Ce livre est destiné aux étudiants de la première année Licence Mathématiques et Informatique, et à tous ceux qui souhaitent acquérir des bases solides en algorithmique et en structures de données. Les algorithmes de ce livre sont traduits en langage C.
Proposant un apprentissage progressif, ce livre s'appuie largement sur notre expérience d'enseignement de la matière "Algorithmique et structures de données" pendant plusieurs années. A la fin de chaque chapitre, il y a un ensemble d'exercices corrigés. Après avoir lu et compris le cours, l'étudiant est conseillé d'essayer de résoudre les exercices par lui-même avant de consulter la correction. L'étudiant ne doit pas oublier qu'un même problème peut être résolu par différents algorithmes.
L'auteur de ce livre sera très reconnaissant de recevoir toute remarque ou suggestion.

Prof. Djelloul BOUCHIHA est un Enseignant-Chercheur au Centre Universitaire - SALHI Ahmed - de Naâma, Algérie. Il enseigne comme modules : "Algorithmique et structures de données" et "Programmation mobile". Sur le plan scientifique, il est à la tête de l'équipe de recherche DOS, rattachée au laboratoire EEDIS, UDL-SBA.

978-613-9-57231-1

Algorithmique et programmation en C

BOUCHIHA

ÉDITIONS
UNIVERSITAIRES
EUROPÉENNES

Djelloul BOUCHIHA

Algorithmique et programmation en C

Cours avec 200 exercices corrigés

Djelloul BOUCHIHA

Algorithmique et programmation en C

Djelloul BOUCHIHA

Algorithmique et programmation en C

Cours avec 200 exercices corrigés

Éditions universitaires européennes

Imprint

Cover image: www.ingimage.com

Publisher:
Éditions universitaires européennes
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius

Printed at: see last page
ISBN: 978-613-9-57231-1

Algorithmique et programmation en C

Cours avec 200 exercices corrigés

Par

Prof. Djelloul BOUCHIHA

CENTRE UNIVERSITAIRE -SALHI AHMED- DE NAAMA

EVOLUTIONARY ENGINEERING AND DISTRIBUTED INFORMATION SYSTEMS LABORATORY

(EEDIS) UDL-SBA

djelloul.bouchiha@univ-sba.dz; bouchiha.dj@gmail.com

Table des matières

Avant-propos **9**

Chapitre 1 : Introduction **10**
1. Bref historique sur l'informatique 10
1.1. Définition 10
1.2. Parties de l'informatique 10
1.3. L'ordinateur 10
1.3.1. Définition 10
1.3.2. Historique 10
1.3.3. Composantes d'un ordinateur 10
1.4. Unités de mesure de la mémoire 12
1.4.1. Structure du disque dur 12
1.4.2. Tailles de quelques mémoires 13
1.5. La relation Homme/Machine 14
1.5.1. Système d'exploitation 14
1.5.2. Fichier et répertoire 15
2. Introduction à l'algorithmique 15
2.1. Résolution d'un problème par ordinateur 15
2.2. Notion d'algorithme 15
2.3. Exemple 15
3. Exercices corrigés 21
3.1. Exercices 21
3.2. Corrigés 22

Chapitre 2 : Les algorithmes séquentiels simples **24**
1. Parties d'un algorithme 24
2. Les données 24
3. Les types 25
4. Opérations de base 26
4.1. L'affectation 26
4.2. Les entrées/sorties 27
5. Construction d'un algorithme simple 27
6. Représentation d'un algorithme par un organigramme 28
7. Traduction en langage C 28
7.1. Exemple 28
7.2. Manipulation des nombres 29
7.3. Manipulation des caractères 30

7.4. Manipulation des booléens 32
7.5. Opérateurs en C 32
7.6. Autres possibilités et règles en C 35
8. Exercices corrigés 42
8.1. Exercices 42
8.2. Corrigés 46

Chapitre 3 : Les structures conditionnelles 58
1. Introduction 58
2. Structure conditionnelle simple 58
3. Structure conditionnelle composée 59
4. Structure conditionnelle multiple 60
5. Le branchement 62
6. Exercices corrigés 63
6.1. Exercices 63
6.2. Corrigés 64

Chapitre 4 : Les boucles 73
1. Introduction 73
2. La boucle Tant que 73
3. La boucle Répéter 74
4. La boucle Pour 75
5. Les boucles imbriquées 77
6. Exercices corrigés 78
6.1. Exercices 78
6.2. Corrigés 80

Chapitre 5 : Les tableaux et les chaînes de caractères 93
1. Introduction 93
2. Le type tableau 93
2.1. Manipulation d'un tableau 94
2.2. Tri d'un tableau 100
2.3. Tableau à deux dimensions 102
3. Les chaînes de caractères 103
3.1. Déclaration d'une chaîne de caractères 103
3.2. Manipulation des chaînes de caractères 103
3.3. Tableau de chaînes de caractères 107
4. Exercices corrigés 107
4.1. Exercices 107
4.2. Corrigés 111

Chapitre 6 : Les sous-programmes : Procédures et Fonctions 126
1. Introduction 126
2. Les sous-programmes 128
3. Les variables locales et les variables globales 134
4. Le passage des paramètres 137
5. La récursivité (récursion) 142
5.1. Définition 142
5.2. Exemple (Calcul de la factorielle) 142
6. Exercices corrigés 144
6.1. Exercices 144
6.2. Corrigés 148

Chapitre 7 : Les types personnalisés 161
1. Introduction 161
2. Définition des types en C 161
3. Le type structure 161
4. Le type union 164
5. Le type énumération 166
6. Exercices corrigés 168
6.1. Exercices 168
6.2. Corrigés 171

Chapitre 8 : Les fichiers 183
1. Introduction 183
2. Les fichiers 183
2.1. La structure FILE 183
2.2. Ouverture d'un fichier 184
2.3. Fermeture d'un fichier 185
2.4. Accès au fichier 186
2.4.1. Lecture/écriture non-formatées en mode caractère 186
2.4.2. Lecture/écriture non-formatées en mode chaîne 187
2.4.3. Lecture/écriture formatées 188
2.4.4. Lecture/écriture par bloc 189
2.4.5. Accès direct 190
3. Exercices corrigés 191
3.1. Exercices 191
3.2. Corrigés 193

Chapitre 9 : Les listes chaînées 205
1. Introduction 205

2. Les pointeurs 205
3. Gestion dynamique de la mémoire 206
4. Les listes chaînées 208
5. Opérations sur les listes chaînées 210
5.1. Créer et remplir une liste 210
5.2. Afficher les éléments de la liste 212
5.3. Ajouter un élément au début de la liste 213
5.4. Insérer un élément dans la liste 214
5.5. Supprimer la tête de la liste 215
5.6. Supprimer un élément de la liste 216
6. Les listes doublement chaînées 217
7. Les listes chaînées particulières 220
7.1. Les piles 220
7.1.1. Primitives d'accès 221
7.1.2. Représentation d'une pile par une liste doublement chaînée 221
7.2. Les files 222
7.2.1. Accès à une file 223
7.2.2. Représentation d'une file par une liste doublement chaînée 223
8. Exercices corrigés 223
8.1. Exercices 223
8.2. Corrigés 234

Chapitre 10 : Les arbres 254
1. Introduction 254
2. Définitions 254
3. Arbre binaire 255
3.1. Définition 255
3.2. Passage d'un arbre n-aire à un arbre binaire 255
3.3. Représentation chaînée d'un arbre binaire 255
3.4. Parcours d'un arbre binaire 257
3.4.1. Parcours préfixé (préordre ou RGD) 257
3.4.2. Parcours infixé (projectif, symétrique ou encore GRD) 258
3.4.3. Parcours postfixé (ordre terminal ou GDR) 258
3.5. Arbres binaires particuliers 258
3.5.1. Arbre binaire complet 258
3.5.2. Arbre dégénéré 259
3.5.3. Arbre binaire ordonné 259
4. Exercices corrigés 259
4.1. Exercices 259
4.2. Corrigés 262

Chapitre 11 : Les graphes 269
1. Introduction 269
2. Définitions 269
3. Représentation d'un graphe 269
3.1. Liste d'adjacence 269
3.2. Matrice d'adjacence 270
4. Parcours d'un graphe 271
4.1. Parcours en largeur d'abord 271
4.2. Parcours en profondeur d'abord 271
5. Exercices corrigés 272
5.1. Exercices 272
5.2. Corrigés 273

Références bibliographiques additionnelles 280

INDEX 281

Avant-propos

Ce livre est destiné aux étudiants de la première année Licence Mathématiques et Informatique, et à tous ceux qui souhaitent acquérir des bases solides en algorithmique et en structures de données. Les algorithmes de ce livre sont traduits en langage C.

Proposant un apprentissage progressif, ce livre s'appuie largement sur notre expérience d'enseignement de la matière "Algorithmique et structures de données" pendant plusieurs années. A la fin de chaque chapitre, il y a un ensemble d'exercices corrigés. Après avoir lu et compris le cours, l'étudiant est conseillé d'essayer de résoudre les exercices par lui-même avant de consulter la correction. L'étudiant ne doit pas oublier qu'un même problème peut être résolu par différents algorithmes.

L'auteur de ce livre sera très reconnaissant de recevoir toute remarque ou suggestion.

Chapitre 1 : Introduction

1. Bref historique sur l'informatique

1.1. Définition

Le mot informatique vient de deux mots : **informa**tion et auto**matique**.
L'informatique est le traitement automatique de l'information.
Traitement : c'est la création, la suppression, la modification et le stockage.
Automatique : en utilisant une machine.
Information : il faut distinguer entre donnée et information.

- Donnée : c'est un élément brut, par exemple, *5*, *jaune*, *dimanche*, *ahmed*...
- Information : c'est une donnée interprétée, par exemple, *5 est un nombre*.

1.2. Parties de l'informatique

L'étude de l'informatique nous ramène à étudier deux parties :

- HARDWARE : c'est la partie physique (matériel) dans un ordinateur, par exemple, circuits intégrés, câbles électriques, disques magnétiques, etc.
- SOFTWARE : c'est la partie programme (logiciel) dans un ordinateur, par exemple, système d'exploitation, programmes d'application, etc.

1.3. L'ordinateur

1.3.1. Définition

C'est une machine capable de traiter de l'information.

1.3.2. Historique

On peut dire que la deuxième guerre mondiale a donné naissance aux premiers ordinateurs :

- Les Z2 et Z3 de l'allemand Zuse ont été prêts en 1939 et 1941.
- Ensuite, la série des "Automatic Sequence Controlled Computer Mark" conçue par Howard Aiken. Le Mark I fonctionnera en 1944.
- L'ENIAC (Electronic Numerical Integrator Analyser and Computer) (1946) de Prosper Eckert et John Mauchly, souvent référencé comme le premier ordinateur. Il pesait 30 tonnes occupant une superficie de 160 mètres carrés et une hauteur de 3 mètres.
- Ensuite, la 2de génération d'ordinateurs où on a utilisé les transistors, les disques magnétiques, etc.
- En 1963, on a utilisé les circuits intégrés (3^{e} génération).
- Les unités centrales réalisées sous de très faibles volumes ont été largement diffusées en 1976.

1.3.3. Composantes d'un ordinateur

Un PC (Personal Computer) est constitué principalement de trois parties :

A. Unité centrale : elle contient :
   a. La carte mère : elle regroupe les composantes de l'ordinateur.

b. Le processeur : c'est le cerveau de l'ordinateur, constitué à son tour d'une UC (unité de commandes) et d'une UAL (unité arithmétique et logique).

Toute action ou activité faite par l'ordinateur, son ordre provient de l'UC.

L'UAL permet de calculer les résultats des expressions arithmétiques et logiques.

Les expressions arithmétiques : elles servent à effectuer des calculs dont les résultats sont de type numérique. Ces expressions comportent :

- Des opérandes de type numérique.
- Des opérateurs : `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), etc.
- Des fonctions mathématiques : `ln`, `sin`, `arctg`, etc.

Exemple : `5 + 4 / 2 + (7 - 1) = 13`.

L'ordre de priorité entre opérations est le suivant:

- Les fonctions mathématiques.
- La multiplication et la division.
- L'addition et la soustraction.

On peut utiliser les parenthèses pour clarifier, ou pour changer l'ordre de priorité. Par exemple, `4 * (2 + 9)` permet d'effectuer l'addition avant la multiplication.

Dans le cas d'égalité de priorité, les opérations sont effectuées de gauche à droite.

Les expressions logiques : elles servent à effectuer des calculs dont les résultats sont de type logique (booléen), en utilisant les opérateurs `NON`, `ET` et `OU` triés par ordre de priorité. En effet, les parenthèses changent l'ordre de priorité. Les opérandes logiques peuvent prendre deux valeurs : `VRAI` ou `FAUX`. Une expression logique implique éventuellement des opérateurs de comparaison, à savoir >, >=, <, <=, =, <>.

Remarque : Le symbole <> est utilisé dans ce cours pour indiquer la différence (≠).

Les équations logiques sont les suivantes :

NON	
VRAI	FAUX
FAUX	VRAI

ET	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

Exemple : `(10>5) ET (2<3) = VRAI`.

c. La mémoire : permet le stockage des informations. Il existe deux types de mémoire :

1. La mémoire centrale : comporte (1) la RAM (Random Access Memory) où les informations sont stockées pendant le traitement ;

(2) la ROM (Read Only Memory) qui stocke le BIOS (Basic Input Output System), le programme de démarrage de l'ordinateur.

2. La mémoire auxiliaire : elle permet le stockage permanent des informations. La mémoire auxiliaire peut être un disque dur, un CD-ROM, une clé USB, etc.

d. En plus, d'autres dispositifs, tels que le lecteur CD-ROM, les câbles, les nappes, etc.

