



- Module 1 -

Algorithmique et Programmation

- Cours -

Pr. Kaoutar BAIBAI

Chapitre 1 :

Généralités sur l'Algorithmique

Introduction

Notion précise **d'algorithme** a été découverte en 825 par le fameux Abou Jaafar Mohammed Ibn Moussa **Al Khawarizmi** auteur de livre «Al Jabr oua El Mokabala» et membre d'un académie des sciences à Bagdad.

Les ordinateurs sont utilisés pour :

- ☐ Traitement d'informations ;
- ☐ Le stockage d'informations.

Généralités & Définitions

1.1 Définitions

Algorithmique :

- ❑ Un algorithme est une **séquence** d'étapes de calcul qui **transforment** l'entrée en sortie.
- ❑ Un algorithme est aussi considéré comme un outil permettant de résoudre un problème de calcul bien spécifié.

Un programme:

- ❑ Un programme est une **suite d'instructions** permettant à un système informatique **d'exécuter** une tâche donnée.
- ❑ Suite d'instructions

Généralités & Définitions

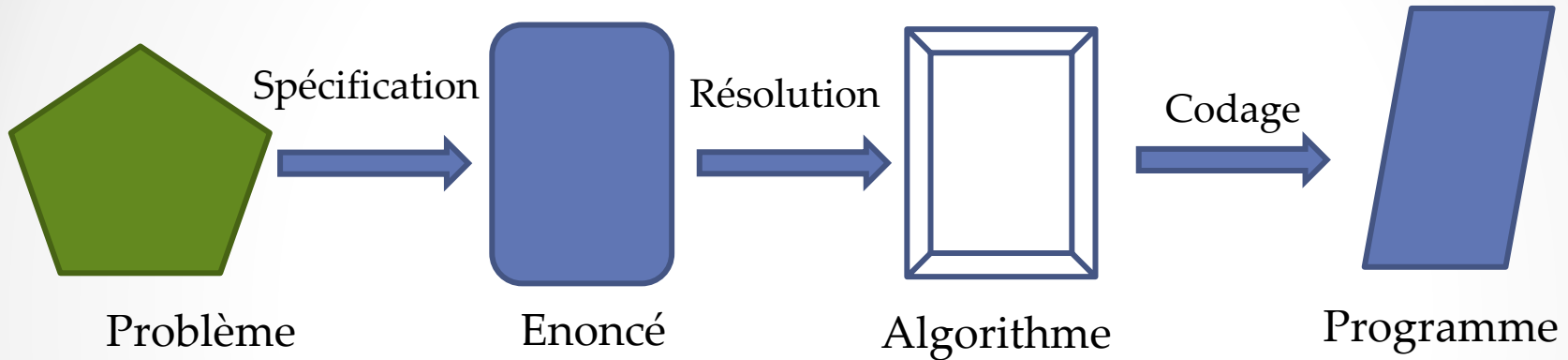
Les étapes de Résolution d'un problème par ordinateur :

Pour résoudre un problème à l'aide d'un ordinateur, il faut :

1. **Analyser ce problème** : Définir les entrées et les sorties.
2. **Déterminer la méthode de résolution** : Déterminer la suite des opérations à effectuer pour résoudre le problème. Plusieurs méthodes peuvent être trouvées ; il faut choisir la plus efficace.
3. **Formuler l'algorithme définitif** : Représenter la méthode de résolution par un algorithme écrit en un langage algorithmique, appelé aussi langage de description d'algorithme (LDA), ou encore pseudo-code.
4. **Traduire l'algorithme** en un langage de **programmation** adapté.

Généralités & Définitions

➤ Concevoir un algorithme



➤ Les étapes de Résolution d'un problème par ordinateur :



Généralités & Définitions

Exemple 1 : Somme de deux Variables

Soit le problème de calcul de la somme de deux nombres. Ce problème peut être résolu de la manière suivante :

A. Analyse

- ☐ Entrées : valeur1 et valeur2.
- ☐ Sortie : la somme des deux valeurs.

B. Solution

Le calcul de la somme consiste à:

1. Avoir les deux valeurs (lire valeur1 et valeur2).
2. Additionner les deux valeurs.
3. Afficher le résultat (Ecrire la somme).

