```
Si  condition  alors
            Début
            instruction(s) 1
      Fin
Sinon
      Début
            instruction(s) 2
      Fin
```

Elle indique le traitement à faire selon qu'une condition (expression logique) donnée est satisfaite ou non.
Il est possible d'omettre début et fin si le bloc d'instructions à exécuter est constitué d'une seule instruction.

Exemple :

Calculer la taxe sur le chiffre d'affaire (CA) sachant qu'elle est de :

- 10% si le CA<5000DH
- 20% si le CA >=5000DH

Exemple d'algorithme utilisant l'instruction Si

```
lire CA
Si CA<5000 alors
      Taxe←CA*10%
Sinon
      Taxe←CA*20%
Ecrire Taxe
```

Remarques:

1- Une instruction de contrôle peut se **réduire** à :

```
Si condition alors
      Début
            instruction(s)
      Fin
```

2- On peut avoir plusieurs **si imbriqués** comme:

```
Si condition1 alors
      Si condition2 alors
            Si condition3 alors
                  <instruction(s) 1>
            Sinon
                  <instruction(s) 2>
      Sinon
            <instruction(s) 3>
Sinon
      <instruction(s) 4>
```

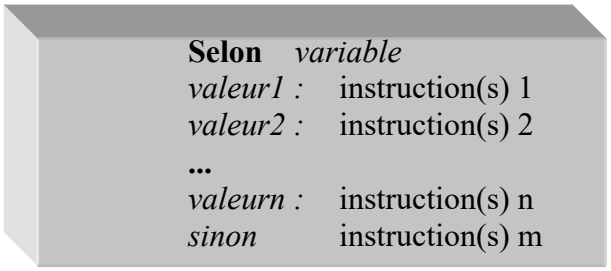
Exemple:

Vérifier si un agent a droit de bénéficier du crédit de logement ou non selon ses années d'ancienneté et sa note d'appréciation.

Exemple d'algorithme utilisant des si imbriqués

```
lire (ancienneté, note)
Si ancienneté < 5 alors
        Si ancienneté=4 et note≥16 alors
                écrire ('L'agent a droit de bénéficier du crédit')
        Sinon
                Si ancienneté=3 et note≥18 alors
                        écrire ('L'agent a droit de bénéficier du crédit')
                Sinon
                        écrire ('L'agent n'a pas droit de bénéficier du crédit')
Sinon
        Si note≥13 alors
                écrire ('L'agent a droit de bénéficier du crédit')
        Sinon
                écrire ('L'agent n'a pas droit de bénéficier du crédit')
```

b) Instruction Selon



Elle indique le traitement à faire selon la **valeur d'une variable**.

Exemple:

Vérifier et Afficher si un caractère saisi est une voyelle ou consonne.

Exemple d'algorithme utilisant l'instruction Selon

```
lire c
Selon c
        'a' :       écrire ('le caractère est une voyelle')
        'e' :       écrire ('le caractère est une voyelle')
        'i' :       écrire ('le caractère est une voyelle')
        'o' :       écrire ('le caractère est une voyelle')
        'u' :       écrire ('le caractère est une voyelle')
        'y' :       écrire ('le caractère est une voyelle')
        sinon       écrire ('le caractère est une consonne')
```

2.4.2 INSTRUCTIONS ITERATIVES

a) Instruction Pour

```
Pour compteur ← valeur1 à valeurn faire
    Début
        Instruction(s)
    Fin
```

Elle permet de **répéter** un traitement **un nombre de fois précis** et connu en utilisant un compteur (variable à incrémenter d'une itération à l'autre).

Exemple :

Afficher la somme des entiers compris entre 0 et une valeur n saisie au clavier (n≥0).

Exemple d'algorithme utilisant l'instruction Pour

```
lire n
s←0
Pour i←1 à n faire
    Début
        s←s+i
    Fin
écrire s
```

b) Instruction Tant que

```
Tant que condition faire
    Début
        instruction(s)
    Fin
```

Elle permet de **répéter** un traitement tant qu'une **condition** est satisfaite.

Exemple :

Calculer la somme s des entiers compris entre 0 et un nombre n saisi au clavier (on suppose que n≥0).

Exemple d'algorithme utilisant l'instruction Tant que

```
lire n
s←0
Tant que n > 0 faire
    Début
        s←s+n
        n←n-1
    Fin
Ecrire s
```

Remarque :

Dans le cas d'utilisation de l'instruction tant que, si la **condition est fausse au départ**, **le bloc d'instructions ne sera pas du tout exécuté** (Dans l'exemple ci-dessus, c'est le cas où n≤0 au départ).

c) Instruction Faire Tant que

```
Faire
    Début
        instruction(s)
    Fin
Tant que  condition
```

Elle permet de **répéter** un traitement tant qu'une **condition** est satisfaite.

Exemple :

Calculer la somme s des entiers compris entre 0 et un nombre n saisi au clavier.

Exemple d'algorithme utilisant l'instruction Faire Tant que

```
lire n
s←0
i ←0
Faire
    Début
        s←s+i
        i←i+1
    Fin
Tant que  i≤n
Ecrire s
```

Remarque :

Dans le cas d'utilisation de l'instruction Faire Tant que, **le bloc d'instructions est exécuté au moins une fois quelle que soit la condition**.
Reprenons l'exemple précédent sous une autre forme (On initialise cette fois-ci i à 1) :

Exemple d'algorithme utilisant l'instruction Faire Tant que

```
lire n
s←0
i ←1
Faire
    Début
        s←s+i
        i←i+1
    Fin
Tant que  i≤n
Ecrire s
```

Dans ce cas, si on suppose que n est égal à 0, le programme affichera 1 comme somme. Ce qui est incorrect. Par ailleurs, l'addition de i à la somme ne doit être effectuée que si i est inférieur ou égal à n, condition qui n'est vérifiée avec l'usage de l'instruction « Faire Tant que » que plus tard. Alors qu'on commence la boucle avec i égal à 1 qui est strictement supérieur à n, 0 dans cet exemple.

3 EXEMPLE : CONVERSION EN BASE

Réaliser un algorithme qui affiche le résultat de conversion d'un nombre entier positif strictement dans une base quelconque (comprise entre 2 et 10). Le résultat doit être affiché commençant par le bit0.

Entrées : Le nombre n et la base b.
Sorties : Le résultat de conversion r.
Traitement : Convertir le nombre n en base b.
Cas particuliers : $n \leq 0$, $b < 2$ et $b > 10$.

Version 1 de l'algorithme :

Début
1- Lire n et b
2- Convertir le nombre n en base b
3- Ecrire r
Fin

Analyse par l'exemple :

Considérant le cas par exemple de n=11 et b=2, la résolution manuelle de ce problème aide à dégager les actions principales composant la solution algorithmique du cas général.

1- 11/2=5 reste=1
2- 5/2=2 reste=1
3- 2/2=1 reste=0
4- 1/2=0 reste=1 (on ne peut pas continuer) $\Rightarrow$ r=1011

Nous remarquons que :

- Il s'agit d'effectuer une succession de division de n par b, donc un traitement à répéter autant de fois qu'il le faut.
- Le résultat est la combinaison des différents restes obtenus commençant par le bit 0.

Version 2 de l'algorithme :

Début
1- Lire n et b
2- Effectuer une succession de division de n par b autant de fois qu'il le faut
3- Le résultat est la combinaison des différents restes obtenus à écrire commençant par le bit 0.
Fin

Essayons maintenant de détailler les actions principales en actions plus simples.

Version 3 de l'algorithme :

```
Début
1-  Lire (n,b)
2-  Tant qu'il le faut (condition à chercher) faire
        Début
3-          Diviser n par b pour calculer le reste
4-          Ecrire le reste
        Fin
Fin
```

Version 4 de l'algorithme :

```
Début
1-  Lire (n,b)
2-  Tant que le résultat de la division est différent de 0 faire
        Début
3-          r ← n mod b
4-          n ← n div b
5-          Ecrire r
        Fin
Fin
```

Version 5 de l'algorithme :

```
Début
1-  Lire (n,b)
2-  Tant que n ≠ 0 faire
        Début
3-          r ← n mod b
4-          n ← n div b
5-          Ecrire r
        Fin
Fin
```

Version 6 de l'algorithme : (On introduit les cas particuliers)

```
Début
1-  Faire
        Lire n
    Tant que n ≤ 0
2-  Faire
        Lire b
    Tant que b < 2 ou b > 10
3-  Tant que n ≠ 0 faire
        Début
4-          r ← n mod b
5-          n ← n div b
6-          Ecrire r
        Fin
Fin
```

Pour valider l'algorithme obtenu, On peut dans un premier temps appliquer sa trace à un exemple au choix. Il s'agit de prendre la place de la machine pour effectuer manuellement les actions prescrites.

Trace de l'algorithme pour n=11 et b=2 :

Opération	n	b	r
1	11	-	-
2	11	2	-
3	11	2	-
4	11	2	1
5	5	2	1
6	5	2	1 (1er affichage)
3	5	2	1
4	5	2	1
5	2	2	1
6	2	2	1 (2ème affichage)
3	2	2	1
4	2	2	0
5	1	2	0
6	1	2	0 (3ème affichage)
3	1	2	0
4	1	2	1
5	0	2	1
6	0	2	1 (4ème affichage)
3	0	2	1

Le résultat final est correct pour le cas n=11 et b=5. On obtient bien 1011 en commençant par les unités. Néanmoins, il reste à valider l'ensemble de cas possibles selon les différentes valeurs que peuvent prendre les entrées n et b :

Cas où $n \leq 0$ et b quelconque, le programme fonctionne correctement (ce cas sera écarté selon l'énoncé).

Cas où $b<2$ ou $b>10$ et n quelconque, le programme fonctionne correctement (ce cas sera écarté selon l'énoncé).

Cas où $n<b$, le programme fonctionne correctement. Il s'arrête après la première itération avec comme résultat r=n

Cas où $n \geq b$, Après une succession finie de division de n par b, n devient nul; D'où l'arrêt de la boucle. On constate alors que le programme s'arrête au bout d'un temps fini. Néanmoins, la pertinence des résultats ne peut être confirmée. Le programmeur doit alors écrire le programme correspondant à l'algorithme obtenu, le compiler et l'exécuter pour effectuer un jeu d'essai.

Exercice :

Ecrire l'algorithme de la multiplication russe qui consiste à calculer le produit de deux entiers a et b en procédant par une succession parallèle du calcul du double de b et la moitié de a jusqu'à ce que la moitié devient égale à 1. Le produit résultant est la somme des doubles obtenus correspondant à des moitiés impaires.

Exemple : a=19 et b=9

a ____19 9 4 2 1 ____sur l'axe a, on calcule des moitiés

b ____9 18 36 72 144____sur l'axe b, on calcule des doubles

Le résultat est 171, somme des doubles 9, 18 et 144 correspondant aux moitiés impaires 19, 9 et 1.

CHAPITRE 3 CONCEPTS DE BASE DU LANGAGE C

En 1970, Dennis RITCHIE a créé le langage C, un langage de haut niveau, pour écrire le système d'exploitation Unix. La conception de ce langage a été régie par les pré requis suivants :

- la souplesse
- la fiabilité
- la portabilité
- les possibilités de l'assembleur

1 STRUCTURE D'UN PROGRAMME EN C

1.1 UN PREMIER PROGRAMME EN C

Exemple :

Ce programme affiche le message bonjour

```
main( )
{
        printf("bonjour la première année");
}
```

main () indique qu'il s'agit du **programme principal**.

{ et } jouent le rôle de **début et fin** de programme.

Printf est **l'instruction d'affichage** à l'écran, le message étant entre guillemets.

; indique la **fin d'une instruction**.

Remarque :

Une représentation claire et aérée du programme, avec **indentation sensée**, améliore la **lisibilité**. Elle est vivement conseillée.

1.2 STRUCTURE GENERALE D'UN PROGRAMME EN C

Un programme en C se présente en général sous la forme suivante :

```
<Directives de compilation>
<Déclaration de variables externes>
<Déclaration de prototypes de fonctions>
main ( )
{
        corps du programme
        (commentaires,
        déclaration de variables et constantes,
        instructions)
}
<Définition de fonctions>
```

Remarque :

Un programme en C doit contenir **au moins le programme principal** (la partie main).

Nous nous intéressons dans ce chapitre aux directives de compilation et différents éléments de base composant le corps du programme.