B. L'écran : permet de sortir (afficher) les informations à l'utilisateur.

C. Le clavier : permet d'entrer (acquérir) les informations à la machine.

Un PC peut être accompagné d'un ensemble de périphériques pour faciliter son utilisation et augmenter son efficacité. Il existe : la souris, l'imprimante, le scanner, le graveur, etc.

1.4. Unités de mesure de la mémoire

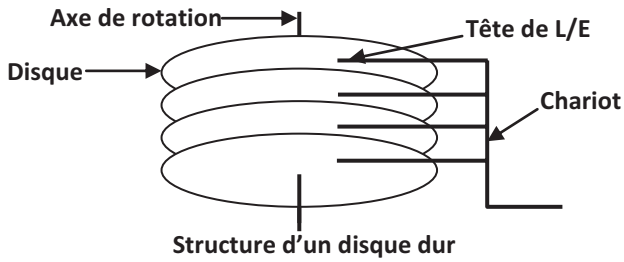
L'unité de mesure de la mémoire est l'octet, noté *O* (en anglais byte). 1 octet correspond à un caractère. 1 octet = 8 bit. Un bit est soit 0, soit 1. Par exemple, le caractère 'A', dont le code ASCII est 65, est stocké 01000001.

Remarques :

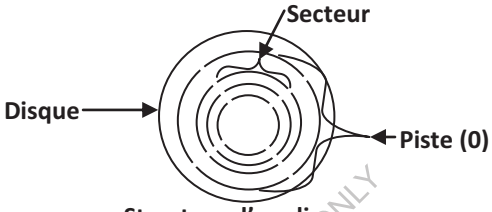
- ASCII (American Standard Code for Information Interchange) : est une norme d'encodage informatique des caractères alphanumériques de l'alphabet latin.
- La représentation du 0 et 1 dépend du support de stockage.
  - ☞ Dans la RAM, le 0 et 1 sont représentés par une charge électrique : chargé (1) ou non chargé (0).
  - ☞ Dans un support magnétique, ça dépend du sens de la magnétisation. Par exemple, dans un disque dur, les secteurs sont divisés en îlots d'aluminium. Pour écrire un 1, il faut que deux îlots placés à côté l'un de l'autre aient un sens magnétique différent. Sinon, on écrit un 0.
  - ☞ Dans un CD-ROM, c'est l'alvéole qui définit l'information. On nomme creux le fond de l'alvéole, et on nomme plat les espaces entre les alvéoles. Le passage d'un creux à un plat (ou l'inverse) représente un 1. Le reste représente un 0.
- Contrairement au disque dur, un CD-ROM n'a qu'une seule piste organisée en spirale.

1.4.1. Structure du disque dur

Le disque dur est constitué d'un ensemble de disques superposés à double-face, i.e. les données sont stockées sur les deux côtés de chaque disque. Un disque ne peut être utilisé qu'après un formatage de bas niveau qui consiste à le subdiviser en pistes. Une piste contient un ensemble de secteurs. Un secteur contient 512 O. Un ensemble de pistes superposées est appelé cylindre.



Structure d'un disque dur



Structure d'un disque

On a les règles :

1 Kilo-octet (KO) = 1024 O = (2^{10}) O
1 Méga-octet (MO) = 1024 KO = $(1024)^2$ O
1 Giga-octet (GO) = 1024 MO = $(1024)^2$ KO = $(1024)^3$ O
1 Téra-octet (TO) = 1024 GO
1 Péta-octet (PO) = 1024 TO

1.4.2. Tailles de quelques mémoires

Voici les tailles de quelques mémoires :

- Disque dur : 320 GO, 500 GO, 1 TO, etc.
- RAM : 512 MO, 1 GO, 2 GO, etc.
- CD-ROM : 700 MO.
- DVD : 4,7 GO.
- Clé USB : 256 MO, 512 MO, 1 GO, 4 GO, etc.

Remarques :

- Un CD-ROM de l'anglais « Compact Disc Read Only Memory » signifie « disque compact en lecture seule ».
- Un DVD de l'anglais « Digital Versatile Disc » signifie « disque numérique polyvalent ».

Exercice : Quel est le nombre de CD-ROM nécessaires pour stocker le contenu d'un DVD plein.

Solution : On utilise la règle de trois (règle de proportionnalité) pour résoudre ce problème.

1 DVD → 4,7 GO
1 GO → 1024 MO
4,7 GO → X

$$X = \frac{1024 * 4,7}{1} = 4812,8MO$$

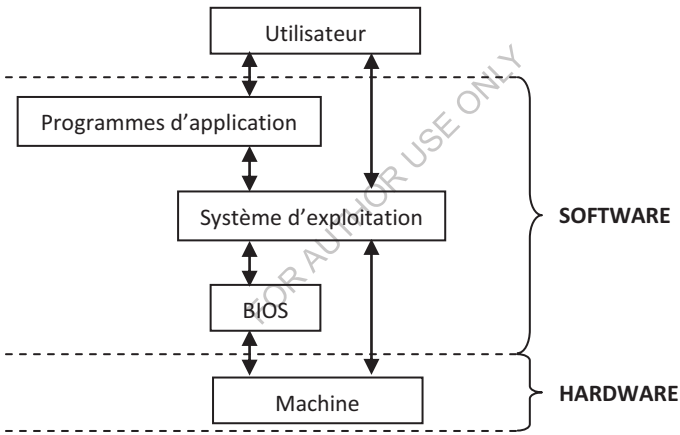
1 DVD → 4812,8 MO
1 CD-ROM → 700 MO
N → 4812,8 MO

$$N = \frac{4812,8}{700} = 6,8 \approx 7CDROM$$

Pour stocker le contenu d'un DVD plein, on aura besoin de 7 CD-ROM.

1.5. La relation Homme/Machine

L'utilisateur ne peut utiliser un ordinateur qu'à travers un système d'exploitation.



1.5.1. Système d'exploitation

C'est un ensemble de programmes permettant la gestion de l'ordinateur (gestion de la mémoire, gestion des fichiers, etc.). Il joue le rôle d'un intermédiaire entre l'utilisateur et la machine.

Exemples :

- MS DOS (Microsoft Disk Operating System) : un système d'exploitation produit par Microsoft de Bill Gates, basé sur les commandes (saisies au clavier), telles que *dir*, *md*, *rd*, etc.
- Windows : un système d'exploitation graphique (la souris est souvent utilisée pour la manipulation) produit par Microsoft.

Il existe d'autres systèmes d'exploitation produits par d'autres sociétés, tels que l'Unix, Solaris, OS2, etc.

1.5.2. Fichier et répertoire

Un fichier est un ensemble d'informations stockées sur un support physique (disque dur, disquette, RAM, etc.). Un fichier peut être un texte, une image, un fichier son, etc.

Un fichier possède un nom, une extension indiquant son type, une taille, date et heure de création ou de modification.

Un répertoire est une partie d'un support de stockage qui contient des fichiers et éventuellement d'autres sous-répertoires. L'objectif de la création des répertoires est l'organisation.

Exemple : Un fichier *Révolution.doc*, qui existe dans le sous-répertoire *Histoire*, qui à son tour existe dans le répertoire *Ouvrage*, qui se trouve dans le disque dur, est référencé par le chemin *C:\Ouvrage\Histoire\Révolution.doc*.

2. Introduction à l'algorithmique

2.1. 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.

2.2. Notion d'algorithme

Le mot algorithme provient du nom du célèbre mathématicien arabe : Mohamed Ibn Moussa El Khawarizmi (780-850).

Un algorithme est une suite d'opérations élémentaires exécutées dans un ordre donné pour résoudre un problème ou accomplir une tâche. En tant que science, on parle de l'algorithmique.

2.3. Exemple

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).

Cette forme de représentation d'un algorithme est dite *Enumération des étapes*.

C. Ecriture en LDA

En informatique, on utilise un langage de description d'algorithme (LDA) pour écrire un algorithme. Ainsi, l'algorithme ci-dessus devient :

```
Algorithme Somme
Variables
  Entier    valeur1, valeur2, som ;
Début
  Lire(valeur1, valeur2) ;
  som ← valeur1 + valeur2 ;
  Ecrire(som) ;
Fin
```

Comme illustré dans l'exemple précédent :

- Un algorithme commence par le mot `Algorithme`, suivi de son nom. Généralement, le nom de l'algorithme indique sa fonction.
- Le mot `Variables` précède la liste des variables manipulées dans l'algorithme et leurs types. Les variables du même type sont séparées par des virgules (`,`). Deux déclarations différentes sont séparées par un point-virgule (`;`).
- Les opérations de l'algorithme sont prises entre les mots `Début` et `Fin` indiquant le début et la fin de l'algorithme. Ces opérations sont séparées par des points-virgules.
- Le mot `Lire` permet la lecture à partir du clavier. Le mot `Ecrire` permet l'affichage à l'écran.
- Le symbole ← correspond à l'opération d'affection. Le symbole + est utilisé pour indiquer l'addition.

D'autres mots et symboles utilisés dans notre langage algorithmique seront découverts dans le reste de ce cours.

Remarque : Le LDA varie d'un document à un autre. Par exemple, le symbole d'affectation, représenté dans notre cours par ←, peut être représenté dans un autre document par `::-`.

Un algorithme doit être :

- Lisible : clair et facile à comprendre.
- De haut niveau : indépendant du langage de programmation et du système d'exploitation.
- Précis : non ambigu.
- Concis : réduire le nombre d'opérations ainsi que l'espace occupé en mémoire.
- Structuré : organisé.

D. Ecriture en langage de programmation

Pour être exécuté sur un ordinateur, un algorithme doit être traduit en un langage compréhensible par la machine, i.e. un langage de programmation, tel que Pascal, C, Java, etc.

Ainsi, l'algorithme de calcul de la somme, traduit en langage Pascal, donne :

```
program Somme ;
var  valeur1, valeur2, som : integer ;
begin
  read(valeur1, valeur2) ;
  som := valeur1 + valeur2 ;
  write(som) ;
end.
```

La traduction en langage C donne le programme :

```
#include <stdio.h>
main()
{
  int    valeur1, valeur2, som ;
  scanf("%d%d", &valeur1, &valeur2) ;
  som = valeur1 + valeur2 ;
  printf("%d", som) ;
}
```

Si l'utilisateur introduit `5` et `7` pour les deux variables (`valeur1` et `valeur2`) lors de l'exécution du programme, alors le résultat d'affichage sera : `12`

Il est possible d'améliorer le programme de calcul de la somme de la manière suivante:

```
#include <stdio.h>
main()
{
  int valeur1, valeur2, som ;
  printf("Entrez la première valeur : ") ;
  scanf("%d", &valeur1) ;
  printf("Entrez la deuxième valeur : ") ;
  scanf("%d", &valeur2) ;
  som = valeur1 + valeur2 ;
  printf("La somme de %d et %d est : %d ", valeur1, valeur2, som) ;
}
```


Le résultat d'exécution du programme précédent, si l'utilisateur introduit `5` et `7` pour `valeur1` et `valeur2`, est le suivant :

```
Entrez la première valeur : 5
Entrez la deuxième valeur : 7
La somme de 5 et 7 est : 12
```

Un programme est donc le résultat de la traduction d'un algorithme en un langage de programmation, i.e. un langage formel compréhensible par la machine.

Dans le reste de ce cours, tous les algorithmes seront traduits en C. Ce langage a été conçu en 1972 par Denis Ritchie pour écrire le système d'exploitation Unix.

Dans le programme C précédent, nous avons :

- L'en-tête `main()` indique que ce qui va être écrit par la suite est le programme principal. En LDA, ça correspond à `Algorithme`.

- Les deux accolades `{ }` correspondent respectivement à `Début` et `Fin` dans un algorithme.
- `int` est utilisé pour déclarer des entiers.
- L'affectation est exprimée en C par `=`. On note ici qu'il y a une grande différence entre l'égalité en mathématiques qui vise généralement une comparaison entre deux valeurs, et l'affectation en algorithmique qui vise à attribuer une valeur à une variable. L'opérateur d'égalité est représenté en C par le symbole `==`.
- La fonction `scanf` permet la lecture à partir du clavier. La fonction `printf` permet l'affichage à l'écran. Les deux fonctions appartiennent à la bibliothèque `stdio.h` incorporée au programme grâce à la directive de pré-compilation `#include`.

Avant d'avancer dans ce cours, nous allons monter pas à pas un programme C :

Exemple 1 :

```
#include <stdio.h>
main()
{
  printf("bonjour") ;
}
```

`printf` permet d'afficher une chaîne de caractères mise entre deux guillemets doubles. Ainsi, le programme de l'exemple 1 affiche : `bonjour`

Exemple 2 :

```
#include <stdio.h>
main()
{
  printf("bonjour\n") ;
  printf("monsieur") ;
}
```

`\n` indique le retour à la ligne suivante. Ainsi, le programme de l'exemple 2 affiche :

```
bonjour
monsieur
```

Sans le `\n`, le programme affichera : `bonjourmonsieur`

Notons aussi que la fonction `puts` affiche une chaîne de caractères et passe à la ligne suivante. Ainsi, `printf("bonjour\n")` est équivalente à `puts("bonjour")`.

Voici encore d'autres possibilités de représentation de caractères spéciaux :

Notation en C	Signification
`\a`	Cloche ou bip
`\b`	Retour arrière
`\f`	Saut de page
`\n`	Saut de ligne

\r	Retour chariot
\t	Tabulation horizontale
\v	Tabulation verticale
\\	\
\'	'
\"	"
\?	?
\0	caractère null
\Onn	nn en octal
\Xnn	nn en hexadécimal
%%	%

Exemple 3 :

```
#include <stdio.h>
main()
{
  int x ;
  x = 5 ;
  printf("valeur %d", x) ;
}
```


En général, la fonction `printf` peut avoir plusieurs arguments. Le premier argument, mis entre deux guillemets doubles, indique le format, i.e. un guide qui montre comment afficher les arguments suivants, s'ils existent. Le caractère `%` signifie que ce qui vient par la suite est un code de format. Le reste des arguments correspond aux valeurs à afficher.