Généralités & Définitions

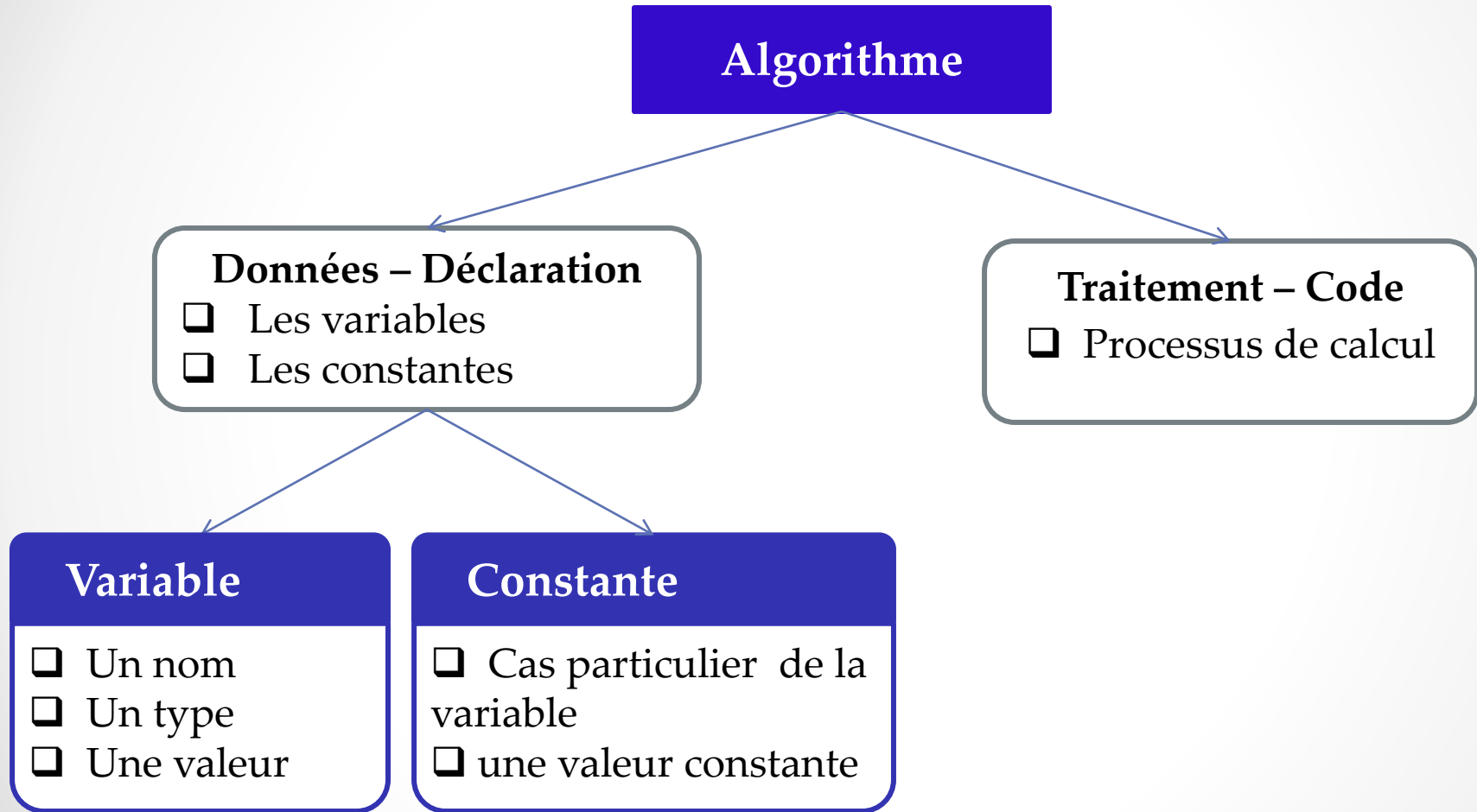
□ Complexité Algorithmique :

- La complexité d'un algorithme est le nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu.
- L'efficacité d'un algorithme est son aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute.

Exemple de complexité

- Complexité en $n \sum_{i=1}^n i$
- Complexité en $2n^2 \sum_{i=1}^n \sum_{j=1}^n (i + j)$

Algorithme



Ecrire un algorithme

Algo nomAlgorithme

Variables

....

Début

Les instructions

Fin

Ecrire un algorithme

Les variables :

Une variable sert à stocker la valeur d'une donnée dans un langage de programmation, elle désigne un emplacement mémoire dont le contenu peut changer au cours d'un programme.

- La variable doit être déclarée avant d'être utilisée,
- Elle doit être caractérisée par un nom (Identificateur).

Début:

Mot réservé indiquant que les lignes qui suivent sont les instructions de l'algorithme.

Fin:

Ce mot réservé indique que l'algorithme est terminé.

Ecrire un algorithme

Les instructions entrée/sortie

Saisir : Instruction permettant de **récupérer** une valeur saisie au clavier (entrée standard) par l'utilisateur et de **ranger** cette valeur dans une variable déclarée au préalable.

- **Une variable** : réceptacle de valeurs (données)
- Cette variable est spécifiée entre ().

On dit : passée en paramètre.

Attention: Prendre en considération le type associé à la variable

Ce genre d'instructions est appelé commande d'entrée.

Ecrire un algorithme

Les instructions entrée/sortie

Afficher : Instruction permettant d'afficher les données passées en paramètres sur la sortie standard (écran).

- **On dit que c'est une commande de sortie.**

Exemple Algorithme 1

Rectangle

Algo rectangle

Début

Variables largeur, longueur, S, P : réel

Ecrire (entrer la largeur)

Lire (largeur)

Ecrire (entrer la longueur)

Lire (longueur)

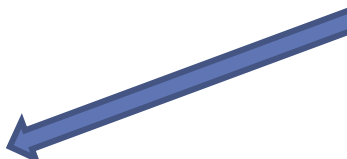
$S = \text{largeur} * \text{longueur}$

$P = 2 * (\text{largeur} + \text{longueur})$

Ecrire (la surface d'un rectangle est :), S

Ecrire (le périmètre d'un rectangle est :), P

Fin



Instruction permettant de réserver de l'espace mémoire pour stocker des données

Exemple Algorithme 2

Saluer une personne

Algorithme saluer

Var

chaîne nom

entier age

Début

Afficher(" Saisir votre nom")

Saisir(nom)

Afficher(" Saisir votre age")

Saisir(age)

Afficher("Salut "+ nom+ " votre âge est "+ age+ "ans")

Fin

Les Types

Variable <nom de donnée>: **type**

- Instruction permettant de réserver de l'espace mémoire pour stocker des données
- Dépendant du type des données ;

Une variable peut se voir attribuer trois nature de valeurs :

1. Valeur numérique : 1, 30, 3.5, ...

2. Valeur en chaîne de caractère : " A" ,"Salut ",

...