2 LES DIRECTIVES DE COMPILATION

Nous en citons les directives **include** et **define**:

Syntaxe :

#include < *nom_fichier*>

indique le ***fichier de bibliothèque C*** à inclure dans le programme. Ce fichier contient les ***définitions de certaines fonctions prédéfinies*** utilisées dans le programme.

Syntaxe :

#define *expression_à_remplacer* *expression_de_remplacement*

permet de ***remplacer un symbole par*** une ***constante*** ou un ***type*** ou de faire des substitutions avec arguments dans le cas des ***macros***.

Exemples :

Directives de compilation

```
#include <stdio.h>          /*ce fichier contient les fonctions d'entrées/sorties comme
                              printf*/
#define pi 3.14             /*pi sera remplacé par la valeur 3.14*/
#define entier int          /*entier sera remplacé par le type prédéfini int*/
#define somme(x,y) x+y      /*la macro somme(x,y) sera remplacée par x+y*/
```

3 LES COMMENTAIRES

Un commentaire est un texte placé entre les signes /* et */. Il permet de commenter une ou plusieurs lignes de commandes en vue d'éclairer le lecteur.

Syntaxe :

```
/* texte du commentaire */
```

Exemples :

Commentaires

```
main( )
{
      printf("bonjour");              /* ce programme affiche bonjour*/
}
```

4 LES VARIABLES ET LES CONSTANTES

4.1 DECLARATION DE VARIABLES

A toute variable utilisée dans un programme C doivent être associés d'abord (avant toute utilisation) un nom dit identificateur et un type de données (entier, réel ou caractère…). Lors de l'exécution, une zone mémoire (dont la taille dépend du type) sera réservée pour contenir la variable.

Syntaxe :

Type *identificateur*;

Ou

Type *identificateur1*, *identificateur2*, ... ,*identificateur n*;

Exemples :

Déclaration de variables

```
int i;          /* i est une variable de type entier */
float j,k;      /* j et k sont des variables de type réel */
char c;         /* c est une variable de type caractère */
```

a) Identificateurs

L'emploi des identificateurs doit répondre à un certain nombre d'exigences :

- un identificateur doit être composé indifféremment de lettres et chiffres ainsi que du caractère de soulignement (_) qui peut remplacer des espaces.
- Un identificateur doit commencer par une lettre ou le caractère de soulignement. Néanmoins, celui-ci est souvent utilisé pour désigner des variables du système.
- Seuls les 32 premiers caractères (parfois, uniquement les 8 premiers) sont significatifs (pris en compte par le compilateur).
- Majuscules et minuscules donnent lieu à des identificateurs différents.
- Un identificateur ne doit pas être un mot réservé (utilisé dans le langage C comme int, char, …).

Exemples :

Identificateurs

solution1 est un identificateur **valide** (constitué de lettres et de 1)
1solution n'est pas un identificateur valide.
prix unitaire n'est pas un identificateur valide (Il contient un espace).
prix_unitaire est un identificateur **valide**.
jour, ***Jour*** *et* ***JOUR*** sont 3 identificateurs **différents**
int n'est pas un identificateur valide. C'est un mot utilisé en C

b) Types de données

Un type est un ensemble de valeurs que peut prendre une variable. Il y a des types prédéfinis et des types qui peuvent être définis par le programmeur.

Types simples prédéfinis en C

Type	Signification	Représentation système	
		Taille (bits)	Valeurs limites
int	Entier	16	-32768 à 32767
short (ou short int)	Entier	16	-32768 à 32767
long (ou long int)	Entier en double longueur	32	-2147483648 à 2147483647
char	Caractère	8	
float (ou short float)	Réel	32	$\pm 10^{-37}$ à $\pm 10^{38}$
double(ou long float)	Réel en double précision	64	$\pm 10^{-307}$ à $\pm 10^{308}$
long double	Réel en très grande précision	80	$\pm 10^{-4932}$ à $\pm 10^{4932}$
unsigned	Non signé (positif)	16	0 à 65535

Remarques :

- **int** équivaut à ***short sur PC*** et à ***long sur station***.
- La fonction **sizeof** retourne la **taille en octets** d'un objet.

Exemples :

```
n=sizeof(int);          /* n reçoit 2 */
n=sizeof(3.14);         /* n reçoit 8 */
```

c) fonctions prédéfinies sur les types simples

Des fonctions appliquées au différents types de données sont prédéfinies dans des fichiers de bibliothèque C.

❍ Fonctions mathématiques

Math.h
Ce fichier contient des **fonctions mathématiques** pouvant être appliquées aux types numériques.

Exemples :

Exemples de fonctions mathématiques

```
#include <math.h>     /*pour inclure le fichier math.h*/
main( )
{
int p,i=4,j=-2;       /* p entier et i et j entiers initialisés à 4 et -2*/
float r;              /* r réel*/
p=pow(i,2);           /* p reçoit 16 (4 à la puissance 2) */
r=sqrt (i);           /* r reçoit 2 (racine carrée de 4) */
i=abs(j);             /* i reçoit 2 (valeur absolue de -2)*/
}
```

❍ Fonctions sur les caractères

ctype.h
Ce fichier contient les définitions des **fonctions** pouvant être **appliquées à des caractères**.
Ces fonctions permettent de vérifier si un caractère appartient à une catégorie donnée. Elles retournent 0 si faux et une valeur différente si vrai.

Listes des fonctions sur les caractères

Fonction	Signification
isalpha (c)	c est une lettre
isupper (c)	c est une lettre majuscule
islower (c)	c est une lettre minuscule
isdigit (c)	c est un chiffre
isxdigit (c)	c est hexadécimal [0-9], [A-F] ou [a-f]
isalnum (c)	c est alphanumérique (chiffre ou lettre)
isspace (c)	c est un blanc, tabulation, retour chariot, newline ou formfeed
ispunct (c)	c est un caractère de ponctuation
isprint (c)	c est un caractère imprimable (de 32 (040) à 126 (0176) tilde)
isgraph (c)	c est un caractère imprimable différent d'espace
iscntrl (c)	c est un caractère de contrôle différent d'espace et (<32) ou delete (0177)
isascii (c)	c est un caractère ASCII ($0 \leq c < 128$)

4.2 DECLARATION DE CONSTANTES

Une **constante** est une donnée dont la valeur **ne varie pas** lors de l'exécution du programme. Elle doit être déclarée sous forme :

Syntaxe :

const *Type* *Identificateur* = *Valeur* **;**

Remarque:

Une valeur constante peut, également, être exprimée au moyen d'un identificateur défini en utilisant la directive **define**:

#define *Identificateur* *Valeur*

Exemples :

Déclaration de constantes

```
1-  main( )
    {
    const  float   pi=3.14;          /*déclare la constante pi avec const*/
    printf("pi égale à %f",pi);      /*affiche la valeur de pi*/
    }
2-  #define pi  3.14                 /*définit la constante pi avec define*/
    main( )
    {
    printf("pi égale à %f",pi);      /*affiche la valeur de pi*/
    }
```

4.3 INITIALISATION DE VARIABLES

Une **valeur initiale** peut être affectée à une variable dès la déclaration sous forme :

Syntaxe :

a) Cas de types numériques :

Type *Identificateur* = *Valeur numérique* **;**

b) Cas du type caractère :

Type *Identificateur* = **'***caractère***' ;**

Ou

Type *Identificateur* = *code ASCII d'un caractère* **;**

Exemples :

Initialisation de variables

```
main( )
{
int i, j=3, k;          /* seul j est initialisé à 3*/
float f=1.2 e5;         /* f est initialisé à 120000 (1.2*10^5)*/
int i=011;              /* i est initialisé à 11 en octal soit 9 en décimal*/
char c='A';             /* c est initialisé à la lettre A*/
char c=65;              /* c est initialisé à A dont le code ASCII est 65*/
}
```

Remarque :

- Des **caractères spéciaux** sont représentés à l'aide du métacaractère \.

Exemples :

Initialisation des variables

```
main( )
{
Char c = '\'';          /*c reçoit un apostrophe*/
Char c = '\x041';       /* c reçoit 41 code ASCII en hexadécimal de A*/
}
```

Liste des caractères spéciaux

Représentation	Signification
\0	Caractère NULL
\a	Bip (signal sonore)
\b	Espace arrière
\t	Tabulation
\n	Nouvelle ligne
\f	Nouvelle page
\r	Retour chariot
\"	Guillemet
\'	Apostrophe
\\	Antislash (\)
\ddd	Caractère ayant pour valeur ASCII octale ddd
\x hhh	Caractère ayant pour valeur ASCII hexadécimale ddd

5 L'AFFECTATION (ASSIGNATION)

L'affectation est l'opération qui attribue à une variable, au moyen de l'opérateur =, une valeur constante ou résultat d'une expression.

Syntaxe :

Variable = *Valeur ou expression* **;**

Exemples :

Affectation

```
main( )
{
int i, j;
i=2;              /*i reçoit 2*/
j=(i*3)+5;        /*j reçoit 11 résultat de (2*3)+5*/
i=j=2;            /*j reçoit d'abord 2 puis i reçoit la valeur de j (2)*/
}
```

Remarque :

C permet de faire des **assignations entre des variables de types différents**. Des conversions de types sont alors automatiquement réalisées.

Exemples :

Conversion automatique de types

```
1-   main( )
     {
     int i;              /*i entier*/
     float j=3.5;        /*j réel initialisé à 3.5*/
     i=j;                /*i reçoit 3, le résultat est tronqué*/
     }

2-   main( )
     {
     int i;              /*i entier*/
     char c='A';         /*c caractère initialisé à A*/
     i=c;                /*i reçoit 65 le code ASCII de A*/
     }
```

6 LES ENTREES/SORTIES

6.1 L'AFFICHAGE

L'instruction **printf** permet d'obtenir un affichage formaté à l'écran.

Syntaxe :

a) affichage de message constitué de texte et de caractères de contrôle

Printf ("*texte et caractères de contrôle***") ;**

b) affichage de valeurs de variables ou d'expressions

Printf("*message et formats d'affichage***",** *arg1*, *arg2*,…,*argn***);**

Noms de variables ou expressions

Exemples :

Affichage

```
main( )
{

int i=2,k=5;                          /*i et k entiers initialisés à 2 et 5*/
float j=3.5;                          /*j réel initialisé à 3.5*/

printf("Donnez le prix unitaire");    /*le programme affiche
                                        Donnez le prix unitaire */

printf("Donnez le prix unitaire \n");  /*le programme affiche Donnez le prix
                                         unitaire et retourne à la ligne (\n)*/

printf("la valeur de i est %d\n ",i);  /*le programme affiche la valeur
                                         de i est 2 et retourne à la ligne*/

printf("i=%d      j=%f",i,j);         /*le programme affiche
                                        i=2         j=3.5*/


printf("i=%d\nj=%f",i,j);             /*le programme affiche
                                              i=2
                                              j=3.5*/

printf("somme(%d,%d)=%d\nFIN",i,k,i+k);   /*le programme affiche
                                              somme(2,5)=7
                                              FIN*/

printf("j=%4.2f\n",j);                /*le programme affiche
                                        j=  3.50*/
}
```

Remarques :

- Un **caractère de contrôle est précédé de** \ comme \n qui provoque un interligne (voir liste des caractères spéciaux, paragraphe 2-4).
- Chaque **format d'affichage est introduit par le caractère %** suivi d'un caractère qui indique le type de conversion.
- Des **indications** peuvent être **rajoutées entre le % et le caractère** comme le nombre minimum de caractères réservés à l'affichage de la **mantisse** d'un nombre et le **nombre de décimales**.

Liste des formats d'affichage

Format d'affichage	Signification
%d	Conversion en décimal
%o	octal
%x	hexadécimal (0 à f)
%X	hexadécimal (0 à F)
%u	entier non signé
%c	caractère
%s	chaîne de caractères
%l	long ou double
%L	long double
%e	sous forme m.nnnexx
%E	sous forme m.nnnExx
%f	sous forme mm.nn
%g	Semblable à e ou f selon la valeur à afficher

6.2 LA LECTURE DE DONNEES

L'instruction **scanf** effectue la lecture des variables.

Syntaxe :

scanf("*formats d'affichage***",** *variable1***,** *variable2***,…,***variablen***) ;**

Remarque :

Seules les variables scalaires (entiers, réels et caractères**) doivent être précédées de &**.