Le `%d` dans le programme précédent sera remplacé par la valeur de `x` en décimal. Ainsi, le programme affiche : `valeur 5`

Dans le format transmis en premier argument de `printf`, tout ce qui n'est pas code de format est affiché tel quel. Si nous avions, par exemple, `printf("j'ai %d ans", x)` au lieu de `printf("valeur %d", x)`, ceci affiche : `j'ai 5 ans`

Notons aussi qu'il est possible en C de déclarer et d'initialiser une variable au même temps (`int x = 5`). La valeur d'une variable, juste après sa création, est inconnue. Pour éviter tout problème, il faut initialiser la variable quand on la déclare, ou bien juste après sa déclaration. L'initialisation consiste donc à donner à une variable une première valeur (valeur initiale).

D'autres formats sont aussi possibles : `d` pour décimal, `f` pour `float` (réel) et `c` pour caractère. L'instruction `printf("%c %d %d %f", 'A', 'A', 5, 5.2)` affiche : `A 65 5 5.200000`. Notons ici qu'un caractère est mis entre deux guillemets simples.

Exemple 4 :

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

```
  int x, y, z;
  x = 5 ;
  y = 2 ;
  z = x + y ;
  printf("La somme de %d et %d est : %d ", x, y, z) ;
}
```

Dans cet exemple, le programmeur a déclaré trois variables entières `x`, `y` et `z`. Il a initialisé le `x` à `5` et le `y` à `2`. Il a ensuite affecté le résultat d'addition de `x` et `y` à `z`.

Ainsi, le programme de l'exemple 4 affiche : `La somme de 5 et 2 est : 7`

On aurait pu remplacer l'instruction `printf("La somme de %d et %d est : %d ", x, y, z)` par `printf("La somme de %d et %d est : %d ", x, y, x+y)`. Comme ça, on aurait pu éviter la déclaration de la variable `z` et l'affectation `z = x + y`.

Exemple 5 :

```
#include <stdio.h>
main()
{
  int x, y ;
  scanf("%d%d", &x, &y) ;
  printf("valeur1 = %d et valeur2 = %d ", x, y) ;
}
```


De même que pour la fonction `printf`, la fonction `scanf` possède en argument un format exprimé sous forme d'une chaîne de caractères, ici `"%d%d"`, ce qui correspond à deux valeurs entières. Nous trouvons par la suite une liste d'arguments correspondant à des adresses, précisant dans quelles variables on souhaite placer ces valeurs, ici `x` et `y`.

L'opérateur `&` (le *et* commercial) signifie *adresse de*. Sans cet opérateur, le compilateur ne détectera aucune erreur, mais les données lues au clavier seront rangées dans des emplacements aléatoires.

On note aussi que, pour la fonction `scanf`, les données tapées au clavier ne seront validées qu'après avoir tapé la touche *Entrée*.

Il est aussi possible de séparer la saisie de deux nombres par un espace, et valider le tout par la touche *Entrée* à la fin. Maintenant, si on sépare les deux formats dans `scanf` par un caractère bien précis, ceci nous oblige à le saisir pour séparer les deux nombres, comme c'est le cas dans `scanf("%d*%d", &x, &y)`, où la saisie des deux valeurs doit être séparée par le caractère `'*'`.

Remarques et définitions :

1. Dans un algorithme, on parle d'opérations. Dans un programme, on dit instructions. Un programme est donc une suite d'instructions.
2. Il faut aussi distinguer entre les opérations arithmétiques et logiques effectuées en utilisant les opérateurs (`+`, `-`, `/`, `and`, `or`, etc.), et les opérations d'un algorithme (affectation, lecture, écriture, etc.).

3. Langage machine ou code machine ou code binaire : instructions codées en binaire et directement exécutées au niveau du CPU (Central Processing Unit, ou le micro-processeur) sans traduction.
4. Compilateur : c'est un logiciel qui transforme un programme écrit en un langage de programmation (code source) en un programme dont les instructions sont écrites en langage machine (code binaire ou fichier exécutable). Lors de la compilation, des erreurs peuvent surgir ; il faut les corriger avant d'exécuter le programme.
5. On distingue en particulier deux types de problèmes (*erreurs de compilation*) qui peuvent être signalés par le compilateur : [Error] et [Warning]. [Error] signale un problème grave qui doit être corrigé pour que le programme s'exécute. [Warning] signale un cas pouvant entraîner un comportement inattendu, mais n'empêche pas l'exécution du programme.
6. Une *erreur d'exécution* se produit quand le programme n'a pas le comportement attendu lors de son exécution. Elle n'est pas détectée par le compilateur, mais par le programmeur lui-même. Par exemple, l'absence du `&` dans `scanf`, pour lire un entier, provoque une erreur d'exécution.
7. Le langage le plus facile à convertir en code machine est l'assembleur. L'assembleur est un langage de programmation dont les instructions correspondent au langage machine (c'est-à-dire aux instructions élémentaires du CPU), mais sont écrites en abréviations textuelles plutôt qu'en code binaire.

3. Exercices corrigés

3.1. Exercices

Exercice 1 :

1. Calculez l'expression arithmétique suivante :
   ```
   5 + 2 / (3 - 1 / 1) * 6 - 9
   ```
2. Calculez l'expression logique suivante :
   ```
   ((0 < 1) OU FAUX) ET (VRAI OU (9-3 = 3))
   ```

Exercice 2 :

Question 1 : Est-ce qu'il est possible de graver le contenu d'un disque dur de 30 GO dans 44 CDROM ? Justifiez votre réponse.

Question 2 : Combien de CDROM faut-il pour copier le contenu d'une clé USB de 4 GO ?

Question 3 : Quel est le nombre de caractères qui peuvent se trouver dans un disque dur de 4 GO ?

Question 4 : Quel est le nombre de cylindres qui existent dans un disque dur de 20 GO, en supposant que le disque dur contient 10 disques superposés à double-face, et il existe 32 secteurs par piste ?

3.2. Corrigés

Solution 1 :

1. Calcul de l'expression arithmétique :

```
5 + 2 / (3 - 1 / 1) * 6 - 9 = 5 + 2 / (3 - 1) * 6 - 9
                            = 5 + 2 / 2 * 6 - 9
                            = 5 + 1 * 6 - 9
                            = 5 + 6 - 9
                            = 11 - 9
                            = 2
```

2. Calcul de l'expression logique :

```
((0 < 1) OU FAUX ) ET (VRAI OU (9 - 3 = 3))
= (VRAI OU FAUX ) ET (VRAI OU (9 - 3 = 3))
= VRAI ET (VRAI OU (9 - 3 = 3))
= VRAI ET (VRAI OU (6 = 3))
= VRAI ET (VRAI OU FAUX)
= VRAI ET VRAI
= VRAI
```

Solution 2 :

Réponse 1 :

Taille du disque dur en MO :

1 GO → 1024 MO
30 GO → X
X = (1024 * 30)/1 = 30720 MO

Taille des CDROM en MO :

1 CDROM → 700 MO
44 CDROM → X
X = (700 * 44)/1 = 30800 MO

La taille des CDROM est supérieure à la taille du disque dur (30800 > 30720), alors la réponse est : **OUI**, il est possible de graver le contenu d'un disque dur de 30 GO dans 44 CDROM.

Réponse 2 :

Taille de la clé USB en MO

1 GO → 1024 MO
4 GO → X
X = (1024 * 4) / 1 = 4096 MO

Nbr de CDROM pour copier le contenu de la clé USB

1 CDROM → 700 MO
X CDROM → 4096 MO
X = (1*4096) / 700 = 5.85 ≈ **6 CDROM**

On a besoin de **6 CDROM** pour copier le contenu d'une clé USB de 4 GO.

Réponse 3 :

On a : 1 caractère → 1 octet
1 GO → 1024 MO = $(1024)^2$ KO = $(1024)^3$ O
=> 1 GO → $(1024)^3$ caractères
4 GO → X

$X = \frac{4*1024^3}{1}$ = **4294967296 caractères**

Théoriquement, il est possible de stocker l'équivalent de **4294967296 caractères** dans un disque dur de 4 GO.

Réponse 4 :

Taille du cylindre en O

On a : 1 piste → 32 secteurs
On a : 1 secteur → 512 O
32 secteurs → X
X = (512*32)/1 = 16384 O
=> 1 piste → 16384 O
On a aussi : 1 cylindre → 10 pistes * 2 = 10 * 16384 * 2 = 327680 O

Taille du disque dur en O

1 GO → 1024 MO = $(1024)^2$ KO = $(1024)^3$ O
20 GO → X

$X = \frac{20*1024^3}{1}$ = 21474836480 O

Nbr de cylindres/disque dur

On a : 1 cylindre → 327680 O
X → 21474836480 O

$X = \frac{21474836480*1}{327680}$ = **65536 cylindres**

Le disque dur contient **65536 cylindres**.

Chapitre 2 : Les algorithmes séquentiels simples

1. Parties d'un algorithme

Un algorithme contient deux parties:

- La partie données (déclaration) : contient les variables et les constantes.
- La partie traitement (code, opérations ou corps de l'algorithme) : elle correspond au processus de calcul.

2. Les données

Les données sont les objets manipulés dans un algorithme.
Dans un algorithme, toute donnée utilisée doit être déclarée. Les données peuvent être des variables ou des constantes.

Les variables : Une variable correspond à un objet dont la valeur peut varier au cours de déroulement de l'algorithme. Une variable est caractérisée par :

- Un nom (identificateur) qui doit être explicite, i.e. indique le rôle de la variable dans l'algorithme.
- Un type : indique les valeurs qui peuvent être prises par la variable.
- Une valeur : indique la grandeur prise par la variable à un moment donné.

Sur le plan technique, une variable correspond à une case mémoire (emplacement de mémoire) avec : le non de la variable est l'adresse de la case mémoire, le type indique la taille (le nombre d'octets) de la case, et la valeur représente le contenu de la case. C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire.

Les constantes : Une constante est un cas particulier de la variable. Il s'agit d'une variable dont la valeur est inchangeable dans l'algorithme tout entier.

Exemple :

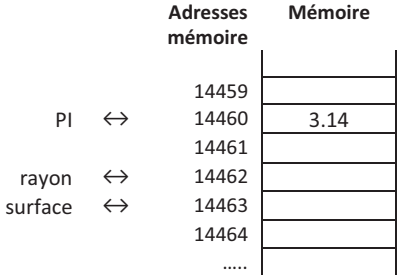
```
Algorithme calcul_surface
  Constantes
    PI=3.14 ;
  Variables
    Réel    rayon, surface ;
Début
…
```

En C :

```
#define    PI    3.14
main()
{
float    rayon, surface ;
…
```

Une constante est donc introduite en C par la directive de pré-compilation `#define` suivie du nom de la constante, et enfin sa valeur. Lors de la pré-

compilation, chaque fois où la constante figure dans le code source, elle sera remplacée par sa valeur.
Les variables et la constante du programme précédent peuvent être représentées en mémoire comme suit :



En fait, cette représentation n'est pas tout à fait exacte, car réellement la mémoire est constituée d'un ensemble d'octets, chacun est identifié par une adresse. La case mémoire correspondant à une variable est constituée d'un ensemble d'octets selon son type. L'adresse d'une variable est l'adresse du premier octet de l'espace mémoire réservé (alloué) à la variable.
On note aussi qu'il n'y a pas de place allouée en mémoire pour une constante introduite par `#define`. Le nom de la constante (`PI`) sera remplacé par `3.14` tout au long du programme. Pour déclarer des constantes typées, on utilise le mot clé `const`. Dans ce cas-là, un espace mémoire sera réservé à la constante selon son type. Par exemple : `const float PI=3.14 ;`

3. Les types

Le type correspond au genre ou la nature de la variable que l'on souhaite utiliser. Il indique donc les valeurs qui peuvent être prises par cette variable.
La déclaration d'une variable consiste à l'associer à un type. Chaque type donné peut être manipulé par un ensemble d'opérations.
Il existe des types simples et des types structurés.

Les types simples (scalaires) : Un type simple est un type de données décrivant une information atomique, i.e. constituée d'une seule valeur. A leur tour, les types simples peuvent être classés en deux catégories :

1. Types numériques :
   - `Entier` : Par exemple : `12`, `-55`. Les opérateurs de manipulation des entiers sont :
     - Les opérateurs arithmétiques classiques : `+`, `-`, `*`, `/`.
     - Le modulo `%`, avec `n%p` donne le reste de la division de `n` par `p`.
     - Les opérateurs de comparaison classiques : `<`, `>`, `=`, etc.
   - `Réel` : Par exemple : `12.5`, `-2.09`. Les opérateurs de manipulation des réels sont :

- Les opérateurs arithmétiques classiques : +, –, *, /.
- Les opérateurs de comparaison classiques : <, >, =, etc.

2. Types symboliques :
   - `Booléen` : comporte les deux valeurs `VRAI` et `FAUX`. Les opérateurs de manipulation des booléens sont : `NON`, `ET`, `OU`.
   - `Caractère` : comporte les données alphanumériques, symboliques, signes de ponctuation, etc., contenant un seul caractère, par exemple `'a'`, `'?'`, `'3'`, `';'`. Notons qu'un caractère se met entre deux guillemets simples. Les opérateurs de manipulation des caractères sont les opérateurs de comparaison : >, <, =, etc. Le résultat de la comparaison dépend du code ASCII des caractères comparés.

Les types structurés (complexes) : Un type structuré est un type de données décrivant une information composite, i.e. constituée de plusieurs valeurs, elles-mêmes de types simples ou structurés. Parmi ces types on cite : le type tableau, le type chaîne de caractères et le type structure qui seront vus ultérieurement.