3. Valeur Booléenne : vrai ou faux

Les Types

Les types numériques sont :

1. **Entier** : nombres entiers
2. **Réel** : nombres à virgule (partie entière +partie décimale)

- ❑ Selon le langage de programmation utilisé, on peut trouver des variantes des ces types
Exemple en C : short, int, long, float, double, ...
- ❑ Ces variantes concernent souvent la taille du nombre et le fait qu'il possède un signe ou non.

Les Types

Les opérations applicables aux types numérique:

Opérations	Symboles
Soustraction	-
Addition	+
Multiplication	*
Division	/
Puissance	^
Partie entier d'un nombre	ent
Partie décimale d'un nombre	dec
Comparaison	<, <=, >, >=, =, <>

Le reste de la division entière mod (modulo).

Les Types

Les opérations applicables aux types symboliques :

❑ **Booléenne :**

NON ET OU

❑ **Caractère :**

Opérateurs de comparaison

<, <=, >, >=, =, <>

Les opérations de base

1. Affectation :

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

Variable \leftarrow *expression*

Exemples d'affectation

$a \leftarrow 10$

a reçoit la constante **10**

$a \leftarrow (a*b)+c$

a reçoit le résultat de **(a*b)+c**

$d \leftarrow 'm'$

d reçoit la lettre **m**

Les opérations de base

2. La lecture :

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

Lire *variable*

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

Les opérations de base

3. Ecriture :

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

Ecrire *expression*

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

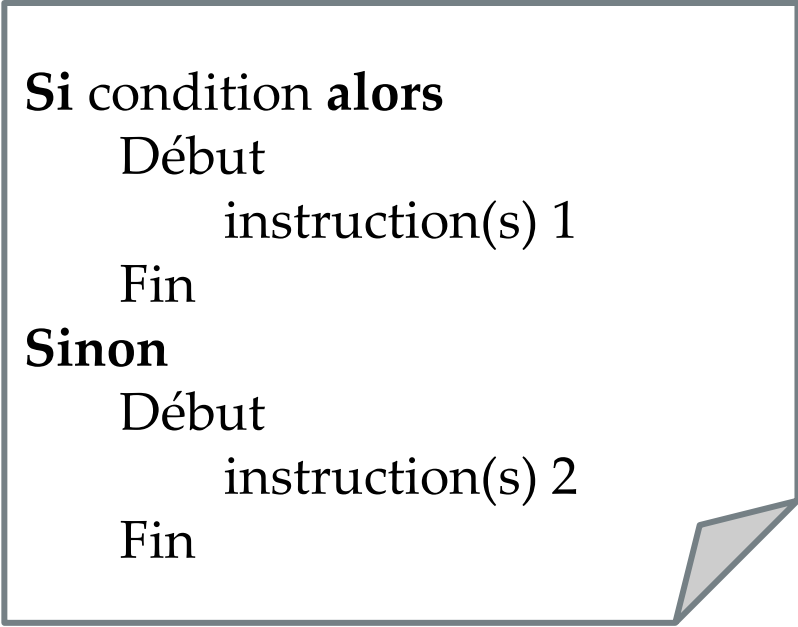
Ecrire $a+b$

Affiche la valeur de $a+b$

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Si

A diagram showing the structure of a conditional instruction block. It is enclosed in a rectangular box with a folded bottom-right corner. The text inside is as follows:

```
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.

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Si

Exemple :

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

- 10% si le $CA < 5000DH$
- 20% si le $CA \geq 5000DH$

Algorithme

lire CA

Si $CA < 5000$ **alors**

Taxe $\leftarrow CA * 10\%$

Sinon

Taxe $\leftarrow CA * 20\%$ Ecrire Taxe

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Si imbriqués

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

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Si imbriqués

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.

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Si imbriqués

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')

Les instructions de contrôle

1. Instructions sélectives

□ Instruction Selon

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

Selon *variable*

valeur1 : instruction(s) 1

valeur2 : instruction(s) 2

...

valeurn : instruction(s) n

...

Sinon instruction(s) m

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Pour

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).

Pour *compteur* \leftarrow *valeur1* à *valeur n* **faire**
 Début
 Instruction(s)
 Fin

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Pour

Exemple :

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

Exemple d'algorithme utilisant l'instruction **Pour**

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

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Tant que

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

Tant que *condition* **faire**

Début

instruction(s)

Fin

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Tant que

Exemple :

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

Exemple d'algorithme utilisant l'instruction **tant que**

lire n $s \leftarrow 0$

Tant que $n > 0$

faire Début

$s \leftarrow s + n$ $n \leftarrow n - 1$

Fin

Ecrire s

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Faire Tant que

Exemple :

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

Faire

Début

instruction(s)

Fin

Tant que *condition*

Les instructions de contrôle

1. Instructions Itératives

□ Instruction Faire Tant que

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

Exemple d'algorithme utilisant l'instruction **Faire Tant que**

lire n $s \leftarrow 0$ $i \leftarrow 0$

Faire

Début

$s \leftarrow s + i$

$i \leftarrow i + 1$

Fin Tant que $i \leq n$

Ecrire s

Exercices

Exercice 1 : Ecrire un algorithme permettant de lire trois nombres, de calculer et d'afficher leur somme, leur produit et leur moyenne.

Exercice 2 : Ecrire un algorithme qui demande un nombre à l'utilisateur, puis calcule et affiche le carré, le double et le triple de ce nombre.

Exercice 3 : Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 4 : Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercices

Exercice 1 :.