Exemples :

Lecture

```
#include <stdio.h>
main( )
{
int i;                          /*i entier*/
float k;                        /*k réel*/
char m;                         /* m caractère*/
scanf("%d",&i);                 /*le programme lit une valeur entière et l'affecte à i*/
scanf("%d%f",&i,&k);            /*le programme lit une valeur entière de i
                                  puis une valeur réelle de k*/
scanf("%c",&m);                 /*le programme lit un caractère et l'affecte à
                                  la  variable m*/
}
```

Remarques :

- La notation **&variable** est utilisée pour indiquer **l'adresse mémoire de la variable** en question.
- Les données tapées au clavier sont d'abord placées dans un **tampon** interne. **Scanf** va chercher ces données dans ce tampon, **sans nécessairement le vider entièrement**. C'est pourquoi, la fonction **scanf** est malheureusement une **source permanente de problèmes** (tampon associé au clavier encombré de **résidus de lectures** précédentes). Elle n'est, en général, acceptable qu'à condition de se limiter à des lectures d'entiers ou de réels.

7 LES OPERATEURS

7.1 LES OPERATEURS ARITHMETIQUES

Les opérateurs arithmétiques traditionnels sont :

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division entière

Exemples :

Opérateurs arithmétiques

```
main( )
{
int i=9,j=4,k;          /*i, j et k entiers*/
float x=9.0, y;         /*x et y réel*/

k=i+j;                  /*k reçoit 13*/
y=i/j;                  /*y reçoit 2.0 (division entière : i et j entiers)*/
y=x/j;                  /*y reçoit 2.25 (division réelle : x réel*/
y=9.0/j;                /*y reçoit 2.25 (division réelle : 9.0 réel*/
k=i%j;                  /*k reçoit 1 (reste de la division de i par j)*/
}
```

Remarques :

- l'opérateur / effectue, en **fonction du type des opérandes**, une **division entière (euclidienne) ou réelle**.
- L'ordre des **priorités** des opérateurs est important.
- Il est possible de **forcer la conversion du type** d'une variable ou d'une expression en les **préfixant d'un type** au choix.

Exemples :

Conversion de types

```
main( )
{
int i=9,j=4;            /*i et j entiers*/
float y;                /*y réel*/
y=(float) i / j;        /*y reçoit 2.25 (i est converti d'abord en réel=9.0)*/
y=(float) (i/j);        /*y reçoit 2.0 (l'entier 2 (i/j) est converti en réel)*/
}
```

7.2 LES OPERATEURS +=, -=, *=, /=

Ils sont utilisés pour faciliter l'écriture.

Ecriture classique	Ecriture équivalente
i = i + 20;	i += 20;
i = i - 20;	i -= 20;
i = i * 20;	i *= 20;
i = i / 20;	i /= 20;

7.3 LES OPERATEURS LOGIQUES

Les opérateurs logiques sont, par ordre décroissant de priorité :

Opérateurs	signification
!	Non logique
> >= < <=	Test de supériorité et d'infériorité
== et !=	Test d'égalité et d'inégalité
&& et ||	ET et OU logique

Exemples :

Opérateurs logiques

```
#include <stdio.h>
main( )
{
int a,b,c;                      /*a,b et c entiers*/
printf ("Introduire a, b et c : ");
scanf ("%d%d%d",&a,&b,&c);

if (a==b && b!=c)               /*si a=b et b≠c affiche le message suivant*/
        printf("a égale à b et b différent de c\n");

if (!(a<b) || a==0)             /*si a>=b ou a=0 affiche le message suivant*/
        printf("a est supérieure ou égale à b ou a égale à 0\n");
}
```

Remarques :

Soit i une variable numérique,

- l'expression **if (i)** est équivalente à l'expression **if (i!=0)**.
- l'expression if **(!i)** est équivalente à l'expression **if (i==0)**.

7.4 LES OPERATEURS ++ ET --

Ils permettent d'incrémenter ou de décrémenter une variable. L'opérateur ++ (--) effectue une **pré-incrémentation (pré-décrémentation)** ou une **post-incrémentation (post-décrémentation)** ***selon son emplacement*** après ou avant la variable.
Dans une opération d'affectation qui met en jeu l'opérateur de :

- **pré-incrémentation** (pré-décrémentation), la variable est **d'abord incrémentée** (décrémentée) de 1. **L'affectation est ensuite** effectuée.
- **post-incrémentation** (post-décrémentation), **L'affectation** (sans les ++ (--)) est effectuée **avant l'incrémentation** (décrémentation).

Exemples :

Soient i=3 et j=5,

Instruction	Equivalent	Résultats
i++;	i=i+1;	i=4
++i;	i=i+1;	i=4
i--;	i=i-1;	i=2
--i;	i=i-1;	i=2
i= ++j;	j=j+1; i=j;	j=6 et i=6
j= ++i + 5;	i=i+1; j=i+5;	i=4 et j=9
j= i++ + 5;	j=i+5; i=i+1;	j=8 et i=4;

7.5 LES OPERATEURS DE TRAITEMENT DE BITS

Rappel : (opérations logiques)

1 & 1 = 1	**1 | 1 = 1**	**1 ^ 1 = 0**
1 & 0 = 0	**1 | 0 = 1**	**1 ^ 0 = 1**
0 & 1 = 0	**0 | 1 = 1**	**0 ^ 1 = 1**
0 & 0 = 0	**0 | 0 = 0**	**0 ^ 0 = 0**

Les opérateurs de traitement de bits en C s'appliquent uniquement à des entiers ou des caractères. Ils ne s'appliquent, donc, pas à des opérandes réels (de type float ou double).

Opérateur	Traitement
&	et binaire
|	ou binaire
^	ou exclusif
<<	décalage à gauche
>>	décalage à droite
~	complément à 1

Exemples :

Opérateurs de traitement de bits

```
main( )
{
int i=597;          /*i est 0000 0010 0101 0101 en binaire, 255 en hexadécimal*/
int u=12;           /*u est 0000 0000 0000 1100 en binaire*/

i= i & 0x FFFE; /*i reçoit 0000 0010 0101 0100 en binaire,254 en hexadécimal*/
v= u >> 2;          /*v reçoit 0000 0000 0000 0011 en binaire, 3 en décimal
                      (u est décalé de 2 bits à droite)*/
v= ~u;              /*v reçoit 1111 1111 1111 0011 (complément à 1 de u)*/
}
```

8 LES INSTRUCTIONS SELECTIVES

8.1 L'INSTRUCTION SI (IF)

L'instruction if sélectionne le traitement (bloc d'instructions) à faire si une condition est vérifiée.

Syntaxe :

```
if (condition)
        {
            instruction1;
            instruction2;
            …                    Traitement
            instructionn;
        }
```

Exemples :

Instruction if

```
#include <stdio.h>
main( )
{
int a,b;                        /*a et b entiers*/
printf ("Introduire a et b : ");
scanf ("%d%d",&a,&b);
if (a==0)                       /*si a=0 affiche les messages zéro puis FIN*/
    {
        printf("zéro\n");
        printf("FIN\n");
    }
if (a==1)                        /*si a=1 affiche le message un*/
        printf("un\n");
if (a=b)                    /*faire a=b; puis if (a) [si a≠0] affiche le message suivant*/
    {
        printf("a est différent de 0\n");
    }
}
```

Remarque :

- Si le traitement à effectuer est constitué d'**une seule instruction**, il est **possible d'omettre les accolades**.

Lorsque if est utilisée avec **else**, elle indique également le traitement à faire si la condition n'est pas vérifiée.

Syntaxe :

```
if (Condition)
        {
            instruction1;
            instruction2;                    } Traitement1
            …
            instructionn;
        }
else
        {
            instruction1';
            instruction2';                   } Traitement2
            …
            instructionn';
        }
```

Exemples :

Instruction if else

```
#include <stdio.h>
main( )
{
int a,b;                    /*a et b entiers*/
printf ("Introduire a et b : ");
scanf ("%d%d",&a,&b);

if (a==b)                   /*si a=b affiche a égale à b sinon affiche a différent de b */
        printf("a égale à b\n");
else
        printf("a différent de b\n");
}
```

Remarque :

- Dans une **imbrication, else se rapporte** toujours **à if le plus rapproché** (le plus interne).

Exemples :

Instructions if else imbriquées

```
#include <stdio.h>

main( )
{
int a,b,z,n;                /*a, b,z et n entiers*/

printf ("Introduire a, b et n : ");
scanf ("%d%d%d",&a,&b,&n);

if (n>0)
        if (a>b)            /*si n>0 et a>b*/
                z=a;
        else                /*si n>0 et a<=b (else se rapporte à if (a>b))*/
                z=b;

if (n>0)
        {
          if (a>b)          /*si n>0 et a>b*/
                  z=a;
        }
else                        /*si n<=0 (else se rapporte à if (n>0))*/
                z=b;
}
```

8.2 L'INSTRUCTION SWITCH

Elle réalise un **aiguillage vers différentes instructions** en fonction du **contenu d'une variable** de contrôle.
Le **sélecteur** de switch (la variable de contrôle) doit être un **entier** ou un **caractère**.

Syntaxe :

```
switch (Variable de contrôle)
{
        case  Valeur1 :     Traitement1 (bloc d'instructions)
                            break;
        case  Valeur2 :     Traitement2
                            break;
                  …
        case  Valeurn :     Traitementn
                            break;
        default :           Traitementm
}
```

Remarque : Valeuri est une *constante*.

Exemples :

Instruction switch

```
#include <stdio.h>
main( )
{
int a;                  /*a entier*/
char c;                 /*c char*/
printf ("Introduire un nombre et une lettre: ");
scanf ("%d%c",&a,&c);
switch (a)              /*le programme traite tous les cas de a (0,1 ou autres)*/
{
        case 0 :    printf ("Le nombre introduit est zéro\n");
                    break;
        case 1 :    printf ("Le nombre introduit est 1\n");
                    break;
        default :   printf ("Le nombre introduit est différent de 0 et 1\n");
}
switch (c)              /*Ici, on traite uniquement les cas où c égale à x ou y*/
{
        case 'x' :  printf ("La lettre saisie est la lettre x\n");
                    break;
        case 'y' :  printf ("La lettre saisie est la lettre y\n");
                    break;
}
}
```

Remarque :

- L'instruction **break** fait **sortir de switch**.

9 LES INSTRUCTIONS ITERATIVES

9.1 L'INSTRUCTION TANT QUE (WHILE)

L'instruction **while** permet de **répéter un traitement autant de fois qu'une condition est vérifiée**. Les instructions en question sont alors exécutées tant que la condition est vraie.

Syntaxe :

while (*Condition***)**
{
Traitement (bloc d'instructions qui se terminent par **;**)
}

fonctionnement :

Le système teste d'abord si la condition est vraie; si oui, exécute le traitement et remonte automatiquement à la ligne while pour tester de nouveau la condition. Elle s'arrête quant la condition devient fausse.

Exemples:

Instruction while

```
#include <stdio.h>
main( )
{
int n,somme=0;          /*n entier et somme entier initialisé à 0*/

printf("Introduire n : " );
scanf("%d",&n);
while (n>0)             /*tant que n>0, le programme rajoute n à la somme obtenue
                          puis décrémente n*/
        {
           s=s+n;
           n--;
        }
printf ("%d",somme);    /*le programme affiche la somme des nombres compris entre 0
                          et n*/
}
```

Remarque :

- Lors de l'utilisation de l'instruction while, à chaque itération, la **condition est évaluée en premier**, avant l'exécution du traitement.

9.2 L'INSTRUCTION FAIRE TANT QUE (DO WHILE)

L'instruction **do while** permet de **répéter** un traitement jusqu'à ce qu'une condition ne soit plus vérifiée. Elle joue le même rôle que while. Néanmoins, Lors de l'utilisation de l'instruction while, à chaque ***itération*** (fois), le **traitement est exécuté en premier**, avant que la condition ne soit évaluée.

Syntaxe :

do
{
Traitement (bloc d'instructions qui se terminent par ;)
}
while (*Condition***) ;**

fonctionnement :

Le système exécute d'abord le traitement puis teste si la condition est vraie; si oui, il remonte automatiquement à la ligne do pour exécuter de nouveau le traitement. Il s'arrête quant la condition devient fausse.

Exemples:

Instruction do while

```
#include <stdio.h>
main( )
{
int n, somme=0, i=0;          /*n entier et somme et i entiers initialisés à 0*/

printf("Introduire n : " );
scanf("%d",&n);
do                    /*le programme rajoute i à somme puis l'incrémente tant que i ≤ n*/
      {
            somme=somme+i
            i++;
      }
while (i<=n)      ;
printf ("%d",somme);   /*le programme affiche la somme des nombres compris entre 0
                         et n*/
}
```

Remarque :

- Lors de l'utilisation de l'instruction **do while**, le **traitement sera exécuté au moins une fois** quelle que soit la condition.