4. Opérations de base

La partie traitement d'un algorithme implique un ensemble d'opérations qui peuvent être :

- Opérations de base ou encore élémentaires qui permettent une exécution séquentielle d'un algorithme.
- Structures de contrôle qui permettent le saut dans un algorithme.

Les opérations de base sont l'affectation et les opérations d'entrée/sortie.

4.1. L'affectation

L'opération la plus importante en algorithmique est l'affectation (assignation) qui se note ← (= en C), et qui consiste à attribuer ou affecter à une variable une valeur appartenant à son domaine de définition (son type). La valeur affectée est souvent le résultat d'un calcul d'une expression arithmétique ou une expression logique.

Exemple :

```
Algorithme calculs
…
Début
  X ← 4 ;
  Y ← X * 2 ;
  Z ← Y ;
  Z ← Z – 6 ;
  H ← (10>5) ET (2<3) ;
…
```

En C :

```
#include <stdio.h>
main()
{ …
  X = 4 ;
  Y = X * 2 ;
  Z = Y ;
  Z = Z – 6 ;
  H = (10>5) && (2<3) ;
…
```

L'exemple est lu comme suit : La variable `X` reçoit la valeur `4` ; La variable `Y` reçoit la valeur de `X` multipliée par `2` ; La variable `Z` reçoit la valeur de `Y` ; La variable `Z` reçoit la valeur courante (actuelle) de `Z` moins `6` ; La variable `H` reçoit la valeur `VRAI`.

Remarque : Comme en C il n'y a pas de type booléen, `H` va prendre dans ce programme la valeur 1.

4.2. Les entrées/sorties

Les échanges d'information entre l'utilisateur et la machine sont appelés opérations d'entrée-sortie. Les opérations d'entrée-sortie sont :

- `Lire()` : qui récupère la valeur tapée au clavier et l'affecte à l'espace mémoire désigné par la variable entre parenthèses. En C, c'est `scanf()`.
- `Ecrire()` : qui récupère la valeur située à l'espace mémoire désigné par la variable entre parenthèses, et affiche cette valeur à l'écran. En C, c'est `printf()`.

Une chaîne de caractères : est une suite de plusieurs caractères, permettant de représenter des mots ou des phrases. Une chaîne de caractères doit être mise entre deux guillemets doubles pour la distinguer d'un identificateur de variable. Par exemple, `Ecrire("bonjour")`, en C `printf("bonjour")`, permet d'afficher le mot `bonjour` à l'écran.

5. Construction d'un algorithme simple

La construction d'un algorithme consiste à lui donner un nom, identifier les variables et les constantes et écrire le corps de l'algorithme.

Le corps de l'algorithme est constitué d'une séquence d'opérations mises entre `Début` et `Fin` et séparées par des points-virgules (`;`). Le corps de l'algorithme peut contenir des opérations de lecture, écriture, affection, etc. Pour éclaircir l'algorithme, son corps peut contenir des commentaires mis entre `/*…*/`.

Exemple :

L'algorithme de calcul de la surface d'un cercle représenté par énumération des étapes est le suivant :

1. Saisir le rayon du cercle (lire le rayon).
2. Calculer la surface par l'expression : $\Pi*(\text{rayon})^2$.
3. Afficher le résultat (écrire la surface).

En utilisant un langage algorithmique, on obtient :

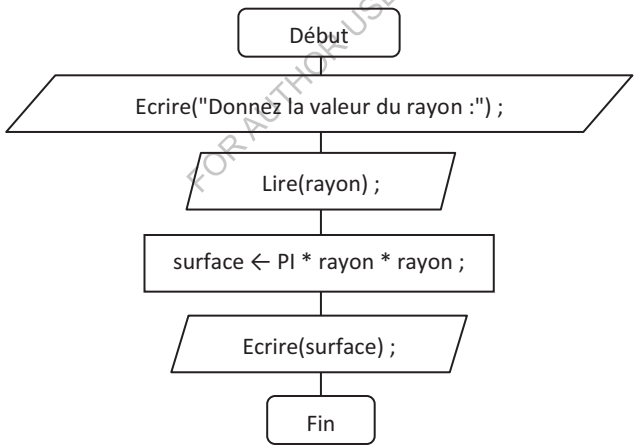
```
Algorithme Calcul_Surface
/* Calcul de la surface d'un cercle */
Constantes
   PI = 3.14159 ;
Variables
  Réel  rayon, surface ;
Début
  Ecrire("Donnez la valeur du rayon :") ;
  Lire(rayon) ;
  surface ← PI * rayon * rayon;
  Ecrire(surface) ;
Fin
```

6. Représentation d'un algorithme par un organigramme

Un algorithme peut être représenté aussi par un organigramme facilitant sa compréhension. Un organigramme est une représentation graphique (diagramme) de la solution d'un problème. Pour cela, on utilise les symboles géométriques suivants :

Début/Fin	Un rectangle aux coins arrondis représente `Début` ou `Fin` de l'algorithme.
Lire/Ecrire	Un parallélogramme représente une opération d'entrée/sortie.
X ← Y + 1	Un rectangle représente un traitement (une affectation par exemple).
Oui Condition Non	Un losange représente un test d'une condition pour une décision ou une sélection.
	Un lien orienté ou non représente une liaison entre deux nœuds et indique aussi l'ordonnancement des opérations.

L'algorithme de calcul de la surface peut être représenté par l'organigramme suivant :



7. Traduction en langage C

7.1. Exemple

La traduction de l'algorithme de calcul de la surface en un programme C sera comme suit :

```
#include <stdio.h>
#define    pi    3.14159
main ()
/* Calcul de la surface d'un cercle */
```

```
{
  float rayon, surface ;
  printf("Donnez la valeur du rayon :\n") ;
  scanf("%f", &rayon) ;
  surface = pi * rayon * rayon ;
  printf("%f",surface) ;
}
```

7.2. Manipulation des nombres

Les entiers :

Le langage C comprend principalement trois types d'entiers, différenciés par la quantité d'espace mémoire utilisée pour représenter la valeur numérique considérée :

- `short` : 2 octets
- `int` : en général 2 octets, parfois 4 ou plus (ça dépend du compilateur)
- `long` : 4 octets

De plus, deux modificateurs peuvent être appliqués à ces types :

- `unsigned` : entiers naturels (positifs). Tous les bits sont utilisés pour représenter le nombre.
- `signed` : entiers relatifs (positifs ou négatifs). Le bit de poids fort (à gauche) représente le signe du nombre (0 représente un signe +, et 1 représente un signe -). Le reste des bits représente le nombre.

Si on ne précise pas le modificateur, le type est `signed` par défaut.

Voici un tableau explicatif :

Type	Octets	Nombre de valeurs possibles	Modificateur	Intervalle de définition
`short`	2	$2^{16} = 2^{2\times8} = 65536$	`unsigned` `signed`	$[0, 2^{16} - 1]$ $[-2^{15}, 2^{15} - 1]$
`long`	4	$2^{4\times8} = 4294967296$	`unsigned` `signed`	$[0, 2^{32} - 1]$ $[-2^{31}, 2^{31} - 1]$

Les réels :

Les nombres réels sont des nombres à virgule flottante, pouvant s'écrire sous la forme : $\pm Mantisse\ E \pm Exposant$, avec $Nombre\ réel = \pm Mantisse * 10^{\pm Exposant}$

En C, un nombre réel exprimé en notation scientifique à virgule flottante (avec le format `%e`) prend la forme (`x.xxxxxxe±xxx`)

Exemple : Le nombre réel (`float` en C) `55.007`, s'écrit avec le format `%e` comme suit : `5.500700e+001`

En langage C, on trouve trois types de réels :

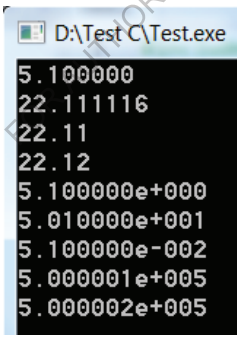
Type	Octets	Signe	Mantisse	Exposant	Intervalle de définition
`float`	4	1 bit	23 bits	8 bits	$[1{,}5 \times 10^{-45} ; 3{,}4 \times 10^{38}]$
`double`	8	1 bit	52 bits	11 bits	$[5 \times 10^{-324} ; 1{,}7 \times 10^{308}]$
`long double`	10	1 bit	64 bits	15 bits	$[3{,}4 \times 10^{-4932} ; 1{,}1 \times 10^{4932}]$

Remarque : Les informations de cette section peuvent nous aider à calculer la complexité spatiale de quelques programmes. La complexité spatiale est le coût en espace d'un programme, i.e., indique combien d'espace mémoire ce programme occupe. Nous avons aussi la complexité temporelle qui correspond au coût en temps d'un programme, i.e., indique combien d'instructions ce programme comporte.

Exemple : Soit le programme C suivant :

```
#include <stdio.h>
main()
{
  float x;
  short y, z;
  x = 5.1; printf("%f\n", x);
  x = 22.1111169; printf("%f\n", x);
  x = 22.115; printf("%.2f\n", x);
  x = 22.116; printf("%.2f\n", x);
  x = 5.1; printf("%e\n", x);
  x = 50.1; printf("%e\n", x);
  x = 0.051; printf("%e\n", x);
  x = 500000.14; printf("%e\n", x);
  x = 500000.15; printf("%e\n", x);
}
```

Le programme affiche :



Notons que le format `%.2f` nous a permis d'afficher un nombre avec deux chiffres après la virgule avec ajustement. Si on l'affiche en notation scientifique à virgule flottante, il y aura aussi un ajustement.

La complexité spatiale du programme = 4 + 2 + 2 = 8 octets.

La complexité temporelle du programme = 18 instructions.

7.3. Manipulation des caractères

Un caractère (lettre, ponctuation, symbole, etc.) est déclaré en langage C par le mot clé `char`. Une variable de type caractère occupe 1 octet en mémoire. Chaque caractère possède un code ASCII. La table ASCII est un tableau de 256

caractères numérotés de 0 à 255. A titre d'exemple, les lettres majuscules de `'A'` à `'Z'` sont codées dans l'ordre par les codes 65 à 90.

Quelques fonctions de manipulation des caractères :

En langage C, il existe une fonction nommée `putchar` permettant d'afficher un seul caractère.

Exemple :

```
#include <stdio.h>
main ()
{
  char x ='e';
  putchar(x);
}
```

Le programme affiche : `e`

Maintenant, la fonction `getchar` permet de lire un caractère :

Exemple :

```
#include <stdio.h>
main ()
{
  char x = getchar();
  printf("%c", x);
}
```


Ce programme permet d'affecter un caractère lu à partir du clavier à la variable `x` grâce à la fonction `getchar`, et de l'afficher ensuite à l'écran.

Remarques :

- `putchar` et `getchar` ne peuvent être exécutées que si on ajoute la directive de pré-compilation `#include` permettant d'incorporer au programme, avant la compilation, le fichier `stdio.h` là où les deux fonctions sont définies.
- Pour la fonction `getchar`, le caractère tapé au clavier ne sera validé qu'après avoir tapé la touche *Entrée*.
- Chaque caractère doit être mis entre deux guillemets simples pour le distinguer d'un identificateur.
- Il est possible de comparer les caractères selon l'ordre du code ASCII. Ainsi, `('A'>'B')` retourne `0` (`FAUX` en algorithmique).
- Puisque chaque caractère correspond à un code ASCII, une variable de type `char` peut être manipulée comme un nombre. Ainsi, l'instruction `char c = 'A'` est équivalente à `char c = 65`. De même, l'instruction `int i = 'A'` est équivalente à `int i = 65`.
- Il ne faut pas confondre le caractère `'6'` ayant le code ASCII 54, et le nombre `6` représentant lui-même le code ASCII d'un autre caractère.

7.4. Manipulation des booléens

Il n'existe pas de type booléen à proprement parler en C. A la place, on utilise des entiers :

- La valeur *zéro* représente la valeur `FAUX`.
- Une valeur *non-nulle* représente la valeur `VRAI`.

Remarque :

Dans d'autres langages, tels que Pascal, le type booléen existe explicitement. Par exemple, soit le programme Pascal suivant :

```
program exemple ;
  var b : boolean ;
begin
  b := ('A'>'B') ;
  writeln(b) ;
end.
```

Ce programme affiche : `FALSE`

Son équivalent en C est :

```
#include <stdio.h>
main()
{
  int b ;
  b = ('A'>'B') ;
  printf("%d", b) ;
}
```

Ce programme affiche : `0`

7.5. Opérateurs en C

Opérateurs arithmétiques :

Le tableau suivant montre des opérateurs arithmétiques en C :

Opérateur	Description	Type	Priorité	Associativité
+	identité	unaire	14	DàG
−	opposé	unaire	14	DàG
+	addition	binaire	12	GàD
−	soustraction	binaire	12	GàD
*	multiplication	binaire	13	GàD
/	Division (si les deux opérandes sont des entiers, il s'agit d'une division entière)	binaire	13	GàD
%	modulo : reste de la division entière (les deux opérandes doivent être des entiers)	binaire	13	GàD

Les deux premiers opérateurs (identité et opposé) sont dits unaires, portant sur un seul opérande. Les autres opérateurs sont dits binaires, portant sur deux opérandes.

Les opérateurs en C sont classés selon 16 niveaux de priorité : le plus haut niveau étant 15 ; le plus bas niveau est donc 0.

Le sens d'associativité indique si les expressions contenant plusieurs opérateurs de même priorité doivent être évaluées en considérant les opérandes de gauche à droite (GàD) ou de droite à gauche (DàG).

Les parenthèses peuvent être utilisées pour améliorer la lisibilité de l'expression, ou pour changer l'ordre de priorité.

Ces opérateurs s'appliquent à des valeurs numériques (y compris les caractères qui sont représentés par leurs codes ASCII) et renvoient un résultat lui aussi numérique.