```
Algorithme calculs
Variables
    Réel    somme, produit, moyenne, nb1, nb2, nb3 ;
Début
    Ecrire("Donnez trois nombres :") ;
    Lire(nb1, nb2, nb3) ;
    somme ← nb1 + nb2 + nb3 ;
    produit ← nb1 * nb2 * nb3 ;
    moyenne ← somme / 3 ;
    Ecrire("La somme de ces trois nombres est : ", somme) ;
    Ecrire("Le produit de ces trois nombres est : ", produit) ;
    Ecrire("La moyenne de ces trois nombres est : ", moyenne) ;
Fin
```

Exercices

Exercice 2 :

```
Algorithme carre_double_triple
Variables
    Entier    nb, c, d, t ;
Début
    Ecrire("Entrez un nombre :") ;
    Lire(nb) ;
    c ← nb * nb ;
    d ← 2 * nb ;
    t ← 3 * nb ;
    Ecrire("Le carré = ", c) ;
    Ecrire("Le double = ", d) ;
    Ecrire("Le triple = ", t) ;
Fin
```

Exercices

Exercice 3 :

```
Variable N en Entier
Debut
N ← 0
Ecrire "Entrez un nombre entre 1 et 3"
TantQue N < 1 ou N > 3
    Lire N
    Si N < 1 ou N > 3 Alors
        Ecrire "Saisie erronée. Recommencez"
    FinSi
FinTantQue
Fin
```

Exercices

Exercice 4 :

```
Variable N en Entier
Debut
N ← 0
Ecrire "Entrez un nombre entre 10 et 20"
TantQue N < 10 ou N > 20
  Lire N
  Si N < 10 Alors
    Ecrire "Plus grand !"
  SinonSi N > 20 Alors
    Ecrire "Plus petit !"
  FinSi
FinTantQue
Fin
```

Chapitre 2

Concepts de base du langage C

Langage C

C'est 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

Langage C

La compilation

Le C est un langage *compilé* (par opposition aux langages interprétés). Cela signifie qu'un programme C est d'écrit par un fichier texte, appelé *fichier source*. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé *compilateur*. La compilation se décompose en fait en 4 phases successives.

Phases de compilation

1. **Le traitement par le préprocesseur** : Le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles.
2. **La compilation** : La compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est à dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible
3. **Assemblage** : Cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Le fichier produit par l'assemblage est appelé fichier *objet*.

Phases de compilation

4. **L'édition de liens** : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des bibliothèques de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'Édition de liens produit alors un fichier dit *exécutable*.

Structure d'un programme en c

1. Un premier programme en c

Ce programme affiche le message bonjour

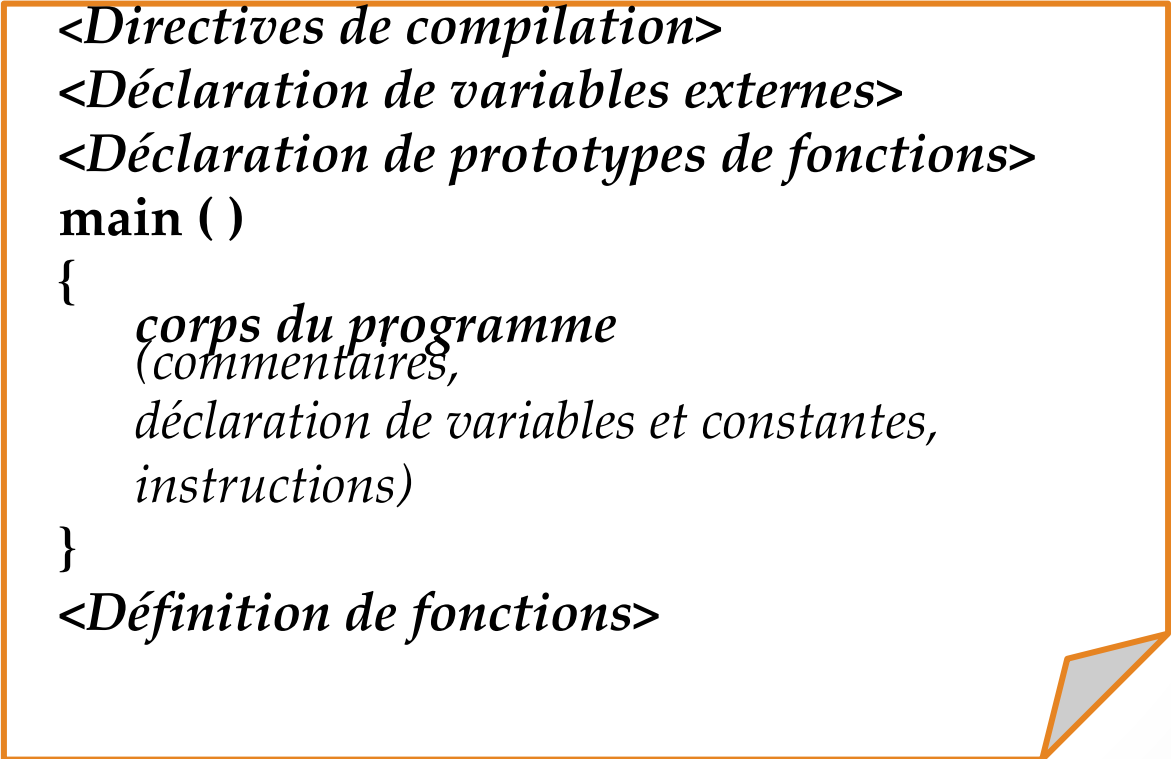
```
main()  
{  
    printf("bonjour la première année");  
}
```

- **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**.

Structure d'un programme en c

2. Un Structure générale d'un programme en c

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

A diagram showing the general structure of a C program. It is enclosed in a rectangular box with an orange border. The box has a folded bottom-right corner, indicated by a grey triangle. The text inside the box is as follows:

```
<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>
```

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

Les directives de compilation

1. Include:

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.

2. Define :

Syntaxe : `#define expression_à_remplacer expression_de_replacement`

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

Les directives de compilation

Exemple

#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*/

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 */`

Exemple

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

Les variables et les constantes

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*

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

Exemple

```
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 */
```

Les variables et les constantes

DECLARATION DE VARIABLES

- 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, ...).

Les variables et les constantes

DECLARATION DE VARIABLES

- Identificateurs

Exemple

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

Les variables et les constantes

DECLARATION DE VARIABLES

- Type 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.

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

- La fonction **sizeof** retourne la taille en octets d'un objet.
- Exemple : `n=sizeof(int);` `/* n reçoit 2 */`

Les variables et les constantes

DECLARATION DE VARIABLES

- fonctions prédéfinies sur les types simples

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

1. Fonctions mathématiques

Math.h

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

```
#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)*/
}
```

Les variables et les constantes

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.

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
isctrl (c)	c est un caractère de contrôle différent d'espace et (<32) ou delete (0177)
isascii (c)	c est un caractère ASCII ($0 \leq c < 128$)

Exercice

Ecrire un programme qui affiche le nombre d'octets réservés sur votre machine pour les types :

- int, short int et long int
- float, double et long double
- char

Utiliser la fonction `sizeof ()` dans un `printf ()`.

Les opérateurs

1. L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe `=`. Sa syntaxe est la suivante :

variable = expression

- Cette expression a pour effet d'évaluer expression et d'affecter la valeur obtenue à variable.
- La variable peut être une variable simple ou bien un élément de tableau et non pas une constante.
- L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche.

Les opérateurs

1. L'affectation

Exemples

```
main()
{
    int i, j = 2;
    float x = 2.5;
    i = j + x;
    x = x + i;
    printf("\n %f \n", x);
}
```

```
main( )
{
    int i;
    float j=3.5;
    i=j;
}
```

Les opérateurs

2. Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires :

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants.

Les opérateurs

3. Les opérateurs relationnels

>	strictement supérieur
>=	supérieur ou 'égal
<	strictement inférieur
<=	inférieur ou 'égal
==	égal
!=	diffèrent

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Les opérateurs

4. Les opérateurs logiques booléens

<code>&&</code>	et logique
<code> </code>	ou logique
<code>!</code>	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon.

Les opérateurs

5. Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont :

$+=$ $-=$ $*=$ $/=$ $\%=$ $\&=$ $\wedge=$ $|=$ $\ll=$ $\gg=$

Pour tout opérateur op , l'expression

$Expression_1\ op = expression_2$

est équivalente à

$Expression_1 = expression_1\ op\ expression_2$

Les opérateurs

6. Les opérateurs d'incrémentation de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en suffixe (**i++**) qu'en préfixe (**++i**).

- Dans les deux cas la variable **i** sera **incrémentée**, toutefois dans la notation **suffixe** la valeur retournée sera **l'ancienne** valeur de **i** alors que dans la notation **préfixe** se sera la **nouvelle**.

Exemple:

```
int a = 3, b, c;  
b = ++a;      /* a et b valent 4 */  
c = b++;      /* c vaut 4 et b vaut 5 */
```

Les opérateurs

7. Les opérateurs conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

condition ? expression-1 : expression-2

Cette expression est égale à expression-1 si condition est satisfaite, et à expression-2 sinon.

Exemple:

`x >= 0 ? x : -x`



La valeur absolue d'un nombre

`m = ((a > b) ? a : b);`



Affecte à m le maximum de a et de b.

Les entrées / Sorties

1. Affichage

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

Syntaxe :

1. Affichage de message constitué de texte et de caractères de contrôle

Printf("texte et caractères de contrôle");

2. Affichage de valeurs de variables ou d'expressions

Printf("message : ", arg1, arg2,...,argn);

Noms de variables ou expressions

Les entrées / Sorties

1. Affichage

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

Les entrées / Sorties

1. Affichage Exemple

Affichage

```
main()  
{  
  
    int i=2,k=5;  
    float j=3.5;  
  
    printf("Donnez le prix unitaire");  
  
    printf("Donnez le prix unitaire \n");  
  
    printf("la valeur de i est %d\n ",i);  
  
    printf("i=%d    j=%f",i,j);  
  
    printf("i=%d\nj=%f",i,j);  
  
    printf("somme(%d,%d)=%d\nFIN",i,k,i+k);  
  