Exemple:

Instructions while et do while

```
#include <stdio.h>
main( )
{
int i=3;              /*i entier initialisé à 3*/

do
      {
      printf ("%d",i);
      i++;
      }
while (i<3) ;       /*avec do while, i (3) sera affiché même si la condition i<3 est fausse
                      au début*/

i=3;                  /*remet i à 3*/
while (i<3)           /*avec while, rien n'est affiché car la condition i < 3 est fausse*/
      {
      printf ("%d",i);
      i++;
      }
}
```

9.3 L'INSTRUCTION POUR (FOR)

L'instruction **for** permet de **répéter** un traitement donné un **nombre de fois précis**.

Syntaxe :

for (*Initialisations***;***Condition***;***Instructions***)**

{
Traitement (bloc d'instructions qui se terminent par ;)
}

fonctionnement :

for commence au départ, par effectuer les initialisations (en premier argument), exécute le traitement tant que la condition (en deuxième argument) est vérifiée et exécute les instructions (en troisième argument) à chaque fin d'itération.

Exemples:

Instruction for

```
#include <stdio.h>
main( )
{
int i,j,n,somme=0;        /*i, j, n entiers et somme entier initialisée à 0*/

printf("Introduire n : " );
scanf("%d",&n);
for (i=1; i<=n; i++)      /*pour i allant de 1 à n, le programme rajoute i à somme*/
        somme=somme+i ;
printf ("%d",somme);     /*le programme affiche la somme des valeurs comprises entre 0
                           et n*/

for (i=2, j=4; i<5 && j>2; i++, j--)        /*pour i allant de 2 à 4 et j de 4 à 3,
                                              le programme affichera i et j si i<5 et j>2*/
              printf ("i:%d et j:%d\n",i,j);

                        /*Cette boucle affichera donc i:2 et j:4
                                                      i:3 et j:3*/
}
```

9.4 LES INSTRUCTIONS DE SORTIES DE BOUCLES (BREAK ET CONTINUE)

Break permet de **sortir** directement de **la boucle** (for, while ou do while) **la plus interne**.
Continue permet de **passer** directement **à l'itération suivante de la boucle la plus interne**.

Exemple:

Instructions break et continue

```
#include <stdio.h>
main( )
{
int i;                          /*i entier*/
for (i=5; i>0; i--)             /*pour i allant de 5 à 1*/
  {
   if (i==5) continue;          /*arrête l'exécution de l'itération1 et passe à l'itération2*/
   printf ("Ok");
   if (i==4) break;             /*arrête l'exécution à l'itération2 (sort de for)*/
  }                             /*le programme affichera donc une fois OK*/
}
```

9.5 L'INSTRUCTION ALLER A (GOTO)

L'instruction **goto** permet de **brancher** (inconditionnellement) **à une ligne du programme**. Celle-ci doit avoir été **étiquetée** (*précédée* d'une étiquette constituée d'un *identificateur suivi de :*).

Syntaxe :

goto *Etiquette* **;**

fonctionnement :

Le système interrompt l'exécution séquentielle du programme, remonte ou déscend à la ligne appelée étiquette et poursuit l'exécution à partir de celle-ci.

Exemple:

Instruction Goto

```
#include <stdio.h>
main( )
{
int i=0;                                /*i entier initialisé à 0*/

printf("%d",i);                         /*affiche 0*/
goto message;                           /*saute à l'étiquette message*/
i++;                                    /*ne sera alors pas exécuté*/
printf("%d",i);                         /*ne sera alors pas exécuté*/
message : printf("OK\n");               /*affiche OK*/
printf("FIN\n");                        /*affiche FIN*/

}                               /*le programme affichera donc 0, OK et FIN*/
```

Remarque :

- Goto a la réputation de rendre les programmes moins lisibles. Néanmoins, son utilisation est importante dans des cas qui l'impose.

CHAPITRE 4 ETAPES ET DEMARCHES DE RESOLUTION ALGORITHMIQUE

1 CYCLE DE VIE D'UN ALGORITHME

Résoudre un problème par ordinateur consiste à lui fournir un programme (suite d'opérations) qu'il est capable d'exécuter. L'élaboration d'un programme est réalisée suite à une série d'étapes d'**analyse** du problème en question, de **développement** et d'**évaluation** en vue d'aboutir à l'algorithme adéquat.

1.1 DEFINITION DU PROBLEME

A cette étape, il s'agit de répondre aux questions suivantes :

- Quelles sont les données du problème (**entrées**) ?
- Quelles sont les résultats demandés (**sorties**) ?
- Quel(s) est/sont le(s) **traitement**(s) à effectuer ?
- Quels sont les erreurs ou **cas particuliers** susceptibles de se produire et les solutions à proposer ?

1.2 PLANIFICATION

Elle consiste à **décomposer** le problème à traiter **en sous problèmes** plus simples à résoudre. A cette étape, une ***analyse par l'exemple*** aide souvent à dégager les grandes lignes d'un algorithme.

1.3 DEVELOPPEMENT :

Il consiste à effectuer une analyse des sous problèmes obtenus à l'étape 2 en procédant par une démarche descendante ou ascendante. La première démarche consiste à détailler progressivement le problème en opérations plus simples exécutables par la machine. Par ailleurs, la deuxième démarche consiste à traiter le cas simplifié du problème tout en l'enrichissant progressivement jusqu'à résolution finale du problème complexe.

Remarque:

A un problème donné, peuvent correspondre plusieurs découpages en sous-problèmes conduisant à des algorithmes différents.

1.4 VALIDATION

A cette étape, on doit s'assurer que l'algorithme final **répond à toutes les spécifications** du problème définies à l'étape 1.
Pour valider son algorithme, le programmeur peut faire appel dans ***un premier temps*** à la **trace de l'algorithme**. Il s'agit de la **simulation du fonctionnement ou de l'exécution** du programme. Elle permet de *vérifier* si l'algorithme répond à *un cas* donné et aide également à *déduire le type de problème traité* par l'algorithme en question.

Néanmoins, la trace d'un algorithme ne permet pas de confirmer sa validité pour tous les cas possibles. Le programmeur est amené **ensuite à dégager l'ensemble de classes de valeurs** pouvant constituer les **entrées** en jeu pour évaluer et valider son algorithme.

1.5 CODIFICATION

A cette étape, l'algorithme obtenu doit être **codé en un langage de programmation** au choix.

1.6 VERIFICATION ET MISE AU POINT

Il s'agit, à cette étape, de rendre le programme opérationnel : **corriger les erreurs et vérifier les résultats**.
Le **compilateur** du langage utilisé aide à détecter les *erreurs de syntaxe* (mais non pas de conception et de logique). Une fois ces erreurs corrigées, il **traduit le programme source en programme exécutable**. Le programmeur alors pourra faire un jeu d'essais.

2 EXEMPLE DE LA DEMARCHE DESCENDANTE

En procédant par la démarche de résolution descendante, le programmeur part comme on vient de le signaler du général au particulier en détaillant le problème en actions plus simples jusqu'à résolution du problème.

Exemple : Division Euclidienne

Calculer le quotient Q et le reste R de deux nombres entiers positifs A et B.

Définition du problème

Entrées : Les deux nombres entiers positifs A et B.
sorties : Le quotient Q et le reste R.
Traitement : Calculer le quotient Q et le reste R de la division de A par B.
Cas particuliers : B=0.

Version 1 de l'algorithme :

Début
1- Lire A et B
2- Calculer le quotient Q et le reste R de la division de A par B
3- Ecrire (Q, R)
Fin

Planification

Analyse par l'exemple :

Prenons le cas de A=13 et B=4.
Si on suppose que les seules opérations arithmétiques que la machine puisse exécuter sont la soustraction et l'addition de deux nombres, on tâchera alors de résoudre le problème moyennant ces deux opérations. Aussi, nous procédons par des soustractions pour trouver le quotient et le reste.

1- 13-4=9
2- 9-4=5
3- 5-4=1
4- 1-4=-3 (on ne peut pas continuer) ⇒ Q=3 et R=1

Nous remarquons que :

- Q est le nombre de fois que l'on peut effectuer la soustraction.
- R est le résultat de la dernière opération.

Version 2 de l'algorithme :

```
Début
  1- Lire (A,B)
  2- Soustraire B de A autant de fois que c'est possible
  3- Q est égale au nombre de soustractions effectuées
  4- R est le résultat de la dernière soustraction
  5- Ecrire (Q, R)
Fin
```

Développement avec la démarche descendante

Version 3 de l'algorithme :

```
Début
  1- Lire (A,B)
  2- Poser Q=0
  3- Tant que la soustraction A-B est possible faire
       Début
  4-       Soustraire B de A
  5-       Additionner 1 à Q
       Fin
  6- Poser R=A
  7- Ecrire (Q, R)
Fin
```

Version 4 de l'algorithme :

```
Début
  1- Lire (A,B)
  2- Q ← 0
  3- Tant que A≥B faire
       Début
  4-       A ← A-B
  5-       Q ← Q+1
       Fin
  6- R ← A
  7- Ecrire (Q, R)
Fin
```

Validation de l'algorithme

Trace de l'algorithme pour A=11 et B=5 :

Opération	A	B	Q	R
1	11	5	-	-
2	11	5	0	-
3	11	5	0	-
4	6	5	0	-
5	6	5	1	-
3	6	5	1	-
4	1	5	1	-
5	1	5	2	-
3	1	5	2	-
6	1	5	2	1
7	1	5	2	1

On remarque que le résultat final est correct pour le cas A=11 et B=5. Mais, il reste à valider l'ensemble de cas possibles.

Validation par classes de valeurs d'entrées :

Cas où A=0 et B≠0, on obtient R=A et Q=0.
Cas où A≠0 et B=0, le programme **bouclera infiniment**.
Cas où A=0 et B=0, le programme **bouclera infiniment**.
Cas où A≠0 et B≠0, nous avons à valider deux cas :
Cas où A<B, on obtient R=A et Q=0.
Cas où A=B, on obtient R=0 et Q=1.
Cas où A>B, on peut constater qu'au bout d'un nombre fini de fois A devient plus petit que B ; D'où l'arrêt de la boucle.

Nous remarquons que l'algorithme ne marche pas quand **B est égale à 0**. Il s'agit donc d'un **cas particulier dont il faut tenir compte dans l'algorithme.**

Version 5 de l'algorithme :

```
Début
 1- Lire (A,B)
 2- Si B=0 alors
 3-    Ecrire ('Impossible de faire des divisions par 0')
 4- Sinon
       Début
 5-          Q ← 0
 6-          Tant que A≥B faire
                   Début
 7-                      A ← A-B
 8-                      Q ← Q+1
                   Fin
 9-          R ← A
10-          Ecrire (Q, R)
       Fin
Fin
```

Pour les autres cas, nous constatons que l'algorithme s'arrête au bout d'un temps fini. Néanmoins, nous ne pouvons pas **confirmer la pertinence des résultats** obtenus (s'ils sont corrects ou non). Un **jeu d'essai** s'avère alors **nécessaire** mais encore insuffisant pour la validation de l'algorithme.

3 EXEMPLE DE LA DEMARCHE ASCENDANTE

En procédant par la démarche de résolution ascendante, le programmeur part du cas particulier du problème complexe à résoudre et l'enrichit progressivement jusqu'à résolution du problème général.

Exemple : Factures d'électricité

Calculer le montant à payer de plusieurs factures d'électricité sachant que le taux de la TVA est de 20% et le prix unitaire est de 0.6 DH pour la tranche inférieure à 100 KWh et 0.9 DH pour ce qui est au delà .

Définition du problème

Entrées : L'ancien index ai et le nouvel index ni pour chaque facture.
sorties : La consommation, le prix hors taxe PHT et le prix TTC (PTTC) de chaque facture.
Traitement : Calculer la consommation, le prix hors taxe et le prix TTC de chaque facture.
Cas particuliers : ai > ni, ai < 0, ni < 0.

Planification

Traitement : Calculer la consommation, le prix hors taxe et le prix TTC d'une facture.
Généraliser le traitement à plusieurs factures.