Exemples :

- `7*4/3` est équivalente à `(7*4)/3`, car les opérateurs `*` et `/` ont la même priorité, et l'ordre d'évaluation de l'expression est donc déterminé par leur sens d'associativité (GàD).
- `7/4*3` est équivalente à `(7/4)*3`, pour les mêmes raisons.

Opérateurs de comparaison (relationnels) :

Les opérateurs de comparaison prennent comme opérandes des expressions numériques, et produisent un résultat booléen, i.e. `VRAI` ou `FAUX`. Comme il n'y a pas de type booléen en C, alors, si le résultat de la comparaison est :

- `FAUX`, alors l'expression retourne `0`.
- `VRAI`, alors l'expression retourne `1`.

Opérateur	Description	Priorité	Associativité
==	égal	9	GàD
!=	différent	9	GàD
<	strictement inférieur	10	GàD
<=	inférieur ou égal	10	GàD
>	strictement supérieur	10	GàD
>=	supérieur ou égal	10	GàD

Exemples :

- `(0>5)` est une expression de type `int` dont la valeur est `0` (`FAUX`).
- `(1.0>0.5)` est une expression de type `int` dont la valeur est `1` (`VRAI`).

Attention ! Il ne faut pas confondre l'opérateur d'affectation (=) et celui d'égalité (==), car le symbole utilisé en mathématiques pour l'égalité est utilisé en C pour l'affectation. Cette erreur (eg. mettre = au lieu de ==) est difficile à détecter, car elle ne donne pas lieu à une erreur de compilation ; elle provoque plutôt une erreur d'exécution.

Opérateurs logiques :

Les opérateurs logiques sont supposés s'appliquer à des valeurs booléennes (`VRAI` et `FAUX`) et renvoyer un résultat lui aussi booléen.

Puisqu'il n'y a pas de type booléen en C, et qu'on représente `VRAI` et `FAUX` par des entiers, alors ces opérateurs :

- s'appliquent à des valeurs entières :

- ☞ `0` représente `FAUX`,
- ☞ une valeur non-nulle représente `VRAI`.

- et renvoient une valeur entière :
  - ☞ `0` pour un résultat = `FAUX`,
  - ☞ `1` pour un résultat = `VRAI`.

Opérateur	Description	Priorité	Associativité
!	non (ou négation)	14	DàG
&&	et (conjonction)	5	GàD
||	ou (disjonction)	4	GàD

Exemples :

- `(0>1) && (10>20)` est une expression de type `int` dont la valeur est `0` (`FAUX`).
- `(0>4) || (5>4)` est une expression de type `int` dont la valeur est `1` (`VRAI`).
- `(0>4) || 19` retourne `1` (`VRAI`). La valeur non-nulle `19` correspond à `VRAI`.
- `(0>4) || 19 && 0 || (0<4)` retourne encore `1` (`VRAI`). Cette expression va correspondre à `VRAI || VRAI && FAUX || FAUX`, qui va être calculée de la manière suivante : `VRAI || VRAI && FAUX || FAUX = VRAI || FAUX || FAUX = VRAI || FAUX = VRAI`.

Remarques :

- L'opérateur `ET` est appliqué tant qu'on ne rencontre pas d'opérande `FAUX` (auquel cas la valeur de l'expression est `FAUX`). Par exemple, pour `1 && 1 && 0 && 1 && 1`, on évalue seulement la partie rouge de l'expression, car dès qu'on rencontre le `0`, on sait que l'expression est fausse sans avoir besoin d'en connaitre la suite.
- L'opérateur `OU` est appliqué tant qu'on ne rencontre pas d'opérande `VRAI` (auquel cas la valeur de l'expression est `VRAI`). Par exemple, pour `0 || 1 || 0 || 0`, là-aussi, on évalue seulement la partie rouge de l'expression, car dès qu'on rencontre le `1`, on sait que l'expression est vraie sans avoir besoin d'en connaitre la suite.

Opérateurs bit-à-bit

Sachant que toutes les données (entier, réel, caractère, etc.) sont stockées dans la mémoire d'un ordinateur en binaire, ces opérateurs permettent de travailler directement sur la représentation binaire des données, bit par bit.

Opérateur	Description	Priorité	Associativité
~	complément à un	14	DàG
&	et bit-à-bit	8	GàD
^	ou exclusif bit-à-bit	7	GàD
|	ou bit-à-bit	6	GàD

>>	décalage à droite	11	GàD
<<	décalage à gauche	11	GàD

Le complément à un d'un nombre binaire est la transformation obtenue en remplaçant tous ses `0` par des `1`, et tous ses `1` par des `0`.

et bit-à-bit : 0&0=0, 0&1=0, 1&0=0, 1&1=1.

ou exclusif (XOR) bit-à-bit : 0^0=0, 0^1=1, 1^0=1, 1^1=0.

ou bit-à-bit : 0|0=0, 0|1=1, 1|0=1, 1|1=1.

Le décalage à droite se fait en supprimant le bit de poids faible (à droite), et en ajoutant un 0 à gauche.

Le décalage à gauche se fait en supprimant le bit de poids fort (à gauche), et en ajoutant un 0 à droite.

7.6. Autres possibilités et règles en C

Dans ce qui suit, nous citons quelques possibilités offertes par le langage C, et quelques règles qui doivent être respectées lors de l'écriture d'un programme C :

L'opérateur sizeof :

L'opérateur `sizeof(t)` permet de connaitre l'espace mémoire occupé par une valeur de type `t` spécifié. Ainsi, `sizeof(float)` retourne `4`. Il peut aussi être appliqué directement à une expression, pour obtenir l'espace qu'elle occupe en mémoire.

Dépassement de capacité :

Si jamais, lors d'un calcul, la valeur manipulée sort de l'intervalle prévu, on dit qu'il y a un dépassement de capacité.

```
unsigned short x = 65536 ;
```

`65536` est représenté en binaire `10000000000000000`, ce qui va dépasser les 2 octets réservés à un `unsigned short`. La valeur affectée à `x` sera `0`. Notons que cette erreur n'empêche pas l'exécution du programme, mais pourra donner des résultats erronés.

Identificateurs et mots-clés :

Un identificateur est un nom donné à un élément de programmation : fonction, variable, constante, etc. Il doit être significatif, i.e. indique le rôle de l'élément dans le programme.

Un identificateur doit être unique. Il doit être constitué de lettres (minuscules et/ou majuscules), des chiffres et/ou le caractère tiret-bas (aussi appelé underscore : _). Il ne doit pas contenir des signes diacritiques sur les lettres (accents, trémas, tilde, etc.). Il ne doit pas dépasser 127 caractères. Le premier caractère doit être une lettre.

Remarque : Le compilateur C fait la distinction entre les lettres minuscules et les lettres majuscules. Par exemple, les deux identificateurs `val` et `VAL` sont considérés différents.

Les mots clés (ou mots réservés) du langage C, tels que `main`, `printf`, `scanf`, `const`, etc. ne doivent pas être utilisés comme identificateurs.

Commentaires et lisibilité :

Un commentaire est un texte exprimé en langage naturel, et dont le but est d'expliquer le fonctionnement du programme à un programmeur. Ce texte est ignoré par le compilateur.

En C, un commentaire est délimité par `/*` et `*/`, et peut occuper plusieurs lignes. On peut aussi faire des commentaires d'une seule ligne en insérant `//` en début de la ligne.

Exemple :

```
/* Ce programme calcule la surface
d'un rectangle*/

// Voici la première méthode
```

Les commentaires améliorent la lisibilité du code.

La lisibilité du code source est la propriété d'un code source à être facilement compris par une personne qui ne l'a pas forcément écrit.

Un autre moyen pour améliorer la lisibilité du code est l'indentation.

L'indentation est un décalage introduit en début de ligne pour aligner horizontalement certaines lignes du code source.

Exemple : Aligner les accolades, aligner les instructions appartenant au même bloc…

Constantes symboliques vs. Constantes littérale :

Une constante définie par le mot clé `const`, ou par la directive `#define` est dite symbolique. Elle permet d'associer un identificateur à une valeur qui ne change pas au cours d'exécution du programme.

Une constante littérale ne possède pas de nom, il s'agit simplement d'une valeur donnée explicitement dans le code source. On distingue :

- Les constates numériques :
  - Constantes entières, par exemple : `19`
  - Les constantes réelles sont exprimées en utilisant un point décimal, par exemple : `19.0` ou `19.`
- et les constantes textuelles :
  - Constantes caractères, par exemple `'A'`
  - Constantes chaîne de caractères, par exemple `"bonjour"`

Conversions :

Dans une instruction d'affectation, le type de l'expression à droite doit correspondre au type de la variable de destination à gauche.

La partie gauche d'une affectation est souvent dite *lvalue*, pour *left value* en anglais.

Pour une souplesse de programmation, le langage C offre des possibilités de conversion.

Grâce à cette conversion, l'expression (à droite) va être calculée, même si les types de ses opérandes sont différents ; et l'affectation sera effectuée même si le type du résultat (à droite) est différent au type de la *lvalue*.

La conversion est l'action de changer le type de données d'une expression. Elle peut être implicite ou explicite.

La conversion implicite est effectuée automatiquement sans aucune intervention du programmeur. Le compilateur la déduit du code source. Cela se fait comme suit :

- ☞ Le type du résultat d'une expression étant imposé par celui des opérandes ayant le type le plus fort, sachant que l'ordre de force décroissante s'établit ainsi : `long double`, `double`, `float`, `unsigned long`, `long`, `unsigned int`, `int`.
  Les opérandes de type (`unsigned`) `char` ou (`unsigned`) `short` sont systématiquement convertis en (`unsigned`) `int`.
- ☞ Le résultat obtenu après calcul va être converti au type de la *lvalue* de l'affectation.

Exemple 1 :

```
#include <stdio.h>
main()
{
  float a=2.1, b;
  int x, y=7;
  x = a;
  printf("x = %d\n",x);
  b = a / y;
  printf("b = %f",b);
}
```

Dans la première affectation (`x = a`), la valeur de `a` sera convertie en entier pour pouvoir l'affecter à `x`. `x` aura la valeur `2`. Dans ce cas-là, il y a eu une perte de précision ou dégradation de l'information (la partie décimale du nombre réel a été perdue). Dans la deuxième affectation (`b = a / y`), la valeur de `y` sera convertie en réel (car c'est le type de `a`), et le résultat réel sera affecté à `b`.

Ce programme affiche donc :

```
x = 2
b = 0.300000
```

Passant maintenant à ***la conversion explicite*** qui est une conversion demandée par le programmeur grâce à l'opérateur de transtypage, appelé aussi *cast*.

Format général : `(type) expression`

Exemple 2 :

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

```
main()
{
  int x=10, resultat;
  float y=3.0;
  resultat = x % (int)y;
  printf("modulo = %d", resultat);
}
```

Dans cet exemple, nous avons converti un réel en entier pour pouvoir appliquer l'opérateur modulo, et le programme affiche ainsi : `modulo = 1`

Exemple 3 :

```
#include <stdio.h>
main()
{
  int x=10,y=3;
  float resultat;
  resultat = x/(float)y;
  printf("resultat = %f", resultat);
}
```

La conversion explicite nous a permis d'obtenir un résultat réel `3.333333`, au lieu de `3.000000` si on enlève la conversion.

Exemple 4 :

```
#include <stdio.h>
main()
{
  float surface = 100 / 3;
  printf("%f\n",surface) ;
  surface = 100. / 3;
  printf("%f",surface) ;
}
```


`100` et `3` sont deux constantes entières ; c'est pourquoi le premier opérateur `/` a permis d'effectuer une division entière. Le résultat étant converti en réel et affecté à surface.

`100.` est une constante réelle ; c'est pourquoi, le deuxième opérateur `/` a permis d'effectuer une division entre réels après avoir converti la constante `3` en réel.

Le programme de l'exemple 4 affiche donc :

```
33.000000
33.333332
```

Attention ! Le calcul de la valeur (résultat) d'une expression se fait selon l'ordre de priorité entre opérateurs. Dans une expression, chaque fois qu'on calcule le résultat d'une opération entre deux opérandes, on applique les règles de conversion déjà vues. C'est pourquoi, l'expression `1 / 2 * 9.2` donne un résultat peut être inattendu (`=0.000000`). Puisqu'il y a une égalité de priorité entre `/` et `*`, la division entière s'effectuera avant la multiplication, ce qui donne *`1/2 =0`*. A la différence de *`9.2 * 1 / 2`* qui donnera *`4.600000`* ; ce

même résultat correct peut être obtenu avec `1./2* 9.2` ou `1/2.*9.2` ou `1/(float)2*9.2`, etc.
On remarque aussi que la conversion des réels en nombres entiers supprime la partie décimale (troncature).

L'opérateur =

En C, l'affectation est un opérateur. Il possède une associativité de droite à gauche (DàG).

`k = i = 5` est équivalente à `k = (i = 5)`. On affecte le `5` à `i`, puis la valeur de l'expression `i=5`, qui est `5`, sera affectée à `k`.

`x = y = z = 7` est équivalente à `x=(y=(z=7))`. Donc, `7` sera affecté d'abord à `z`, puis la valeur de l'expression `z=7` (qui est aussi `7`) sera affectée à `y`, et ainsi de suite.

Il ne faut pas confondre l'opérateur d'affectation `=` avec l'opérateur d'égalité `==`. `x==y==1` est équivalente à `(x==y)==1`, car le sens d'associativité de `==` est de GàD. On teste donc d'abord si `x` et `y` sont égaux, puis on compare l'entier résultant de ce test avec la valeur `1`.

Fonctions mathématiques :

La bibliothèque `math.h` contient les fonctions de calcul scientifique : `pow` (puissance), `sqrt` (racine carrée), `cos`, `sin`, `tan`, etc.