}
```

ix
,

Les entrées / Sorties

2. La lecture des données

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

Syntaxe :

`scanf("formats d'affichage", variable1, ...,variablen) ;`

Remarque :

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

Les entrées / Sorties

1. Lecture

Liste des formats de lecture

déclaration	lecture	écriture
int i; int i; int i; unsigned int i;	scanf("%d",&i); scanf("%o",&i); scanf("%x",&i); scanf("%u",&i);	printf("%d",i); printf("%o",i); printf("%x",i); printf("%u",i);
short j; short j; short j; unsigned short j;	scanf("%hd",&j); scanf("%ho",&j); scanf("%hx",&j); scanf("%hu",&j);	printf("%d",j); printf("%o",j); printf("%x",j); printf("%u",j);
long k; long k; long k; unsigned long k;	scanf("%ld",&k); scanf("%lo",&k); scanf("%lx",&k); scanf("%lu",&k);	printf("%d",k); printf("%o",k); printf("%x",k); printf("%u",k);
float l; float l; float l;	scanf("%f",&l); scanf("%e",&l);	printf("%f",l); printf("%e",l); printf("%g",l);
double m; double m; double m;	scanf("%lf",&m); scanf("%le",&m);	printf("%f",m); printf("%e",m); printf("%g",m);
long double n; long double n; long double n;	scanf("%Lf",&n); scanf("%Le",&n);	printf("%Lf",n); printf("%Le",n); printf("%Lg",n);
char o; char p[10];	scanf("%c",&o); scanf("%s",p); scanf("%s",&p[0]);	printf("%c",o); printf("%s",p);

Les entrées / Sorties

2. La lecture des données

Exemple

Lecture

```
#include <stdio.h>
main( )
{
  int i;
  float k;
  char m;
  scanf("%d",&i);
  scanf("%d%f",&i,&k);

  scanf("%c",&m);

}
```

Les instructions sélectives

1. Instruction si (if)

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

Syntaxe :

```
if (condition)  
{  
    Instruction_1;  
    Instruction_2;  
    ...  
    instruction_n;  
}
```

Les instructions sélectives

1. Instruction si (if)

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

Syntaxe :

```
if (condition)  
    {  
        Instruction_1;  
        ...  
        instruction_n;  
    }  
else  
    {  
        Instructions;  
    }
```

Traitement 1

Traitement 2

Les instructions sélectives

2. Instruction switch

Elle réalise un **aiguillage vers différentes instructions** en fonction du **contenu d'une variable** de contrôle.

❑ 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
}

```

Les instructions sélectives

2. Instruction switch

Exemple :

```
#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;
    }
}
```

Les instructions itératives

1. 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 ;)
}
```

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 quand la condition devient fausse.

Les instructions itératives

1. Instruction tant que (*while*)

Exemple :

```
#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*/
}
```

Les instructions itératives

2. 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) ;
```

Les instructions itératives

2. Instruction faire tant que (do while)

Exemple :

```
#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  $i \leq n$ */
    {
        somme=somme+i
        i++;
    }
    while (i<=n)                ;
    printf ("%d",somme); /*le programme affiche la somme des nombres compris entre 0
                           et n*/
}
```

Les instructions itératives

2. Instruction faire tant que (do while)

Exemple :

```
#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++;
    }
}
```

Les instructions itératives

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 ;)  
  }
```

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

Les instructions itératives

3. L'instruction pour (for)

Exemple :

```
#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*/

}
```

Les instructions itératives

4. Les instructions de sorties de boucles

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

Exemple :

```
#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*/
}
```

Les instructions itératives

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 ;
```

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

Les instructions itératives

5. L'instruction aller a (goto)

Exemple :

```
#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*/
```

Exercices

Exercice 1

```
#include <stdio.h>

main ()
{
    char op ;
    int n1, n2 ;

    printf ("opération souhaitée (+ ou *) ?\n ") ;
    scanf ("%c", &op) ;

    printf ("donnez 2 nombres entiers : \n") ;
    scanf ("%d %d", &n1, &n2) ;

    if (op == '+') printf ("leur somme est : %d \n", n1+n2) ;
    else printf ("leur produit est : %d \n", n1*n2) ;
    getch () ; } // Fin du programme
```

Exercices

Exercice 2

```
#include <stdio.h>

main ()
{
    char op ;
    int n1, n2 ;

    printf ("opération souhaitée (+ ou *) ?\n ") ;
    scanf ("%c", &op) ;

    printf ("donnez 2 nombres entiers : \n") ;
    scanf ("%d %d", &n1, &n2) ;

    if (op == '+') printf ("leur somme est : %d \n", n1+n2) ;
    else printf ("leur produit est : %d \n", n1*n2) ;
    getch () ; } // Fin du programme
```

Exercices

Exercice 3

Soient les déclarations suivantes :

int i; short int j; long int k; float x; double y; long double z; char c;

- Ecrire un programme qui lit au clavier toutes ces variables et affiche leur adresse ainsi que leur valeur respective. Quel est l'ordre de placement de ces variables en mémoire ?
- Pour afficher l'adresse d'une variable var, utilisez &var dans un printf ().
- **Exemple pour float x :**
printf ("La valeur de x est : %f et son adresse est : %d \n", x, &x);

Exercices

Exercice 4

Ecrire un programme qui lit au clavier les valeurs de trois résistances et de trois capacités et calcule leur résistance et leur capacité équivalente, respectivement, dans les deux cas :

- Les trois résistances et les trois capacités sont placées en série.
- Les trois résistances et les trois capacités sont placées en parallèle

Les résultats doivent être affichés dans chaque cas. Toutes les variables utilisées dans ce programme doivent être déclarées avec le type double.

Exercices

Exercice 5

Ecrire un programme qui lit trois nombres entiers A, B et C et affiche leur valeur maximale.