Version 1 de l'algorithme :

Début
1- Lire (ai,ni)
2- Calculer la consommation, le prix hors taxe et le prix TTC d'une facture.
3- Ecrire consommation
4- Ecrire PHT
5- Ecrire PTTC
6- Généraliser le traitement à plusieurs factures.
Fin

Développement avec la démarche ascendante

Version 2 de l'algorithme : Traitement d'une facture

Début
1- Lire (ai,ni)
2- Calculer la consommation
3- Calculer le prix hors taxe PHT
4- Calculer le prix TTC PTTC
5- Ecrire (consommation, PHT, PTTC)
Fin

Version 3 de l'algorithme :

```
Début
  1- Lire (ai,ni)
  2- consommation ← ni - ai
  3- Si consommation ≤ 100 alors
  4-     PHT ← consommation * 0.6
  5- Sinon
  6-     PHT ← (100 * 0.6) + (consommation – 100) * 0.9
  7- PTTC ← PHT * (1+0.2)
  8- Ecrire (consommation, PHT, PTTC)
Fin
```

Version 4 de l'algorithme : Introduire les cas particuliers

```
Début
  1- Lire (ai,ni)
  2- Si ai > ni ou ai < 0 ou ni < 0 alors
  3-     Ecrire ('Erreur de saisie')
  4- Sinon
        Début
  5-          consommation ← ni - ai
  6-          Si consommation ≤ 100 alors
  7-                PHT ← consommation * 0.6
  8-          Sinon
  9-                PHT ← (100 * 0.6) + (consommation – 100) * 0.9
 10-          PTTC ← PHT * (1+0.2)
 11-          Ecrire (consommation, PHT, PTTC)
        Fin
Fin
```

Version 5 de l'algorithme : Généraliser à plusieurs factures

```
Début
  1- Lire (ai,ni)
  2- Si ai > ni ou ai < 0 ou ni < 0 alors
  3-     Ecrire ('Erreur de saisie')
  4- Sinon
        Début
  5-          consommation ← ni - ai
  6-          Si consommation ≤ 100 alors
  7-                PHT ← consommation * 0.6
  8-          Sinon
  9-                PHT ← (100 * 0.6) + (consommation – 100) * 0.9
 10-          PTTC ← PHT * (1+0.2)
 11-          Ecrire (consommation, PHT, PTTC)
        Fin
 12- S'il y a une nouvelle facture refaire le même traitement que précédemment
Fin
```

Version 6 de l'algorithme :

```
Début
 1- Faire
        Début
 2-         Lire ai et ni
 3-         Si ai > ni ou ai < 0 ou ni < 0 alors
 4-             Ecrire ('Erreur de saisie')
 5-         Sinon
                Début
 6-                 consommation ← ni - ai
 7-                 Si consommation ≤ 100 alors
 8-                     PHT ← consommation * 0.6
 9-                 Sinon
10-                     PHT ← (100 * 0.6) + (consommation – 100) * 0.9
11-                 PTTC ← PHT * (1+0.2)
12-                 Ecrire (consommation, PHT, PTTC)
                Fin
13-         Lire nouvelle_facture
        Fin
14- Tant que nouvelle_facture = 'oui'
Fin
```

Validation de l'algorithme

Trace de l'algorithme pour ai=100 et ni=170 et ensuite ai=100 et ni=250 :

Opération	Ai	ni	cons	PHT	PTTC	Nouvelle facture
1	-	-	-	-	-	-
2	100	170	-	-	-	-
3	100	170	-	-	-	-
5	100	170	-	-	-	-
6	100	170	70	-	-	-
7	100	170	70	-	-	-
8	100	170	70	42	-	-
11	100	170	70	42	50.4	-
12	100	170	70	42	50.4	-
13	100	170	70	42	50.4	Oui
14	100	170	70	42	50.4	Oui
2	100	250	70	42	50.4	Oui
3	100	250	70	42	50.4	Oui
5	100	250	70	42	50.4	Oui
6	100	250	150	42	50.4	Oui
7	100	250	150	42	50.4	Oui
9	100	250	150	42	50.4	Oui
10	100	250	150	90	50.4	Oui
11	100	250	150	90	108	Oui
12	100	250	150	90	108	Oui
13	100	250	150	90	108	Non
14	100	250	150	90	108	Non

On remarque que le résultat final est correct pour chacun des cas étudiés. Mais, il reste à valider l'ensemble de cas possibles.

Validation par classes de valeurs d'entrées :

Cas où nouvelle facture=non, le programme s'arrête.
Cas où nouvelle facture=oui, le programme commence à calculer la facture concernée.
Cas où ai < 0 ou ni < 0, le programme affiche un message d'erreur et demande s'il y a ou non une nouvelle facture à traiter.
Cas où ai > ni, le programme affiche un message d'erreur et demande s'il y a ou non une nouvelle facture à traiter.
Cas où ai ≤ ni, le programme calcule les résultats demandés, demande s'il y a d'autres factures à traiter et relance le traitement si oui, sinon il s'arrête. Un jeu d'essai aide à confirmer la pertinence des résultats obtenus.

CHAPITRE 5 LES TABLEAUX

Un tableau est une collection **homogène** de données, **ordonnée** et de **taille statique**. Il fut un ensemble d'octets permettant de représenter une liste d'éléments de même type. Chaque élément est repéré par un indice (son rang dans le tableau).

Exemple :

Tableau t	0	1	2	3	4	5	6
	15	20	25	30	100	200	150

15 est l'élément d'indice 0, il est noté en C par **t[0]**.
100 est l'élément d'indice 4, il est noté en C par **t[4]**.

1 TABLEAUX A UNE DIMENSION (VECTEURS)

1.1 DECLARATION

Syntaxe :

Type *Identificateur* **[***Taille constante***] ;**

- La ***Taille*** du tableau est le ***nombre de ses éléments***. Elle ne peut être une variable. ***Elle doit être une constante*** définie avant ou au moment de la déclaration.

exemples:

Déclaration de tableaux

```
#include <stdio.h>
#define  taille1   5      /*taille1: constante de valeur 5*/
#define  taille2   3      /*taille2: constante de valeur 3*/
main( )
{
int a [taille1];          /*a: tableau de 5 entiers*/
a[0]=15;                  /*la première case du tableau a reçoit 15*/
char b [taille2];         /*b: tableau de 3 caractères*/
b[0]='x';                 /*la première case du tableau b reçoit la lettre x*/
b[1]='y';                 /*la deuxième case du tableau b reçoit la lettre y*/
float c [10];             /*c: tableau de 10 réels*/
scanf("%f", &c[0]);       /*lit un réel et l'affecte à la première case de c*/
}
```

Remarque :

- **Les indices** d'un tableau sont des **entiers commençant à 0**.

1.2 REPRESENTATION PHYSIQUE

Lors de la déclaration d'un tableau, une zone mémoire lui sera réservée. Elle sera utilisée pour le stockage de ses données. La ***taille*** de cette zone en octets est la ***multiplication de la taille du tableau par la taille du type de ses éléments*** (un tableau de trois entiers sera représenté par six octets : chaque entier est codé sur deux octets).
Un tableau T correspond à l'adresse mémoire de son premier élément (**T=&T[0]**). Il s'agit de la première cellule de la zone mémoire qui lui est réservé.

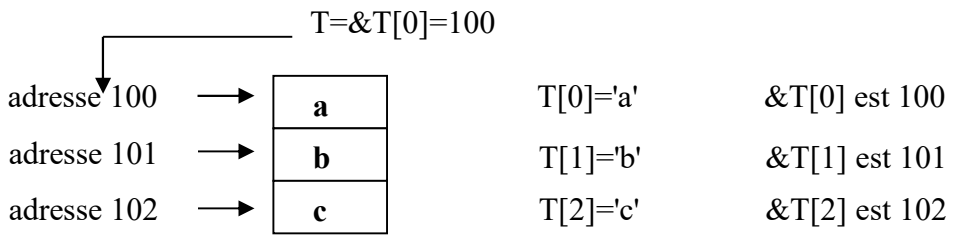
Exemples:

Représentation physique d'un tableau

```
#define taille 3  /*taille: constante de valeur 3*/
main( )
{
     char T [taille];                    /*T : tableau de 3 caractères : a, b et c*/
     T[0]='a';
     T[1]='b';
     T[2]='c';
}
```

Contenu de la mémoire

Supposant que T a été mémorisé à l'adresse mémoire 100, un caractère étant codé sur un octet, le contenu des cellules mémoire sera alors comme suit :



1.3 INITIALISATION

Syntaxe :
Type *Identificateur* **[***Taille constante***] = {***Valeur1***,** *Valeur2***,…,***Valeurn***};**

Exemples:

Initialisation de tableaux

```
#include <stdio.h>
#define taille1 3                 /*taille1: constante de valeur 3*/
main( )
{
float a [taille1]={0.,1.5,3.};    /*a: tableau de 3 réels initialisés à 0 , 1.5 et 3*/
int b [taille1]={1};              /*seul le premier élément du tableau b est initialisé à 1*/
char c [taille1]={'x', ,'y'};     /*seuls le 1er et 3ème éléments de c sont initialisés (à x et y)*/
int d [ ]={4,6,8};                /*d: tableau de 3 entiers initialisés à 4 , 6 et 8*/
}
```

Remarque :

- Un tableau peut être totalement ou **partiellement** initialisé.

1.4 LECTURE ET AFFICHAGE

Les éléments d'un tableau sont à **lire et à afficher élément par élément**.

Exemple:

Lecture et affichage de tableaux

```
#include <stdio.h>
#define taille 20                  /*taille: constante de valeur 20*/
main( )
{
int i, t [taille];                 /*t: tableau de 20 entiers*/
for(i=0;i<taille;i++)              /*lit les 20 entiers élément par élément*/
      scanf ("%d",&t[i]);
for(i=0;i<taille;i++)              /*affiche les 20 entiers élément par élément*/
      printf ("%d\n",t[i]);
}
```

Remarque :

- Lors de la lecture ou de l'affichage d'un tableau, le compilateur C n'empêche pas un dépassement des limites (la taille) du tableau. Une vérification par le programmeur est alors importante.

1.5 AFFECTATION

L'affectation de valeurs aux éléments d'un tableau se fait également individuellement (comme pour la lecture et l'affichage).

Exemple:

Affectation de valeurs à un tableau

```
#include <stdio.h>
#define taille 20                  /*taille: constante de valeur 20*/
main( )
{
int i, t [taille];                 /*t: tableau de 20 entiers*/
for(i=0;i<taille;i++)              /*affecte i à chaque élément d'indice i*/
      t[i]=i;
}
```

Remarques :

- L'affectation d'un tableau B à un autre tableau A se fait élément par élément. Une affectation "brutale" de B à A (**A=B**) **n'est pas possible**.
- L'affectation élément par élément d'un tableau B à un autre tableau A (**A[i]=B[i])** réalise une **copie de B dans A**.

Exemple:

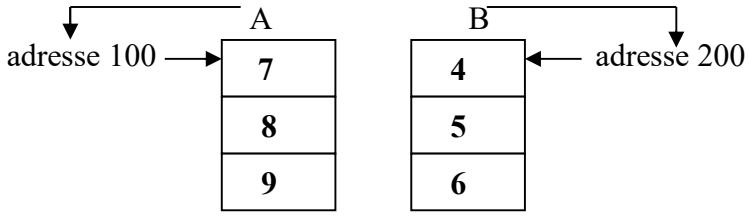
Affectation d'un tableau à un autre élément par élément

```
#define taille 3     /*taille: constante de valeur 3*/
main( )
{
        int i, A [taille]={7,8,9};        /*A : tableau de 3 entiers : 7, 8 et 9*/
        int B [taille]={4,5,6};           /*B : tableau de 3 entiers : 4, 5 et 6*/

        for(i=0;i<taille;i++)             /*copie les éléments de B dans A*/
                A[i]=B[i];

        for(i=0;i<taille;i++)             /*affecte ensuite i à chaque élément d'indice i de A*/
                A[i]=i;
}
```

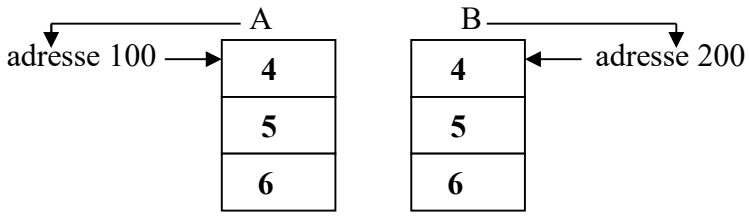
Contenu de la mémoire

Supposant que A a été mémorisé à l'adresse mémoire 100 et B à l'adresse 200, le contenu des cellules mémoire sera comme suit :

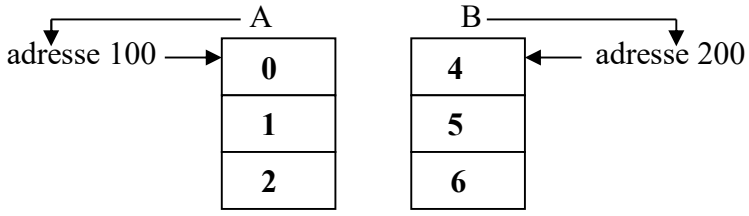
Etat initial :



Après la copie de B dans A :



Après l'affectation des i aux A[i] :



1.6 COMPARAISON DE DEUX TABLEAUX

La **comparaison** des éléments de deux tableaux doit **se faire élément par élément**.