Entrées et sorties formatées :

Les deux fonctions `scanf` et `printf` ont besoin d'un argument spécifiant le format. Il s'agit d'une chaîne de caractères qui indique éventuellement la position et le format d'affichage des autres arguments grâce au caractère `%`.
Le caractère `%` est une balise de positionnement suivie d'une spécification de traduction dont le format général est :

`%[indicateur][largeur][.precision][modification]type`.

Tous les champs notés entre crochets sont optionnels. Le type est obligatoire. Voici les différentes possibilités de type :

Type	Signification
`%c`	Caractère
`%s`	Chaîne de caractères
`%d`	Nombre entier en décimal
`%e`	Nombre réel sous la forme mantisse/exposant [-]m.nnnnnne[+|-]xx
`%E`	Nombre réel sous la forme mantisse/exposant en majuscule [-]m.nnnnnne[+|-]xx
`%f`	Nombre réel sous la forme [-]mmm.nnnnnn
`%g`	Nombre réel sous la forme la plus courte entre les types %e et %f
`%G`	Nombre réel sous la forme la plus courte entre les types %E et %f
`%o`	Nombre entier en octal
`%p`	Pointeur ou adresse de la valeur numérique
`%u`	Nombre entier non signé en décimal
`%x`	Nombre entier en hexadécimal

%X	Nombre entier en hexadécimal ; lettres affichées en majuscules

Exemple :

```
#include <stdio.h>
main()
{
  float x = 2.0 / 3.0 ;
  printf("%f", x);
}
```

Le programme affiche : `0.666667`

Pour afficher un réel en spécifiant le nombre de chiffres après la virgule avec ajustement, le format est `%.?f`.

Par exemple : `printf("%.3f", x)` affiche : `0.667`

`printf("%f",s)` avec `s` entier, ou `printf("%d",s)` avec `s` réel, dans les deux cas, il n'aura pas d'erreur de compilation, mais on aura un résultat d'affichage inattendu.

L'opérateur d'incrémentation :

Notation : `++`

Priorité : la même que pour l'opérateur unaire identité (+).

Dans un programme C, `i++` est équivalente à `i = i + 1`.

`i++` peut figurer dans une expression comme le montre l'exemple suivant :

```
#include <stdio.h>
main()
{
  int i = 5 ;
  int n = i++     + 2 ;
  int k = i ;
  printf("i=%d  n=%d  k=%d", i, n, k);
}
```


`i++` incrémente la valeur de `i` d'une unité. Cependant, cette incrémentation ne sera tenue en compte que dans l'instruction suivante.

Le programme affiche : `i=6  n=7  k=6`

Dans le cas où l'opérateur `++` précède la variable, alors l'incrémentation sera effectuée immédiatement dans l'opération en cours. Ainsi, dans l'exemple suivant :

```
#include <stdio.h>
main()
{
  int i = 5 ;
  int n = ++i     + 2 ;
  int k = i ;
  printf("i=%d  n=%d  k=%d", i, n, k);
}
```

Le programme affiche : `i=6  n=8  k=6`

Avec le même esprit, agit l'opérateur `--`.

Instruction expression :

En langage C, la notion d'instruction et d'expression sont étroitement liées. On parle souvent de *instruction expression*.

```
++i;
i = 5;
k = i = 5;
```

Les deux premières ont l'allure d'une instruction d'affectation. Dans les deux cas il y a l'évaluation d'une expression. La valeur de l'expression (`i = 5`), i.e. `5`, est affectée à `k`. La dernière affectation elle aussi agit comme une expression, et retourne une valeur (`5`) qui pourra être utilisée ultérieurement.

Opérateurs d'affectation élargie :

Le langage C permet de condenser les affectations de la forme :

```
lvalue = lvalue   Opérateur   expression
```

en

```
lvalue    Opérateur = expression
```

Par exemple : `i = i + 5` est équivalente `i += 5`.

Ceci est valable aussi pour `-`, `*`, `/`, `%`, ainsi que les opérateurs de manipulation des bits.

L'opérateur conditionnel :

Soit par exemple :

```
Si (a>b) max ← a ; sinon max ← b;
```

Ceci peut être écrit en C :

```
max = a>b ? a : b;
```

Déclaration d'une variable :

En langage C, il est possible de déclarer une variable n'importe où dans un programme. La règle à respecter est qu'on ne peut utiliser une variable qu'après sa déclaration, comme c'est le cas pour la variable `y` dans l'exemple suivant :

```
#include <stdio.h>
main()
{
  int x ;
  x = 2;
  int y; // Déclaration de y au milieu du programme
  y=x;
  printf("%d  --  %d", x, y) ;
}
```

Il faut aussi tenir compte de la portée d'une variable, i.e. le bloc d'instructions là où la variable peut être utilisée. Dans l'exemple suivant, la variable `y` ne peut être utilisée que dans le bloc d'instructions défini entre `{` et `}`, et non pas ailleurs.

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

```
{
  int x ;
  x = 2;
  { // Début du bloc dans lequel la variable y peut être utilisée
    int y;
    y=x;
    printf("%d  --  %d", x, y) ;
  }// Fin du bloc
}
```

8. Exercices corrigés

8.1. Exercices

Exercice 1 :

Qu'affiche l'algorithme suivant ?

```
Algorithme calcul_double
Variables
  entier val, dbl ;
Début
  val ← 231 ;
  dbl ← val * 2 ;
  Ecrire(val) ;
  Ecrire(dbl) ;
Fin
```


Exercice 2 :

Ecrire un algorithme permettant de lire deux nombres *a* et *b*, de calculer et d'afficher leur moyenne. Traduire l'algorithme en C. Ensuite, représentez l'algorithme par un organigramme. Déroulez l'algorithme dans le cas où l'utilisateur donne pour *a* la valeur 7, et pour *b* la valeur 19.

Note : Le déroulement d'un algorithme (simulation ou trace d'un algorithme) consiste à décrire le changement des variables (sur le plan technique, l'état de la mémoire) au cours d'exécution des opérations de l'algorithme, dans un tableau appelé tableau de situation.

Exercice 3 :

Ecrire un algorithme permettant de lire trois nombres, de calculer et d'afficher leur somme, leur produit et leur moyenne. Traduire l'algorithme en C.

Exercice 4 :

Ecrire un algorithme qui demande un nombre à l'utilisateur, puis calcule et affiche le carré, le double et le triple de ce nombre. Traduire l'algorithme en C.

Exercice 5 :

Ecrire un algorithme qui lit le prix *HT* (Hors Taxe) d'un article, le nombre d'articles et le taux de *TVA* (la Taxe sur la Valeur Ajoutée), et qui fournit le prix total *TTC* (Toutes Taxes Comprises) correspondant. Traduire l'algorithme en C.

Exercice 6 :
Ecrire un algorithme permettant de lire le rayon *R* d'une sphère, de calculer et d'afficher son *aire = 4 pi* R^2 et son *volume = 4/3 pi* R^3. Traduire l'algorithme en C.

Exercice 7 :
Ecrire un algorithme permettant de lire au clavier les longueurs des trois côtés *a*, *b* et *c* d'un triangle. L'algorithme doit calculer et afficher le périmètre et l'aire du triangle. *Périmètre = p = a + b + c*, et *Aire = (p/2 * (p/2-a) * (p/2-b) * (p/2-c))*$^{1/2}$. Traduire l'algorithme en C.

Exercice 8 :
Ecrire un algorithme permettant de lire au clavier le rayon *R* d'un cercle et un angle *a* (en degrés). L'algorithme doit calculer et afficher l'aire du secteur circulaire (= *pi* R^2 *a / 360*). Traduire l'algorithme en C.

Exercice 9 :
Pour convertir des degrés Fahrenheit en degrés Celsius, on a la formule suivante : $C \approx 0.55556*(F-32)$, où *F* est la température en degrés Fahrenheit, et *C* est la température correspondante en degrés Celsius.

- Ecrire un algorithme qui convertit une température entrée au clavier, exprimée en degrés Fahrenheit, et affiche une valeur approchée de la même température en degrés Celsius. Les températures seront exprimées par des nombres réels. Traduire l'algorithme en C.
- Même question pour la conversion inverse : de degrés Celsius en degrés Fahrenheit.

Exercice 10 :
Soit la fonction mathématique *f* définie par $f(x)=(2x+3)(3x^2+2)$.
Ecrire un algorithme qui calcule l'image par *f* d'un nombre saisi au clavier. Traduire l'algorithme en C.

Exercice 11 :
Une bille de plomb est lâchée du haut d'un immeuble et tombe en chute libre. Au bout d'un temps *t* (exprimé en secondes), la bille est descendue d'une hauteur *h* (en mètres) : $h=1/2 \ *g * t^2$, avec *g = 9.81*

- Ecrire un algorithme qui calcule la hauteur descendue au bout d'un temps *t* saisi au clavier. Traduire l'algorithme en C.
- Ecrire un programme C qui calcule la durée totale de la chute connaissant la hauteur total *h* de l'immeuble saisi au clavier. On pourra utiliser la fonction `sqrt` de la bibliothèque `math.h` qui calcule la racine carrée d'un nombre.

Exercice 12 :
Ecrire un algorithme permettant de saisir deux nombres, de les permuter puis les afficher. Traduire l'algorithme en C.

Note : La permutation de deux variables consiste à échanger (intervertir) leurs valeurs.

Exercice 13 :

Par un algorithme, exprimez un nombre de secondes sous forme d'heures, minutes et secondes. La seule donnée est le nombre total de secondes que nous appellerons *nsec*; les résultats consistent en trois nombres *h*, *m*, *s*. Traduire l'algorithme en C.

Exercice 14 :

Les exercices de 14 à 17 utilisent les déclarations suivantes en langage C :

```
char c1 = 'A', c2 = 'D' ;
int n = 10, p = 4 ;
long q = 2 ;
float x = 1.75 ;
```

Donnez le type et la valeur de chacune des expressions suivantes :

- `n + q`
- `n + x`
- `n % p * q`
- `n % -p * n`

Exercice 15 :

Donnez le type et la valeur de l'expression : `c1 + x`

Exercice 16 :

Donnez le type et la valeur de chacune des expressions suivantes :

- `n < p`
- `n >= p`
- `n > q`
- `q + 3 * (n > p)`
- `c1 > c2`
- `c1 > n`
- `(c2 > n) + (c1 < p)`
- `(c2 > n) && (c1 < p)`

Exercice 17 :

Donnez le type et la valeur de chacune des expressions suivantes :

- `(c1 > c2) || (c1 < c2)`
- `q && n`
- `(q - 2) && (n - 10)`
- `x * (q == 2)`

Exercice 18 :

Qu'affiche le programme C suivant ?

```
#include <stdio.h>
main()
{
  int n=10, p=5, q ;
```

```
  n += 2 ;
  p++ ;
  q = n/p ;
  printf("%d", q) ;
}
```

Exercice 19 :

- Essayez de mieux présenter le programme C suivant :

```
#include <stdio.h> main() { int
n ; scanf("%d",
&n
) ; printf(
"%d a pour double %d"
,
n, 2*
n) ; }
```

- Que se passe t-il si on enlève la déclaration `int n` ?
- Que se passe t-il si on remplace l'identificateur `n` par `if` ?

Exercice 20 :

Qu'affiche le programme C suivant ?

```
#include <stdio.h>
main()
{
  unsigned short a = 1, b=4;
  printf("%d  %d", a<<2, b>>1);
}
```


Exercice 21 :

Qu'affiche le programme C suivant ?

```
#include <stdio.h>
main()
{
  int n=7, p=4 ;
  printf("%d  %d  %d\n", n&p, n|p, n^p) ;
  printf("%d  %d  %d\n", n>>2, n>>4, n<<2) ;
  printf("%d  %d  %d", ~n, ~n>>1, ~n<<2) ;
}
```

Exercice 22 :

Ecrire un programme C qui affiche le code ASCII d'un caractère saisi au clavier.

Exercice 23 :

Ecrire un programme C qui :

- lit deux entiers *n1* et *n2* au clavier ;
- affiche la partie entière de leur quotient ;
- affiche la partie fractionnaire *frac* de leur quotient ;
- lit un nombre réel *l* au clavier ;

- calcule la partie entière du produit de *l* par *frac* modulo 256, puis convertit le résultat en caractère ;
- affiche le caractère obtenu.

Tester votre programme pour *n1*=1321, *n2*=500, *l*=500.