Utiliser les deux méthodes suivantes :

- a) if - else et une variable d'aide MAX
- b) if - else if - ... - else sans variable d'aide

Exercices

Solution

```
{  
  int A, B, C;  
  int MAX;  
  printf("Introduisez trois nombres entiers :");  
  scanf("%d %d %d", &A, &B, &C);  
  if (A>B)  
    MAX=A;  
  else  
    MAX=B;  
  if (C>MAX)  
    MAX=C;  
  printf("La valeur maximale est %d\n", MAX);  
  return 0;  
}
```

Exercices

Solution

```
b) if - else if - ... - else sans variable d'aide
int A, B, C;
printf("Introduisez trois nombres entiers :");
scanf("%d %d %d", &A, &B, &C);
printf("La valeur maximale est ");
if (A>B && A>C)
    printf("%d\n",A);
else if (B>C)
    printf("%d\n",B);
else
    printf("%d\n",C);
```

Exercices

Exercice 6

Ecrire un programme qui calcule la distance entre deux points A et B du plan Oxy.

Le programme doit lire les coordonnées de A et B. Afficher le résultat final.

Exercices

Solution

```
#include <stdio.h>
#include <math.h>
main()
{
    int XA, YA, XB, YB;
    double DIST;
    /* Attention: La chaîne de format que nous utilisons */
    /* s'attend à ce que les données soient séparées par */
    /* une virgule lors de l'entrée. */
    printf("Entrez les coordonnées du point A : XA,YA ");
    scanf("%d,%d", &XA, &YA);
    printf("Entrez les coordonnées du point B : XB,YB ");
    scanf("%d,%d", &XB, &YB);
    DIST=sqrt(pow(XA-XB,2)+pow(YA-YB,2));
    printf("La distance entre A(%d,%d) et B(%d, %d) est %.2f\n",
XA, YA, XB, YB, DIST);
    return 0;
}
```

Chapitre 3

Les types composés

Les types composés

A partir des types prédéfinis du C (char , int, float), on peut créer de nouveaux types, appelés *types composés*, qui permettent de représenter des ensembles de données organisées.

Les types composés

A. Tableaux

1. Les tableaux : déclaration

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

Syntaxe :

***type** nom-du-tableau[nombre-éléments];*

- **nombre-éléments** : est une expression constante entière positive.

Exemple :

int T[6]; \equiv

- Tableau de 6 éléments de type **int**
- Alloue en mémoire 6*4 octets consécutif pour l'élément T.

Les types composés

1. Les tableaux : déclaration

- On accède à un élément du tableau en lui appliquant l'opérateur []. Les éléments d'un tableau sont toujours numérotés de 0 à **nombre-éléments-1**.
- Il est recommandé de donner un nom à la constante nombre-éléments par une directive au préprocesseur, par exemple :

```
#define nombre-elements 10
```

- 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.

Les types composés

2. Les tableaux : Initialisation

- On peut initialiser un tableau lors de sa d'déclaration par une liste de constantes de la façon suivante :

Syntaxe :

***type** nom-du-tableau[N] = {constante1, constante ... , constante-N};*

Les types composés

2. Les tableaux : Initialisation

Exemple :

```
#define N 4
int tab[N] = {1, 2, 3, 4}; main()
{
int i;
for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i, tab[i]);
}
```

Les types composés

3. Les tableaux : Lecture et Affichage

Exemple :

```
#define N 10
main()
{
    int tab[N];
    int i;
    /*lecture des éléments du tableau
    for (i = 0; i < N; i++)
        scanf("%d", &tab[i]);
    /*Affichage des éléments du tableau
    for (i = 0; i < N; i++)
        printf("tab[%d] = %d\n", i, tab[i]);
}
```

Les types composés

4. Les tableaux : Affectation

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

- 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.

Les types composés

4. Les tableaux : Comparaison

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

```
#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,*/
            {egaux=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");
}
```

Les types composés

4. Les tableaux à deux dimension

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau.

Syntaxe :

```
type nom-du-tableau[nombre-lignes][nombre-colonnes];
```

- On accède à un élément du tableau par l'expression :
tableau[i][j]
- Pour initialiser un tableau `a plusieurs dimensions `a la compilation, on utilise une liste dont chaque `el`ement est une liste de constantes :

Les types composés

4. Les tableaux à deux dimension

- Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

Exemple :

```
#define M2
#define N3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main()
{
    int i, j;
    for (i = 0 ; i < M; i++)
    {
        for (j = 0; j < N; j++)
            printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```


Les types composés

4. Les tableaux à deux dimensions

Lecture des éléments du tableau à deux dimensions

```
#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]);
}
```

Les types composés

B. Chaines de Caractères

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.

1. Initialisation

Syntaxe :

```
char Identificateur [Taille constante] = "Texte" ;
```

Les types composés

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`** et **`puts`**.

Syntaxe :

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

Les types composés

2. Lecture et affichage

Exemple :

```
#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*/
}
```

Les types composés

3. Fonctions sur les chaînes de caractères

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

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.

Les types composés

3. Fonctions sur les chaînes de caractères

Exemple :

```
#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)*/
}
```

Les types composés

C. Les structures

Une structure est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé membre ou champ, est désigné par un identificateur.

Les types composés

C. Les structures

1. Déclaration

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est *modele* suit la syntaxe suivante :

Syntaxe :

```
struct    modele
{
    type-1 membre-1;
    type-2 membre-2;
    ...|
    type-n membre-n;
};
```


Les types composés

C. Les structures

1. Déclaration

Pour déclarer un objet de type structure, on utilise la syntaxe :

Syntaxe :