Exemple:

Comparaison de deux tableaux

```
#include <stdio.h>
#define taille 20  /*taille: constante de valeur 20*/
main( )
{
char a [taille], b[taille];        /*a et b: tableaux de 20 caractères*/
int i, egaux;                      /*egaux entier utilisé comme variable logique égale à
                                     vrai (1) ou faux (0)*/
for(i=0;i<taille;i++)              /*lit les éléments de a et b*/
        {
                scanf ("%c",&a[i]);
                scanf ("%c",&b[i]);
        }
egaux=1;                                /*on suppose que a et b sont égaux*/

for(i=0;i<taille;i++)                   /*compare les caractères un par un*/
        if (a[i]!=b[i])                 /*si un élément est différent de son correspondant,*/
                {égaux=0; break;}       /*on arrête la comparaison et on déduit que a et b
                                          ne sont pas égaux*/

if (egaux)                              /*si egaux est vrai (1)*/
        printf ("a et b contiennent le même mot\n");

else                                    /*si egaux est faux (0)*/
        printf ("a et b contiennent des mots différents\n");
}
```

2 CHAINES DE CARACTERES

Une chaîne de caractères est un **tableau de caractères**. Elle représente un cas particulier des tableaux qui **bénéficie de certains traitements particuliers** en plus de ceux réservés aux tableaux en général. Une chaîne de caractères peut également être **déclarée** comme **pointeur sur char** (voir le chapitre pointeurs).

2.1 INITIALISATION

Syntaxe :

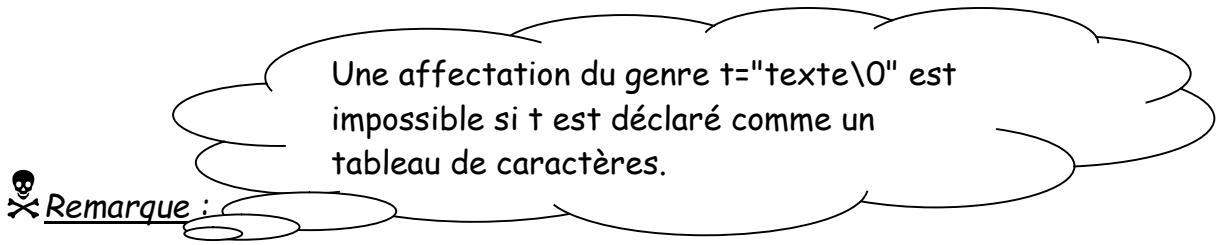
char *Identificateur* **[***Taille constante***]** **=** **"***Texte\0***"** **;**

- Le caractère '**\0**' indique la **fin de la chaîne de caractères**. Il est conseillé de ne pas l'omettre.

Exemples:

Initialisation de chaînes de caractères

```
#define taille1 3              /*taille1: constante de valeur 3*/
#define taille2 4              /*taille2: constante de valeur 4*/

main( )
{
char t [taille1]="ali";        /*t chaîne de caractères initialisée au mot ali*/

char a [taille2]="ali\0";      /*a chaîne de caractères initialisée au mot ali
                                 et terminée par le caractère '\0'*/
}
```



2.2 LECTURE ET AFFICHAGE

Une variable de type chaîne de caractères peut être **lue et affichée caractère par caractère** au moyen de **scanf et printf** utilisant le **format %c**.
Elle peut également être **lue (affichée)** globalement (**d'un seul coup**) au moyen de la fonction **scanf (printf)** utilisant cette fois-ci le **format %s** ou au moyen de la fonction **gets (puts)**.

Syntaxe :

scanf("%s", *Chaîne de caractères***);**
printf("%s", *Chaîne de caractères***);**
gets(*Chaîne de caractères***);**
puts(*Chaîne de caractères***);**

- scanf amène uniquement le texte introduit **avant le premier blanc** (espace) dans la variable à lire.
- gets amène tout le texte introduit **jusqu'au retour chariot** (retour à la ligne) dans la variable à lire.

Exemple:

Lecture et affichage de chaînes de caractères

```
/*Soit Turbo C la chaîne de caractères que l'on introduit lors de l'exécution de l'exemple suivant*/
    #include <stdio.h>
    #define taille 20  /*taille : constante de valeur 20*/
    main( )
    {
    char t [taille];          /*t : chaîne de caractères de taille 20*/
    scanf ("%s",t);           /*lit t (on ne met pas d'adresse &)*/
    printf ("%s",t);          /*affiche Turbo*/
    gets (t);                 /*lit t (on ne met pas d'adresse &)*/
    puts (t);                 /*affiche Turbo C*/
    }
```

2.3 FONCTIONS SUR LES CHAINES DE CARACTERES

Des **fonctions prédéfinies** appliquées aux chaînes de caractères sont définies dans le fichier **"string.h"**. Nous en citons :

fonction	Appliquée à	retourne	rôle
strlen	Une chaîne de caractères	Un entier	Retourne la longueur d'une chaîne de caractères.
strcmp	Deux chaînes de caractères	Un entier	Compare deux chaînes et retourne 0 si elles sont égales, une valeur différente sinon.
strcpy	Deux chaînes de caractères	Rien (void)	Copie une chaîne en deuxième argument dans une autre en premier argument.

Exemples:

Fonctions sur les chaînes de caractères

```
    #include <string.h>
    #define taille      20     /*taille : constante de valeur 20*/
    main( )
    {
    char a [taille], b[taille]="ali\0";     /*a et b chaînes de caractères; b initialisée à "ali\0"*/
    int n;                                  /*n entier*/
    n=strlen(b);                            /*n reçoit 3 (longueur du mot "ali")*/
    strcpy(a,b);                            /*a reçoit la chaîne "ali\0"(copie de b)*/
    n=strcmp(a,b);                          /*n reçoit 0 (résultat de la comparaison de a et b)*/
    }
```

Remarque :

Les **fonctions** du fichier **"string.h"** ne peuvent être appliquées à des caractères de type char. Elles sont **appliquées à** des chaînes de caractères (**tableaux de caractères**) **qui se terminent par '\0'** (On ne peut alors comparer, par exemple, deux caractères a='x' et b='y' au moyen de strcmp).

3 TABLEAUX A PLUSIEURS DIMENSIONS

3.1 DECLARATION

Syntaxe :

Type *Identificateur* **[***Taille1***]** **[***Taille2***]** … **[***Taille n***] ;**

- Taillei est la taille de la dimension i. Elle doit être une constante définie avant ou au moment de la déclaration.
- ***Un élément*** d'un tableau t à n dimensions est repéré par ses indices, sous forme ***t[i1][i2]…[in]***.

Exemples:

Déclaration et lecture d'un tableau à deux dimensions

```
/*Ce programme lit le nombre de buts marqués par chacun des 11 joueurs de 8 équipes*/

    #include <stdio.h>
    #define   taille1   8              /*taille1: constante de valeur 8*/
    #define   taille2   11             /*taille2: constante de valeur 11*/

    main( )

    {
    int t [taille1][taille2];          /*t : matrice de 8 lignes, 11 colonnes*/
    int i, j;

    for(i=0;i<taille1;i++)             /*lit les éléments de t ligne par ligne*/
          for (j=0; j<taille2;j++)
                scanf ("%d",&t[i][j]);
    }
```

Déclaration et lecture d'un tableau à trois dimensions

/*Ce programme lit les notes en Informatique et Mathématiques de 5 étudiants. Chaque étudiant a quatre notes par matière :

Etudiants	Informatique				Mathématiques			
	Note1	Note2	Note3	Note4	Note1	Note2	Note3	Note4
1	19	17	15	18	14	15.5	16	17
2	13	7	9.5	12	13	9	11.5	10
3	14	14.5	16	16.5	16	17	17.5	16.5
4	8.5	10.5	10	12	12	13	13.5	6.5
5	15	19	15	16.5	15	13	12.5	14.5

*/

```
#include <stdio.h>
#define   taille1   5          /*taille1: constante de valeur 5 (5 étudiants)*/
#define   taille2   2          /*taille2: constante de valeur 2 (2 matières)*/
#define   taille3   4          /*taille2: constante de valeur 4 (4 notes)*/
main( )
{
float t [taille1][taille2][taille3];
int i,j,k;
for(i=0;i<taille1;i++)         /*lit les  quatre notes des étudiants étudiant par
                                 étudiant, commençant par les notes d'informatique
                                 puis celles des mathématiques*/
      for (j=0; j<taille2;j++)
            for (k=0; k<taille3;k++)
                  scanf ("%f",&t[i][j][k]);
}
```

3.2 INITIALISATION

Syntaxe :

Type *Identificateur* **[***m***]** … **[***p***]** = { *Liste0,* … , *Listem-1***}** ;

- Listei est l'initialisation de l'élément d'indice i et qui fut un sous tableau de dimension n-1 si n est la dimension du tableau en question.

Exemple:

Le programme ci-dessous initialise la matrice M (3 lignes, 2 colonnes) :

M

0	1
2	3
4	5

Initialisation d'un tableau à plusieurs dimensions

```
#define taille1 3  /*taille1: constante de valeur 3*/
#define taille2 2  /*taille2: constante de valeur 2*/
main( )
{
int M [taille1][taille2]={{0,1},{2,3},{4,5}};          /*initialisation ligne par ligne*/
}                                                  /*M[0]={0,1}, M[1]={2,3} et M[2]={4,5}*/
```

CHAPITRE 6 LES POINTEURS

Un pointeur sur une variable x est une variable qui contient **l'adresse** mémoire de la variable x.

1 DECLARATION DE POINTEURS

Syntaxe :

Type_variable_pointée *******Pointeur*;

Exemples :

```
char *p; /*p peut contenir l'adresse d'une variable de type caractère ou chaîne de caractères*/
int *p;   /*p peut contenir l'adresse d'une variable de type entier*/
```

Remarque :

Lorsqu'un pointeur ne contient aucune adresse valide, il est égal à **NULL** (Pour utiliser cette valeur, il faut inclure le fichier ***stdio.h*** dans le programme).

2 OPERATEURS & ET *

Le langage C met en jeu deux opérateurs utilisés lors de l'usage de pointeurs. Il s'agit des opérateurs **&** et *.

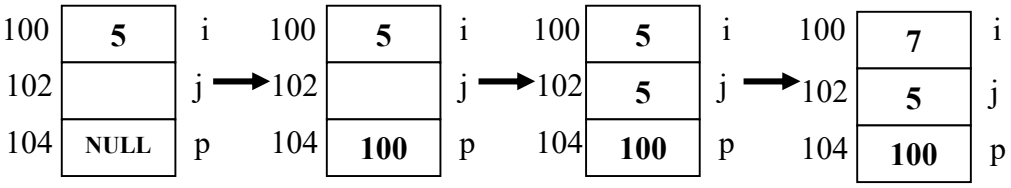
&*variable*	signifie **adresse** de *variable*.
*******pointeur*	signifie **contenu** de l'adresse référencée par *pointeur*.

Exemples:

Opérateurs & et *

```
#include <stdio.h>
main( )
{
 int i,j;                /*i et j des entiers*/
 int *p;                 /*p pointeur sur un entier*/
 i=5;                    /*i reçoit 5*/
 p=&i;                   /*p reçoit l'adresse de i*/
 j=*p;                   /*j reçoit 5 : contenu de l'adresse p*/
 *p=j+2;                 /* le contenu de l'adresse p devient 7 donc i aussi devient 7*/
}
```

/****Représentation mémoire***(supposant que i, j et p se trouvent respectivement aux adresses 100, 102 et 104)*/



3 OPERATEURS ++ ET --

Un pointeur peut être **déplacé** d'une adresse à une autre au moyen des opérateurs ++ et --. **L'unité** d'incrémentation (ou de décrémentation) d'un pointeur est toujours **la taille de la variable pointée**.