8.2. Corrigés

Solution 1 :

L'algorithme affiche :

```
231
462
```

Solution 2 :

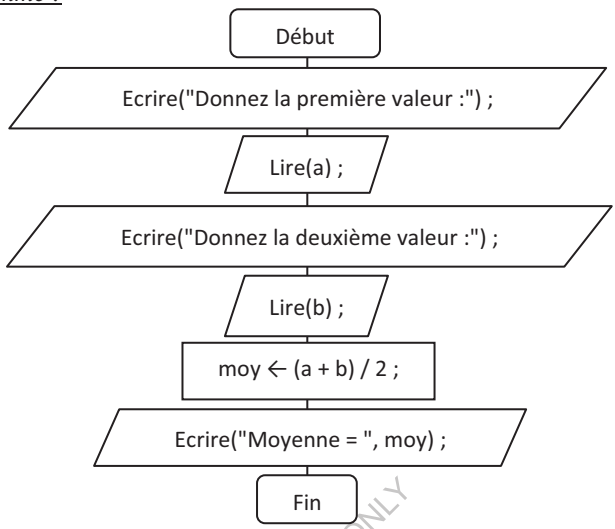
Algorithme :

```
Algorithme Moyenne
Variables
  Réel    a,b, moy ;
Début
  Ecrire("Donnez la première valeur :") ;
  Lire(a) ;
  Ecrire("Donnez la deuxième valeur :") ;
  Lire(b) ;
  moy ← (a + b) / 2 ;
  Ecrire("Moyenne = ", moy) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float a, b, moy;
  printf("Donnez la première valeur :\n");
  scanf("%f",&a);
  printf("Donnez la deuxième valeur :\n");
  scanf("%f",&b);
  moy = (a + b) / 2 ;
  printf("Moyenne = %f", moy);
}
```

Organigramme :



Déroulement de l'algorithme :

Pour le déroulement de l'algorithme dans le cas de *a=7* et *b=19*, on va noter les opérations comme suit :

Opération	Notation
`Ecrire("Donnez la première valeur :") ;`	1
`Lire(a) ;`	2
`Ecrire("Donnez la deuxième valeur :") ;`	3
`Lire(b) ;`	4
`moy ← (a + b) / 2 ;`	5
`Ecrire("Moyenne = ", moy) ;`	6

Le tableau suivant (tableau de situation) correspond au schéma d'évolution d'état des variables, opération par opération :

Variable / Opération	a	b	moy	Affichage (effectué par l'opération Ecrire)
1				`Donnez la première valeur :`
2	7			
3	7			`Donnez la deuxième valeur :`
4	7	19		
5	7	19	13	
6	7	19	13	`Moyenne = 13`

Solution 3 :

Algorithme :

```
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
```

Programme C :

```
#include <stdio.h>
main()
{
  float somme, produit, moyenne, nb1, nb2, nb3;
  printf("Donnez trois nombres :\n");
  scanf("%f %f %f",&nb1, &nb2, &nb3);
  somme = nb1 + nb2 + nb3;
  produit = nb1 * nb2 * nb3;
  moyenne = somme / 3 ;
  printf("La somme de ces trois nombres est : %f\n", somme);
  printf("Le produit de ces trois nombres est : %f\n", produit);
  printf("La moyenne de ces trois nombres est : %f\n", moyenne);
}
```


Solution 4 :

Algorithme :

```
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
```

Programme C :

```
#include <stdio.h>
main()
{
  int nb, c, d, t;
  printf("Donnez un nombre :\n");
```

```
  scanf("%d",&nb);
  c = nb * nb;
  d = nb * 2;
  t = nb * 3 ;
  printf("Le carré = %d\n", c);
  printf("Le double = %d\n", d);
  printf("Le triple = %d\n", t);
}
```

Solution 5 :

Algorithme :

```
Algorithme facture
Variables
  Réel    pu, tva, ttc ;
  Entier    nb ;
Début
  Ecrire("Donnez le prix unitaire hors taxes : ") ;
  Lire(pu) ;
  Ecrire("Donnez le nombre d'articles : ") ;
  Lire(nb) ;
  Ecrire("Donnez le taux de TVA : ") ;
  Lire(tva) ;
  ttc ← nb * pu + ((nb * pu) * tva) ;
  Ecrire("Le prix toutes taxes est : ", ttc) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float pu, tva, ttc ;
  int  nb ;
  printf("Donnez le prix unitaire hors taxes : ") ;
  scanf("%f", &pu) ;
  printf("Donnez le nombre d'articles : ") ;
  scanf("%d", &nb) ;
  printf("Donnez le taux de TVA : ") ;
  scanf("%f", &tva) ;
  ttc = nb * pu + ((nb * pu) * tva) ;
  printf("Le prix toutes taxes est : %f", ttc) ;
}
```


Solution 6 :

Algorithme :

```
Algorithme sphere
Constantes
   pi = 3.1416 ;
Variables
  Réel    R, Aire, Vol ;
Début
  Ecrire("Introduisez la valeur du rayon : ") ;
```

```
  Lire(R) ;
  Aire ← 4 * pi * R * R ;
  Vol ← 4/3 * pi * R * R * R ;
  Ecrire("Aire = ", Aire) ;
  Ecrire("Volume = ", Vol) ;
Fin
```

Programme C :

```
#include <stdio.h>
#define pi 3.1416
main()
{
  float R, Aire, Vol;
  printf("Introduisez la valeur du rayon : ") ;
  scanf("%f", &R) ;
  Aire = 4 * pi * R * R ;
  Vol = 4/3. * pi * R * R * R ;
  printf("Aire = %f\n", Aire) ;
  printf("Volume = %f", Vol) ;
}
```

Solution 7 :

Algorithme :

```
Algorithme triangle
Variables
  Réel    a, b, c, Perim, Aire, p;
Début
  Ecrire("Donnez la longueur du côté a : ") ;
  Lire(a) ;
  Ecrire("Donnez la longueur du côté b : ") ;
  Lire(b) ;
  Ecrire("Donnez la longueur du côté c : ") ;
  Lire(c) ;
  Perim ← a + b + c ;
  p ← Perim / 2 ;
  Aire ← (p * (p-a)*(p-b)*(p-c)) ^ (1/2) ;
  Ecrire("Périmètre = ", Perim) ;
  Ecrire("Aire = ", Aire) ;
Fin
```

Programme C :

```
#include <stdio.h>
#include <math.h>
main()
{
  float a, b, c, Perim, Aire, p ;
  printf("Donnez la longueur du côté a : ") ;
  scanf("%f", &a) ;
  printf("Donnez la longueur du côté b : ") ;
  scanf("%f", &b) ;
  printf("Donnez la longueur du côté c : ") ;
```

```
  scanf("%f", &c) ;
  Perim = a + b + c ;
  p = Perim / 2 ;
  Aire = sqrt((p * (p-a)*(p-b)*(p-c)) ) ;
  printf("Périmètre = %f\n", Perim) ;
  printf("Aire = %f", Aire) ;
}
```

Solution 8 :

Algorithme :

```
Algorithme secteur
Constantes
   pi = 3.1416 ;
Variables
  Réel    R, Angle, Aire ;
Début
  Ecrire("Introduisez le rayon : ") ;
  Lire(R) ;
  Ecrire("Introduisez l'angle (en degrés) : ") ;
  Lire(Angle) ;
  Aire ← pi * R * R * Angle / 360 ;
  Ecrire("Aire = ", Aire) ;
Fin
```

Programme C :

```
#include <stdio.h>
#define pi 3.1416
main()
{
  float R, Angle, Aire ;
  printf("Introduisez le rayon : ") ;
  scanf("%f", &R) ;
  printf("Introduisez l'angle (en degrés) : ") ;
  scanf("%f", &Angle) ;
  Aire = pi * R * R * Angle / 360 ;
  printf("Aire = %f", Aire) ;
}
```


Solution 9 :

Algorithme :

```
Algorithme Conversion_F_to_C
Variables
  Réel    C, F ;
Début
  Ecrire("Donnez la température en degrés Fahrenheit : ") ;
  Lire(F) ;
  C ← 0.55556 * (F - 32) ;
  Ecrire("La température en degrés Celsius = ", C) ;
Fin
```

Programme en C :

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

```
main()
{
  float C, F ;
  printf("Donnez la température en degrés Fahrenheit : ") ;
  scanf("%f", &F) ;
  C = 0.55556 * (F - 32) ;
  printf("La température en degrés Celsius = %f", C) ;
}
```

Programme en C pour la conversion inverse :

```
#include <stdio.h>
main()
{
  float C, F ;
  printf("Donnez la température en degrés Celsius : ") ;
  scanf("%f", &C) ;
  F = C/0.55556 + 32 ;
  printf("La température en degrés Fahrenheit = %f", F) ;
}
```

Solution 10 :

Algorithme :

```
Algorithme fct
Variables
  Réel    x, fx ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire(x) ;
  fx ← (2 * x + 3)/(3 * x * x + 2) ;
  Ecrire("F(", x, ") = ", fx) ;
Fin
```


Programme C :

```
#include <stdio.h>
main()
{
  float x, fx ;
  printf("Entrez un nombre : ") ;
  scanf("%f", &x) ;
  fx = (2 * x + 3)/(3 * x * x + 2) ;
  printf("F(%.2f) = %.2f", x, fx) ;
}
```

Solution 11 :

Algorithme :

```
Algorithme Calcul_hauteur
Variables
  Réel    h, t ;
Début
  Ecrire("Entrez une durée : ") ;
  Lire(t) ;
  h ← ½ * 9.81 * t * t ;
```

```
  Ecrire("Hauteur = ", h) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float h, t ;
  printf("Entrez une durée : ") ;
  scanf("%f", &t) ;
  h = 1/2. * 9.81 * t * t ;
  printf("Hauteur = %f", h) ;
}
```

Programme C : calcul du temps en fonction de la hauteur

```
#include <stdio.h>
#include<math.h>
main()
{
  float h, t ;
  printf("Entrez une hauteur : ") ;
  scanf("%f", &h) ;
  t = sqrt(2*h/9.81) ;
  printf("Temps = %f", t) ;
}
```


Solution 12 :

Permutation en utilisant une variable intermédiaire :

Algorithme :

```
Algorithme Permuter1
Variables
  Entier    x, y, z ;
Début
  Ecrire("Donnez la valeur de x : ") ;
  Lire(x) ;
  Ecrire("Donnez la valeur de y : ") ;
  Lire(y) ;
  z ← x ;
  x ← y ;
  y ← z ;
  Ecrire("Après permutation :") ;
  Ecrire("x = ", x) ;
  Ecrire("y = ", y) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int x, y, z ;
  printf("Donnez la valeur de x : ") ;
  scanf("%d", &x) ;
```

```
  printf("Donnez la valeur de y : ") ;
  scanf("%d", &y) ;
  z = x ;
  x = y ;
  y = z ;
  printf("Après permutation :\n") ;
  printf("x = %d\n", x) ;
  printf("y = %d", y) ;
}
```

Permutation sans variable intermédiaire :

Algorithme :

```
Algorithme Permuter2
Variables
 Entier    x, y ;
Début
  Ecrire("Donnez la valeur de x : ") ;
  Lire(x) ;
  Ecrire("Donnez la valeur de y : ") ;
  Lire(y) ;
  x ← x + y ;
  y ← x – y ;
  x ← x – y ;
  Ecrire("Après permutation :") ;
  Ecrire("x = ", x) ;
  Ecrire("y = ", y) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int x, y ;
  printf("Donnez la valeur de x : ") ;
  scanf("%d", &x) ;
  printf("Donnez la valeur de y : ") ;
  scanf("%d", &y) ;
  x = x + y ;
  y = x - y ;
  x = x - y ;
  printf("Après permutation :\n") ;
  printf("x = %d\n", x) ;
  printf("y = %d", y) ;
}
```


Solution 13 :

Algorithme :

```
Algorithme conversion
Variables
  Entier    nsec, h, m, s ;
Début
```

```
  Ecrire("Introduisez le nombre de secondes : ") ;
  Lire(nsec) ;
  s ← nsec % 60 ;
  m ← (nsec % 3600) / 60 ;
  h ← nsec / 3600 ;
  Ecrire(nsec, "seconde(s) vaut : ", h, "heure(s), ", m, " minute(s), et ", s, " seconde(s)") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int nsec, h, m, s ;
  printf("Introduisez le nombre de secondes : ") ;
  scanf("%d", &nsec) ;
  h = nsec / 3600 ;
  m = (nsec % 3600) / 60 ;
  s = nsec % 60 ;
  printf("%d seconde(s) vaut : %d heure(s) %d minute(s), et %d seconde(s)", nsec, h, m, s) ;
}
```

Solution 14 :

Le type et la valeur de chacune des expressions suivantes :

- `n + q` Type : `long`, valeur : `12`
- `n + x` Type : `float`, valeur : `11.750000`
- `n % p * q` Type : `long`, valeur : `4`
- `n % -p * n` Type : `int`, valeur : `20`

Solution 15 :

Le type et la valeur de l'expression suivante :

`c1 + x` Type : `float`, valeur : `66.750000`

Solution 16 :

Le type et la valeur de chacune des expressions suivantes :

- `n < p` Type : `int`, valeur : `0`
- `n >= p` Type : `int`, valeur : `1`
- `n > q` Type : `int`, valeur : `1`
- `q + 3 * (n > p)` Type : `long`, valeur : `5`
- `c1 > c2` Type : `int`, valeur : `0`
- `c1 > n` Type : `int`, valeur : `1`
- `(c2 > n) + (c1 < p)` Type : `int`, valeur : `1`
- `(c2 > n) && (c1 < p)` Type : `int`, valeur : `0`

Solution 17 :

Le type et la valeur de chacune des expressions suivantes :

- `(c1 > c2) || (c1 < c2)` Type : `int`, valeur : `1`
- `q && n` Type : `int`, valeur : `1`
- `(q - 2) && (n - 10)` Type : `int`, valeur : `0`

- `x * (q == 2)` Type : `float`, valeur : `1.750000`

Solution 18 :

Le programme affiche : `2`

Solution 19 :

- Mieux présenter le programme C :

```
#include <stdio.h>
main()
 {
    int n ;
    scanf(&n) ;
    printf("%d a pour double %d", n, 2*n) ;
 }
```

- Si on enlève la déclaration `int n`, une erreur de compilation surgit : `n undeclared`.
- Si on remplace l'identificateur `n` par `if`, une erreur de compilation surgit refusant d'utiliser `if` comme identificateur, car `if` est un mot clé.

Solution 20 :

Le programme C affiche : `4  2`

Solution 21 :

Le programme C affiche :

```
4  7  3
1  0  28
-8  -4  -32
```


Solution 22 :

Programme C :

```
#include <stdio.h>
main()
{
  char c ;
  printf("Tapez un caractère : ") ;
  scanf("%c", &c) ;
  printf("Le code ASCII du caractère %c est %d.", c, c) ;
}
```

Solution 23 :

Programme C :

```
#include <stdio.h>
main()
{
  int n1, n2, quotient;
  float frac, l;
  char c;
  printf("Donnez deux entiers : \n") ;
  scanf("%d %d", &n1, &n2) ;
```

```
  quotient = n1 / n2;
  printf("Partie entière du quotient : %d\n", quotient) ;
  frac = n1/(float)n2 - quotient ;
  printf("Partie fractionnaire du quotient : %f\n", frac) ;
  printf("Donnez un nombre : ") ;
  scanf("%f", &l) ;
  c = ((int)(l*frac) % 256);
  printf("Le caractère obtenu est : %c", c) ;
}
```

*Avec **n1**=1321, **n2**=500, **l**=500, le programme affiche :*

```
Donnez deux entiers :
1321
500
Partie entière du quotient : 2
Partie fractionnaire du quotient : 0.642000
Donnez un nombre : 500
Le caractère obtenu est : A
```

Chapitre 3 : Les structures conditionnelles

1. Introduction

Les algorithmes vus précédemment sont exécutés séquentiellement. Les ruptures des séquences peuvent être effectuées par des structures de contrôle classées en deux catégories : les structures conditionnelles et les boucles. Les structures conditionnelles (simples, composées et multiples) sont aussi appelées structures alternatives, structures de choix ou les tests.