```
struct modele objet;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct    modele
{
    type-1 membre-1;
    type-2 membre-2;
    ...
    type-n membre-n;
} objet;
```

Les types composés

C. Les structures

2. Accéder

On accède aux différents membres d'une structure grâce à l'opérateur membre de structure, noté ".". Le i-ème membre de objet est désigné par l'expression

Syntaxe :

objet.membre-i

On peut effectuer sur le i-ème membre de la structure toutes les opérations valides sur des données de type type-i.

Les types composés

C. Les structures

3. Initialisation

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe z = {2. , 2.};
```

Les types composés

C. Les structures

4. Exemple

```
#include <math.h> struct complexe
{
    double reelle;
    double imaginaire;
};

main()
{
    struct complexe z; double norme;
    ...
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
    printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

Les types composés

C. Les structures

5. Définition de types composés avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **typedef** :

Syntaxe :

```
typedef type synonyme;
```

Exemple :

```
struct complexe
{
    double reelle; double imaginaire;};
typedef struct complexe complexe;

main()
{
    complexe z;
    ...
}
```

Chapitre 4

Les pointeurs

Les Pointeurs

Définition

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

1. Déclaration 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*/

Les Pointeurs

Définition

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

1. Déclaration 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*/

Opérateurs & et *

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

Exemples :

```
#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*/
}
```

Opérateurs ++ et --

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

Exemples :

```
#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)*/
}
```

Allocation mémoire

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**.

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 5

Les fonctions

Structure et prototype d'une fonction

Définition

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

Structure et prototype d'une fonction

1. Définition de la fonction

Syntaxe :

```
Void ou Type_Résultat Nom_Fonction (Void ou Type Param1,...,Type Paramn)  
{  
    <Déclaration des variables locales>  
    Corps de la fonction (instructions)  
}
```

Void ou Type_Résultat : indique le **type du résultat** retourné par la fonction **ou void** si elle ne retourne rien.

Nom_Fonction : Indique le nom de la fonction.

Void ou Type Param1 : Elle indique également ses **paramètres formels** et leurs types si elle en admet, sinon indique **void**.

Structure et prototype d'une fonction

1. Définition de la fonction

- **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.

Structure et prototype d'une fonction

1. Définition de la fonction

Exemple 1 : fonction affiche

```
void afficher(void)    /*affiche le message bonjour*/  
{  
    printf("bonjour");  
}
```

Exemple 2 : fonction triple

```
int triple (int n)  
{  
    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*/  
}
```

Structure et prototype d'une fonction

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.

Syntaxe :

```
Void ou Type_Résultat  Nom_Fonction (Void ou Type1,...,Typen);
```

Exemples :

```
void afficher (void);  
int triple (int);
```

Structure et prototype d'une fonction

2. Appel 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.

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 ();

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 :

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.

Passage de paramètres

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 :

```
#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*/
}
```

Passage de paramètres

2. Passage par adresse (par référence)

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 *****.

Passage de paramètres

2. Passage par adresse (par référence)

Exemple :

```
#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){
*j=3*i;
printf ("%d",*j);
}
```

Passage de paramètres

2. Passage par adresse (par référence)

- ❑ 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).

Passage de paramètres

2. Passage par adresse (par référence)

Exemple :

```
#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]);
}
```

Fonction récursive

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.

Fonction récursive

Exemple :

```
#include <stdio.h>
int puissance (int, unsigned);
main( ){
int x; unsigned y;                /*x entier*/ /*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)
    return (1); /*retourne 1 si y=0*/
else
    return(x*puissance (x,y-1)); /*retourne  $x \cdot x^{y-1}$  (appelle de nouveau puissance
                                pour x et y-1)*
}
```

Exercices

Exercice 1 :

Ecrire un programme qui lit 10 notes d'un étudiant et affiche la moyenne.

Exercice 2 :

Ecrire un programme qui détermine la plus grande et la plus petite valeur dans un tableau d'entiers A. Afficher ensuite la valeur et la position du maximum et du minimum. Si le tableau contient plusieurs maxima ou minima, le programme retiendra la position du premier maximum ou minimum rencontré.

Exercice 3 :

Faire l'addition de deux matrices A et B. Afficher le résultat.

Exercice 4 :

Écrire un programme qui définit deux structures Point et Cercle. Le programme doit lire et afficher les champs respectifs des variables de type structure Point, Cercle

Exercices

Exercice 5 :

Soit P un pointeur qui "pointe" sur un tableau A :

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
```

```
int *P;
```

```
P = A;
```

Quelles valeurs ou adresses fournissent les expressions suivantes :

a- $*P+2$

b- $*(P+2)$

c- $\&P+1$

d- $\&A[4]-3$

e- $A+3$

f- $\&A[7]-P$

g- $P+(*P-10)$

h- $*(P+*(P+8)-A[7])$

Exercices

Exercice 6 :

Ecrire un programme qui lit une ligne de texte (ne dépassant pas 200 caractères) la mémorise dans une variable TXT et affiche ensuite :

- a) la longueur L de la chaîne.
- b) le nombre de 'e' contenus dans le texte.
- c) toute la phrase à rebours, sans changer le contenu de la variable TXT.
- d) toute la phrase à rebours, après avoir inversé l'ordre des caractères dans TXT. Exemple :

Exercices

Exercice 7 :

Écrire un programme qui implémente une fonction qui lit les données relatives à un étudiant appartenant à une institution universitaire telles que son nom, son prénom, son CNE ainsi que les notes obtenues dans 12 modules. Le programme principal doit afficher toutes ces données lues par la fonction ainsi que la moyenne des notes