Exemple:

incrémentation

```
#include <stdio.h>
main( )
{
 int i;                 /*i entier, supposons qu'il se trouve à l'adresse 100*/
 int *p;                /*p pointeur sur un entier*/
 p=&i;                  /*p reçoit 100 : adresse de i*/
 p++;                   /*p s'incrémente de 2 (devient 102)*/
 char c;                /*c char, supposons qu'il se trouve à l'adresse 200*/
 char *q;               /*q pointeur sur char*/
 q=&c;                  /*q reçoit 200 : adresse de c*/
 q++;                   /*q s'incrémente de 1 (devient 201)*/
}
```

4 ALLOCATION MEMOIRE

Pour ***éviter des erreurs*** fréquemment rencontrées lors de l'utilisation des pointeurs, le programmeur doit, ***immédiatement*** après la déclaration d'un pointeur, l'***initialiser*** à l'adresse d'une variable donnée (par exemple, ***p=&i***) ou lui ***allouer de la mémoire*** et l'initialiser au choix.

Après la déclaration d'un pointeur, la zone mémoire réservée à la variable pointée se trouve dans un espace dynamique (heap). Elle peut être allouée au moyen de la fonction **malloc()** du fichier ***malloc.h***.

Exemple:

Allocation mémoire

```
#include <stdio.h>
#include <malloc.h>
main( )
{

int *q;              /*q pointeur sur des entiers*/
q=(int*) malloc(2*sizeof(int));     /*réservation de 4 octets pour le stockage de deux
                                      entiers pointés par q*/
*q=2;                /*q pointe sur l'entier 2*/
*(q+1)=5;            /*On met 5 à l'adresse suivante, q pointe alors sur les entiers 2 et 5*/

char *p;                    /*p pointeur sur char*/
p=(char*) malloc(5*sizeof(char));       /*réservation de 5 octets pour le stockage de
                                          la chaîne de caractères pointée par p*/
 p="toto\0";                /*toto est la chaîne pointée par p*/
}
```

CHAPITRE 7 LES FONCTIONS

1 INTRODUCTION

Un programme en C peut être **découpé en plusieurs** morceaux (**modules**), chacun exécutant une **tâche précise**. Ces modules sont appelés des **fonctions** dont l'une est principale et dite programme principal (main).
Lors de l'exécution, le programme principal est exécuté en premier. Les autres fonctions sont exécutées lorsqu'elles sont appelées.
La programmation modulaire est justifiée par :

- La faculté de maintenance (détection facile des erreurs…).
- L'absence de répétition de séquences d'instructions (paramétrage de fonctions).
- Le partage et la réutilisation de modules (fonction).

Définition :
Une **fonction** est un **ensemble d'instructions réalisant une tâche précise** dans un programme.

2 STRUCTURE ET PROTOTYPE D'UNE FONCTION

2.1 STRUCTURE D'UNE FONCTION

Exemples:

Utilisation de fonctions

```
/*Ce programme utilise la fonction triple pour calculer le triple d'un entier*/
      #include <stdio.h>
      int triple (int);          /*prototype de la fonction triple qui admet un
                                   paramètre entier et retourne un résultat entier*/
      main( )
      {
      int i,j;                   /*i et j variables entiers locales à main
                                   (appartiennent uniquement à main)*/
      i=2;                       /*i reçoit 2*/
      j=triple(i);               /*appel de la fonction triple : j reçoit 6 triple de 2*/
      printf ("%d",j);           /*affiche 6*/
      j=triple(4);               /* appel de la fonction triple : j reçoit 12 triple de 4*/
      printf ("%d",j);           /*affiche 12*/
      }
      int triple (int n)         /*définition de la fonction triple*/
      {
      int r;                     /*r variable entier locale à la fonction triple*/
      r=n*3;                     /*r reçoit le triple de n*/
      return r;                  /*r est la valeur retournée comme résultat de la fonction*/
      }
```

Une fonction doit être définie sous forme :

```
<Déclaration de la fonction>
{
        <Déclaration des variables locales>
        <Bloc d'instructions>
}
```

La partie **déclaration de la fonction** indique le **type du résultat** retourné par la fonction **ou void** si elle ne retourne rien. Elle indique également ses **paramètres formels et leurs types** si elle en admet, **sinon** indique **void**.

Syntaxe :

Void ou *Type_Résultat Nom_Fonction* **(***Void* ou *Type Param1*,…,*Type Paramn***);**
{
    Corps de la fonction (instructions)
}

Exemple :

Type void

```
#include <stdio.h>
void afficher (void);   /*La fonction afficher n'a ni paramètre ni résultat*/
main( )
{
afficher();
}
void afficher(void)     /*affiche le message bonjour*/
{
      printf("bonjour");
}
```

La déclaration de **variables locales** sert à déclarer les variables utilisées par la fonction et dont la **portée est uniquement cette fonction**.
Le **corps d'une fonction** est constitué de **différentes instructions de C** exécutant la tâche accomplie par la fonction, **en plus** de l'instruction **return** si la fonction retourne un résultat. L'instruction ***Return*** **indique le résultat** retourné par la fonction s'il y en a **et constitue un point de sortie** de la fonction.

2.2 PROTOTYPE D'UNE FONCTION

La fonction **prototype** donne la règle d'usage de la fonction : nombre et type de paramètres ainsi que le type de la valeur retournée.
Il est conseillé de ne pas l'omettre car il **permet au compilateur de détecter des erreurs** lors des appels de la fonction (effectuer des conversions de types…).

Syntaxe :

Void ou *Type_Résultat Nom_Fonction* **(***Void* ou *Type1*,…,*Typen***);**

On peut insérer les noms de paramètres mais ils n'ont aucune signification.

3 APPEL DE FONCTIONS

Syntaxe :

- ***Si la fonction retourne un résultat et admet des paramètres***
  Variable = *Nom_Fonction* **(***Paramètres effectifs***);**
- ***Si la fonction retourne un résultat et n'admet pas de paramètres***
  Variable = *Nom_Fonction* **( );**
- ***Si la fonction ne retourne rien et admet des paramètres***
  Nom_Fonction **(***Paramètres effectifs***);**
- ***Si la fonction ne retourne rien et n'admet pas de paramètres***
  Nom_Fonction **( );**

4 DOMAINES D'EXISTENCE DE VARIABLES

En fonction de leur localisation (en tête du programme ou à l'intérieur d'une fonction) ou d'un qualificatif (static par exemple), une variable présente différentes caractéristiques :

Domaine d'existence de la variable	Signification
locale	Elle n'est référencée que dans la fonction où elle est déclarée.
globale	Placée en dehors des fonctions (généralement au début du programme avant le main), elle peut être accédée par toutes les fonctions.
Static	C'est une variable locale à une fonction mais qui garde sa valeur d'une invocation de la fonction à l'autre.
Externe	Elle est déclarée dans un module. Néanmoins, la mémoire qui lui est réservée lui sera allouée par un autre module compilé séparément.
Register	C'est une variable que le programmeur souhaiterait placer dans l'un des registres de la machine afin d'améliorer les performances.

5 PASSAGE DE PARAMETRES

5.1 PASSAGE PAR VALEUR

Après l'appel de la fonction, les paramètres effectifs qui passent par valeur **gardent leurs anciennes valeurs** d'avant l'appel.

Exemple:

Passage par valeur

```
/*Ce programme utilise la fonction triple pour calculer le triple d'un entier*/
      #include <stdio.h>
      void triple (int, int);   /*prototype de la fonction triple qui admet deux paramètres
                                  entiers i et j (j étant le triple de i). i et j passent par valeur*/
      main( )
      {
      int i,j=0;                /*i et j variables entiers locales à main et j initialisé à 0*/
      i=2;                      /*i reçoit 2*/
      triple(i,j);              /*appel de la fonction qui affiche 6, résultat de triple de 2*/
      printf ("%d",j);          /*affiche 0 (ancienne valeur de j avant l'appel de la fonction)*/
      }
      void triple (int i, int j) /*définition de la fonction triple*/
      {
      j=3*i;                    /*j reçoit 6 le triple de 2*/
      printf ("%d",j);          /*affiche 6*/
      }
```

5.2 PASSAGE PAR ADRESSE (PAR REFERENCE)

Toute manipulation (**changement de valeur**) du paramètre, passant par adresse, à l'intérieur de la fonction aura un impact sur celui-ci après l'appel.

Un **paramètre qui passe par adresse** doit être **déclaré comme pointeur** au moyen de *.

Exemple:

Passage par adresse

```
/*Ce programme utilise la fonction triple pour calculer le triple d'un entier*/
      #include <stdio.h>
      void triple (int , int *); /*prototype de la fonction triple qui admet deux paramètres
                                   entiers i et j (j étant le triple de i). i passe par valeur et j passe
                                   par adresse*/
      main( )
      {
      int i,j=0;                 /*i et j variables entiers locales à main et j initialisé à 0*/
      i=2;                       /*i reçoit 2*/
      triple(i,&j);              /*appel de la fonction qui affiche 6, résultat de triple de 2*/
      printf ("%d",j);           /*affiche 6 (nouvelle valeur de j après l'appel de la fonction)*/
      }
      void triple (int i, int *j)      /*définition de la fonction triple*/
      {
      *j=3*i;                          /*j reçoit 6 le triple de 2*/
      printf ("%d",*j);                /*affiche 6*/
      }
```

Remarques :

- Les **tableaux passent** par défaut (et toujours) **par adresse**.

- Un **tableau à une dimension** qui passe comme **paramètre** d'une fonction **peut être déclaré comme** un tableau **dynamique** (sans préciser sa taille).

Exemple:

Fonction avec des tableaux comme paramètres

```
/*Ce programme calcule la somme de deux vecteurs (tableaux d'entiers)*/

    #include <stdio.h>
    #define taille 10
    void somme_vecteurs (int [ ], int [ ], int [ ]);  /*prototype de la fonction*/

    main( )
    {

    int a[taille], b[taille], c[taille],i;          /*a, b et c vecteurs d'entiers et i entier*/
    for (i=0; i<taille; i++)
          scanf ("%d%d",&a[i],&b[i]);               /*lit les éléments de a et b*/
    somme_vecteurs(a,b,c);                          /*c reçoit la somme des vecteurs a et b*/
    for (i=0; i<taille; i++)
          printf ("%d  ",c[i]);                     /*affiche les éléments de c*/

    }

    void somme_vecteurs (int a[ ], int b[ ], int c[ ])      /*définition de la fonction*/
    {
    int i ;
    for (i=0; i<taille; i++)                        /*c reçoit la somme des vecteurs a et b*/
          c[i]=a[i]+b[i];
    }
```

6 FONCTION RECURSIVE

Une fonction récursive est une fonction calculable en un temps fini, qui **dans sa définition fait appel à elle-même**. Cette fonction doit être sujet à une **condition d'arrêt** qui devient fausse au bout d'un temps fini et assure ainsi l'arrêt de l'exécution.

Exemple:

Fonction récursive

```
/*Ce programme appelle la fonction récursive puissance pour calculer x^y (x entier et y entier positif*/

    #include <stdio.h>
    int puissance (int, unsigned);      /*prototype de la fonction puissance*/

    main( )
    {
    int x;                              /*x entier*/
    unsigned y;                         /*y entier positif*/
    scanf ("%d%u",&x,&y);               /*lit les valeurs de x et y*/
    printf ("%d",puissance (x,y));      /*affiche le résultat de x^y*/
    }

    int puissance (int x, unsigned y)   /*définition de la fonction puissance*/
    {
    if (y==0)                           /*retourne 1 si y=0*/
        return (1);
    else                                /*retourne x*x^(y-1) (appelle de nouveau puissance
                                          pour x et y-1)*/
        return(x*puissance (x,y-1));
    }
```

CHAPITRE 8 LES STRUCTURES

1 DEFINITION

Une structure en C est une **collection de données de types différents** regroupées sous une entité logique dite structure.

Exemples:

- La structure **adresse** est composée des champs numéro (**entier**),rue et ville (**chaînes de caractères**).
- La structure **date** est composée des champs jour, mois et année.

2 DEFINITION DES TYPES STRUCTURES

La définition d'un type structure peut être effectuée au moyen des mots réservés **struct** ou **typedef**.

a) Définition par struct

Syntaxe :

```
struct   Nom_Structure
    {
        Type1   Champ1;
        …
        Typen   Champn;
    } ;
```

Exemple:

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

b) Définition par typedef

Syntaxe :

```
typedef struct
    {
        Type1   Champ1;
        …
        Typen   Champn;
    } Nom_Type_Structure ;
```

Exemple:

```
typedef struct
    {
        int jour;
        int mois;
        int annee;
    }date;
```

3 DECLARATION DES VARIABLES STRUCTURES

La déclaration d'une variable de type structure peut introduire le mot struct ou non selon que son type a été défini moyennant struct ou typedef.

Syntaxe :

1- Au fur et à mesure de la définition du type structure moyennant struct

struct *Nom_Structure*
{
Type1 Champ1;
...
Typen Champn;
} *Nom_Variable* ;

2- Après la définition du type structure moyennant struct

struct *Nom_Structure Nom_Variable*;

3- Après la définition du type structure moyennant typedef

Nom_du_Type_Structure Nom_Variable;

Exemples :

Déclaration de variables structures

```
#include <stdio.h>
typedef  struct          /*définit le type structure date*/
        {int jour;
         int mois;
         int annee;
        }date;
main( )
{
 struct                  /*déclare une variable structure d composée de jour, mois et annee*/
        {int jour;
         int mois;
         int annee;
        }d;
 struct  date1           /*déclare une variable d1 de type structure date1*/
        {int jour;
         int mois;
         int annee;
        }d1;
 struct date1 d2;        /*déclare une variable d2 de type structure date1*/
 date d3;                /*déclare une variable d3 du type défini date*/
}
```

4 ACCES AUX CHAMPS D'UNE STRUCTURE

Un champ d'une structure est référencé par son nom précédé du nom de la structure et un point.

Syntaxe :

Nom_Variable_Structure.Nom_Champ

Remarque :

Lorsqu'on utilise un **pointeur sur une structure**. Ses champs seront référencés par le nom du pointeur suivi d'une flèche puis le nom du champ en question.

Syntaxe :

Pointeur_de_Type_Structure->nom_Champ

Exemples :

Accès aux données d'une structure

```
#include <stdio.h>
typedef  struct            /*définit le type structure date*/
        {int jour;
         int mois;
         int annee;
        }date;
main( )
{
 date d1;            /*déclare une variable d1 du type défini date*/
 date *d2;           /*déclare une variable d2 pointeur sur le type défini date*/
                     /*affecte à d1 la date 01/03/2000*/
 d1.jour=1;
 d1.mois=3;
 d1.annee=2000;
                     /*affecte à d2 la date 01/03/2000*/
 d2->jour=1;
 d2->mois=3;
 d2->annee=2000;
}
```

CHAPITRE 9 LES FICHIERS

1 INTRODUCTION

Un fichier est une ***collection homogène*** de données, ***ordonnée*** et de **taille dynamique**. Il fut un document que l'on pourrait garder en mémoire secondaire (disque dur…).
Il existe, en général, deux types de fichiers :

- Les **fichiers textes** : sont considérés comme une collection de ***lignes de texte***. Lorsqu'il s'agit de ce type de fichiers, le système réagit à certains caractères comme le caractère de fin de ligne ou de fin de texte.
- Les **fichiers binaires** : sont des suites d'octets qui peuvent faire l'objet, par exemple, de collections d'entités constituées d'un nombre précis d'octets (***enregistrements ou structures***). Dans ce cas, le système n'attribue aucune signification aux caractères transférés depuis ou vers le périphérique.

Un **fichier en C** est en général considéré comme une **suite d'octets** qu'on peut gérer par un ensemble de fonctions prédéfinies. Ces fonctions sont réparties en deux groupes :

- Les **fonctions de niveau 1** : les données manipulées par ces fonctions sont mémorisées et transmises par le biais ***d'appels directs au système*** d'exploitation.
- Les **fonctions de niveau 2** : elles sont caractérisées par l'usage d'un ***tampon mémoire propre*** au programme (à l'intérieur de son segment de données) pour le stockage temporaire de données. Elles ne font appel au système que pour transmettre les données en mémoire secondaire lorsque le tampon est totalement rempli. Ces fonctions sont prédéfinies dans la bibliothèque ***stdio.h***

Nous nous intéressons dans ce chapitre aux fonctions principales du niveau 2.

2 DECLARATION DE FICHIERS

Un fichier possède un **nom logique** (***nom externe*** ou nom Dos de type chaîne de caractères) et un **nom physique** (***nom interne***) de type ***pointeur sur FILE***. Un fichier est déclaré par son nom physique.

Syntaxe :

FILE ***nom_interne** **;**

Exemple :

```
FILE *fiche;  /*fiche est déclaré comme pointeur sur FILE. Il pourrait alors faire l'objet de
                nom physique d'un document fichier.*/
```

3 FONCTIONS DE NIVEAU 2

Pour manipuler les données d'un fichier (en lecture ou en écriture), il convient toujours de **l'ouvrir au début** de l'intervention et le **fermer à sa fin**.

3.1 OUVERTURE DE FICHIERS

Syntaxe :
fopen(*nom_fichier_interne* **,** *modes d'ouverture et de fichier* **);**

La fonction fopen ***ouvre*** le fichier physique (fichier Dos) et retourne un pointeur sur FILE ou ***Null*** si erreur.

Le paramètre ***modes*** permet d'indiquer le mode de fichier (binaire ou texte) et le mode d'ouverture (création, lecture, écriture ou mise à jour). Il s'agit des paramètres :

b indique le mode ***binaire***.
t indique le mode ***texte***.
w ***ouvre*** un fichier en ***écriture***. Il le crée s'il n'existe pas et l'***écrase s'il existe déjà.***
r ***ouvre*** un fichier en ***lecture*** seule.
a permet d'***ajouter des éléments à la fin*** d'un fichier existant.
r+ permet de ***mettre à jour*** un fichier (modifier des éléments).
w+ ***crée*** un fichier avec accès en ***lecture et écriture***.
a+ permet de faire des ***ajouts*** en fin de fichier et ***des mises à jour.***

Le mode peut alors être de la forme ***wb*** , wt, ***r+b***, ab, etc.

Exemple:

Ouverture de fichiers

```
#include <stdio.h>
main( )
{
 File *f;                              /*f pointeur sur FILE */
 f=fopen("lettre.txt", "rt");          /*ouvre le fichier texte lettre.txt en lecture*/
 if f==Null                            /*affiche un message s'il y a erreur*/
       printf("Erreur");
 else
 {
 …                                     /*instructions de lecture*/
 fclose (f);                           /*ferme le fichier*/
 }
 f=fopen("document.bat", "wb");        /*crée le fichier binaire document.bat*/
 if f==Null                            /*affiche un message s'il y a erreur*/
       printf("Erreur");
 else
       fclose (f);                     /*ferme le fichier*/
}
```

Remarque :
Lorsqu'on ouvre un fichier, son pointeur pointe sur son premier élément.

3.2 FERMETURE DE FICHIERS

Syntaxe :

fclose(*nom_fichier_interne***);**

La fonction fclose ***ferme*** le fichier dont le nom interne est indiqué en paramètre. Elle retourne 0 si l'opération s'est bien déroulée, -1 en cas d'erreurs.

3.3 LECTURE DE FICHIERS

Syntaxe :

fread (*adresse***,** *taille_d'un_bloc_en_octets***,** *nombre_de blocs***,** *nom_fichier_interne***);**

La fonction fread lit un ou plusieurs blocs (structures par exemple) à partir d'un fichier et les copie à l'adresse indiquée (adresse d'une structure par exemple).
Elle retourne le nombre de blocs effectivement lus.

Exemple :

Lecture de fichiers

```
#include <stdio.h>
typedef  struct                          /*définit le type personne*/
        {char Nom [6];
         char Prenom [6];
        }personne;
main( )
{
 File *f;                                /*f pointeur sur FILE */
 personne p;                             /*p variable de type personne*/
 int n;
 f=fopen("personnes.bat", "rb");         /*ouvre le fichier personnes.bat en lecture*/
 if f==Null                              /*affiche un message s'il y a erreur*/
        printf("Erreur d'ouverture");
 else
 {
  while (n=fread(&p,12,1,f))             /* lit un bloc de 12 octets à partir du fichier f et
                                             le copie dans p. Tant qu'il y a de blocs lus
                                             (n≠0), elle les affiche.
                                            12 peut être remplacé par sizeof (personne)*/
        printf("%s  %s",p.nom,p.prenom);

  fclose (f);                            /*ferme le fichier*/
 }
}
```

Remarque :

Quand le résultat retourné par fread est inférieur au nombre de blocs à lire, cela signifie que :

- soit la fin du fichier a été atteinte.
- soit une erreur s'est produite.

Pour avoir plus de précision, le programmeur peut insérer des contrôles au moyen des fonctions ferror ou feof (voir plus loin).

3.4 ECRITURE DANS UN FICHIER

Syntaxe :

fwrite (*adresse***,** *taille_d'un_bloc_en_octets***,** *nombre_de blocs***,** *nom_fichier_interne***);**

La fonction fwrite ***copie*** un ou plusieurs blocs (structures par exemple) dans un fichier à partir de l'adresse indiquée (adresse d'une structure par exemple).
Elle retourne le nombre de blocs effectivement copiés (écrits).

Exemple :

Ecriture dans un fichier

```
#include <stdio.h>
typedef  struct                          /*définit le type personne*/
        {char Nom [6];
         char Prenom [6];
        }personne;
main( )
{
 File *f;                                /*f pointeur sur FILE */
 personne p;                             /*p variable de type personne*/
 int n;
 f=fopen("personnes.bat", "wb");         /*ouvre le fichier personnes.bat en écriture*/
 if f==Null                              /*affiche un message s'il y a erreur*/
       printf("Erreur d'ouverture");
 else
 {
  scanf("%s%s",p.nom,p.prenom);
  n=fwrite(&p,12,1,f)                    /* copie un bloc de 12 octets à partir de p dans
                                            le fichier f */
  if (n != 1)
       printf(Erreur d'écriture");
  fclose (f);                            /*ferme le fichier*/
 }
}
```

Remarque :

Après une **exécution** des fonctions **fread** ou **fwrite**, le **pointeur du fichier se déplace à l'élément suivant**.

3.5 DETECTION DE FIN DE FICHIER

Syntaxe:
feof (*nom_fichier_interne***)**

La fonction feof retourne 0 si la fin de fichier n'a pas été détectée, une valeur différente de 0 sinon. En fait, elle ***n'indique la fin*** de fichier ***qu'après une lecture*** effective qui n'aboutit pas.

3.6 ACCES AUX ELEMENTS D'UN FICHIER

Syntaxe :
fseek (*nom_fichier_interne*, *position*, *mode* **);**

La fonction fseek permet un **accès direct** à une position dans un fichier selon le mode indiqué. Elle retourne 0 si l'opération s'est déroulée normalement, -1 sinon.
Le mode peut être égal à :

0 Position par rapport au ***début*** du fichier.
1 Position par rapport à la ***position courante***.
2 Position par rapport à la ***fin*** de fichier.

Exemple :

Accès aux données dans un fichier

```
#include <stdio.h>
typedef  struct                          /*définit le type personne*/
        {char Nom [20];
         char Prenom [20];
        }personne;
main( )
{
 File *f;                                /*f pointeur sur FILE */
 personne p;                             /*p variable de type personne*/
 int n;
 f=fopen("personnes.bat", "rb");         /*ouvre le fichier personnes.bat en lécture*/
 if f==Null                              /*affiche un message s'il y a erreur*/
       printf("Erreur d'ouverture");
 else
 {
  n=fseek (f,2*sizeof(personne),0)       /*pointe sur le troisième élément du fichier f */
  if (n==0)                              /*s'il n' y a pas d'erreur, il affichera l'élément*/
  {
          fread(&p, sizeof(personne),1,f);
         printf("%s  %s",p.nom,p.prenom);
  }
  else    printf(Erreur d'accès");
  fclose (f);                            /*ferme le fichier*/
 }
}
```

Remarques :

- La fonction rewind remet le pointeur au ***début*** de fichier :
  rewind (*nom_fichier_interne***)**

- La fonction ftell retourne la **position actuelle** de type ***long*** (***exprimée en octets***) par rapport au début du fichier.
  ftell (*nom_fichier_interne***)**

3.7 DETECTION D'ERREUR

Syntaxe :
ferror (*nom_fichier_interne***)**

La fonction ferror retourne 0 s'il n'y a aucune erreur, une valeur différente de 0 en cas d'erreurs.