2. Structure conditionnelle simple

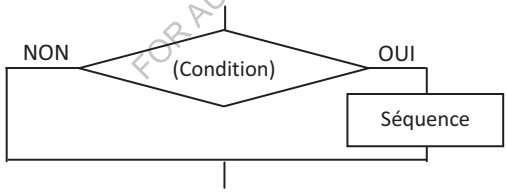
Format général : `Si (Condition) <Séquence>`

Cette opération est lue comme suit : Si la condition est vérifiée (`VRAI`), alors la séquence d'opérations s'exécute.

La condition est une expression logique qui retourne la valeur `VRAI` ou `FAUX`. L'expression peut être simple (condition simple) ou composée (plusieurs conditions composées avec des opérateurs logiques `ET`, `OU` et `NON`).

La séquence peut contenir une ou plusieurs opérations. Si la séquence contient plusieurs opérations, alors elles sont séparées par des points-virgules et mises entre `début` et `fin`. Si la séquence contient une seule opération, alors les mots `début` et `fin` ne sont pas obligatoires.

La structure conditionnelle simple peut être représentée dans un organigramme comme suit :



Voyons les exemples suivants :

- `Si (X > 10) Ecrire(X) ;`
  On affiche la valeur de `X` si elle est supérieure à `10`.
- `Si (X > 10) début Ecrire(X) ; Y ← X ; fin`
  On affiche la valeur de `X` et on affecte la valeur de `X` à `Y`, si `X` est supérieure à `10`.
- `Si ((X > 10) ET (X < 15)) Ecrire(X) ;`
  On affiche la valeur de `X` si elle est prise entre `10` et `15`.
- `Si (X > 10) Si (X < 15) Ecrire(X) ;`
  Cet exemple est équivalent à l'exemple précédent, sauf que cette fois-ci on utilise des tests imbriqués.
- `Si (X < 10) Si (X > 15) Ecrire(x) ;`

Dans cet exemple, la valeur de `X` n'est jamais affichée, car il n'y a aucun cas qui satisfait la condition.

En C, la structure conditionnelle simple s'écrit sous la forme :

```
if (Condition) <Bloc>
```

Exemple : `if (X > 10) printf("%d", X);`

Remarques :

- Dans un programme C, on utilise le terme bloc au lieu de séquence. Le bloc est constitué donc d'une ou plusieurs instructions.
- La condition est obligatoirement mise entre `(` et `)`. Chaque instruction du bloc doit se terminer par un point-virgule (`;`).
- Si le bloc contient plus d'une instruction, alors les deux accolades `{` et `}` sont obligatoires pour délimiter la suite d'instructions. Les accolades sont facultatives s'il n'y a qu'une seule instruction.
- Si on met un point-virgule juste après la condition, le compilateur considère que le bloc de `if` est constitué d'une seule instruction (rien…). Par exemple : `if(x>0); printf("bonjour") ;`
  ici, le `printf` ne se trouve pas dans le `if`, mais à l'extérieur, et par conséquent, il sera exécuté quelle que soit la valeur de `x`.
- Comme il n'y a pas de type booléen en C, alors une expression non nulle (différente de `0`) est considérée comme vraie. Une expression nulle (égale à `0`) est considérée comme fausse.

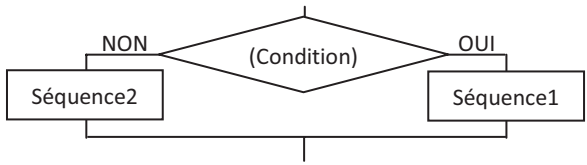
3. Structure conditionnelle composée

Format général : `Si (condition) <Séquence1> Sinon <Séquence2>`

Cette opération est lue comme suit : Si la condition est vérifiée (`VRAI`), alors les opérations de la `Séquence1` sont exécutées. Dans le cas contraire, ce sont les opérations de la `Séquence2` qui vont être exécutées.

Les mots `début` et `fin` sont utilisés pour délimiter une séquence de plusieurs opérations.

La structure conditionnelle composée peut être représentée dans un organigramme comme suit :



Soit l'exemple suivant :

```
Si (X > 10) Ecrire(X) ;
Sinon Ecrire("valeur non acceptée") ;
```

Dans cet exemple, la valeur de `X` est affichée si elle est supérieure à `10`, sinon on affiche le message : `valeur non acceptée`

En C, on aura :

```
if (x > 10) printf("%d", x) ;
else printf("valeur non acceptée") ;
```

Remarques :

- Si nous avions une suite d'instructions dans un bloc, alors on aurait dû utiliser les deux accolades `{` et `}`.
- Le `else` se rapporte toujours au `if`… le plus proche. Pour casser ce rapport, il est possible d'utiliser `{` et `}`. Par exemple, dans le cas de :

```
if (x > 10) { if (x < 20) printf("%d",x) ; }
            else printf("valeur non acceptée") ;
```

  le `else` suit le premier `if` et non pas le deuxième.

4. Structure conditionnelle multiple

La structure conditionnelle multiple, appelée aussi l'alternative classifiée ou le choix multiple, permet de comparer un objet (expression) à toute une série de valeurs, et d'exécuter une séquence d'opérations parmi plusieurs, en fonction de la valeur effective de l'objet. Une séquence par défaut peut être prévue dans le cas où l'objet n'est égal à aucune des valeurs énumérées.

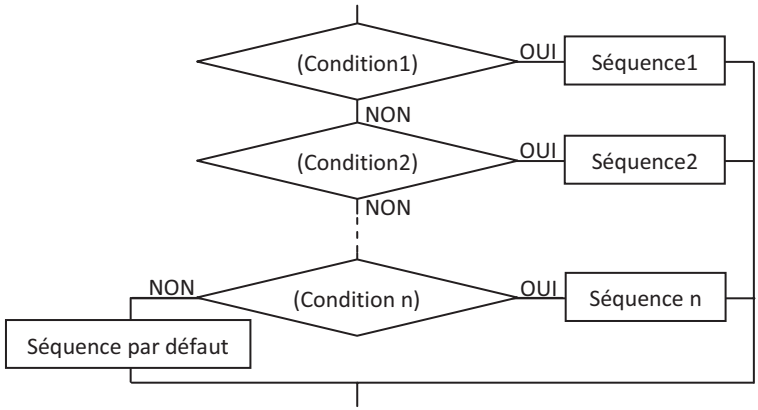
Chaque séquence est étiquetée par une valeur. Pour que cette séquence soit choisie, il faut que sa valeur soit équivalente à l'expression. La structure conditionnelle multiple se présente comme suit :

```
Selon (expression)
  début
    cas Valeur1 : <Séquence1>
    cas Valeur2 : <Séquence2>
    …
    cas Valeurn : <Séquence n>
    Défaut : <Séquence par défaut>
  fin
```

Ceci est équivalent à :

```
Si (expression = Valeur1) <Séquence1>
Sinon Si (expression = Valeur2) <Séquence2>
…
Sinon Si (expression = Valeur n) <Séquence n>
Sinon <Séquence par défaut>
```

La structure conditionnelle multiple peut être représentée dans un organigramme comme suit :



Dans l'exemple suivant, la valeur de `X` est affichée en lettres, si elle est égale à `1` ou `2`, sinon on affiche un message :

```
Selon (X)
  début
    cas Valeur1 : Ecrire("Un") ;
    cas Valeur2 : Ecrire("Deux") ;
    Défaut : Ecrire("Valeur sup à deux ou inf à un") ;
  fin
```


Remarques :

- Dans l'opération de choix multiple, l'ordre de présentation ne change rien.
- Le cas défaut est facultatif.

En C, ça s'écrit :

```
switch (X)
{
  case 1 : printf("Un"); break ;
  case 2 : printf("Deux") ; break ;
  default : printf("Valeur sup à deux ou inf à un") ;
}
```

Ou bien :

```
switch (X)
{
  case 1 : { printf("Un"); break ;}
  case 2 : {printf("Deux") ; break ;}
  default : {printf("Valeur sup à deux ou inf à un") ; }
}
```

Remarques :

- En langage C, l'expression et les valeurs à choisir doivent être de type `int` ou `char`.

- L'instruction `break` termine l'exécution de l'instruction englobante (`switch`) dans laquelle elle apparaît. En cas d'oubli de `break`, les instructions des cas suivants seront aussi exécutées.
  Par exemple, en l'absence de `break`, et pour `X = 2`, l'instruction :

```
 switch (X)
{
  case 1 : printf("Un");
  case 2 : printf("Deux") ;
  default : printf("Valeur sup à deux ou inf à un") ;
}
```

  affiche : `DeuxValeur sup à deux ou inf à un`
  Maintenant, pour l'exemple :

```
switch (X)
{
  case 1 :
  case 2 : printf("%d", X) ; break ;
  default : printf("Valeur sup à deux ou inf à un") ;
}
```

  Le traitement est le même (l'affichage de la valeur de `X`) pour `X` égale à `1` ou `X` égale à `2`.
- Pour une souplesse de programmation, il est possible de mettre :

```
#define val  10
…
switch (n)
{ ….
  case val – 1 :…
  case val :…
  case val + 1 :…
…}
```

  Une modification de `val` se résume à une seule intervention au niveau de la directive `#define`.
- Si vous dupliquez les cas, par exemple :

```
switch (X)
{
  case 1 : printf("bonjour"); break;
  case 1 : printf("merci") ;  break;
  default : printf("Valeur sup à deux ou inf à un") ;
}
```

  Vous aurez une erreur de compilation de la forme *duplicate case value*.

5. Le branchement

Il est possible d'effectuer un saut direct vers une opération en utilisant une opération de branchement de la forme `aller à <étiquette>`.

Exemple :

```
Algorithme afficher_nbr_positif
  Variables
    Entier    I ;
```

```
Début
  Lire(I) ;
  Si (I<0)  aller à etiq1 ;
  Ecrire(I) ;
  etiq1 : Ecrire("Merci") ;
Fin
```

En C, ça s'écrit :

```
#include <stdio.h>
main()
{
  int I;
  scanf("%d", &I);
  if (I<0) goto etiq1 ;
  printf("%d\n", I) ;
  etiq1 : printf("Merci") ;
}
```

On note qu'il est déconseillé d'utiliser l'instruction de branchement, et cela pour réduire la complexité des programmes en termes de temps.

6. Exercices corrigés

6.1. Exercices

Exercice 1 :

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif. Si le nombre vaut zéro, on le considère positif. Représenter l'algorithme par un organigramme, ensuite le traduire en C.

Exercice 2 :

Ecrire un algorithme qui affiche la valeur absolue d'un nombre réel lu à partir du clavier. Traduire l'algorithme en C.

Exercice 3 :

Ecrire un algorithme qui compare deux nombres réels lus à partir du clavier. Traduire l'algorithme en C.

Exercice 4 :

Ecrire un algorithme qui permet de dire si un nombre lu à partir du clavier est pair ou impair. Traduire l'algorithme en C.

Exercice 5 :

Ecrire un algorithme qui affiche le résultat d'un étudiant (accepté ou rejeté) à une matière, sachant que cette matière est évaluée par une note d'oral (coefficient 1) et une note d'écrit (coefficient 2). La moyenne obtenue doit être supérieure ou égale à 10 pour valider la matière. Traduire l'algorithme en C.

Exercice 6 :

Ecrire un algorithme permettant de déterminer le plus grand de trois nombres réels lus à partir du clavier. Traduire l'algorithme en C.

Exercice 7 :
Ecrire un algorithme qui demande deux nombres à l'utilisateur, et l'informe ensuite si le produit est positif, négatif ou nul. Attention ! On ne doit pas calculer le produit. Traduire l'algorithme en C.

Exercice 8 :
Ecrire un algorithme qui permet de calculer les racines de l'équation du second degré suivante : $ax^2 + bx + c = 0$. Comme entrées, l'utilisateur doit introduire les facteurs *a*, *b* et *c*. Traduire l'algorithme en C.

Exercice 9 :
Ecrire un algorithme qui demande d'entrer un nombre entre 1 et 7, et donne le nom du jour correspondant (samedi, dimanche…) en utilisant la structure conditionnelle multiple. Traduire l'algorithme en C.

Exercice 10 :
Ecrire un algorithme permettant à partir d'un menu affiché, d'effectuer la somme, le produit ou la moyenne de trois nombres. Nous appelons menu, l'association d'un numéro séquentiel aux différents choix proposés par un programme. Traduire l'algorithme en C.

Exercice 11 :
Ecrire un algorithme permettant de déterminer si l'année *A* est bissextile. On doit savoir que si *A* n'est pas divisible par 4, l'année n'est pas bissextile. Si *A* est divisible par 4, l'année est bissextile, sauf si *A* est divisible par 100 et non pas par 400. Traduire l'algorithme en C. Tester votre programme pour les années 111, 1984, 1900 et 800.

Exercice 12 :
Ecrire un algorithme permettant d'effectuer la transformation des coordonnées cartésiennes *(x,y)* en coordonnées polaires *(r,t)*. Cette transformation se fait par les formules :

- $r^2 = x^2+y^2$
- Si $x = 0$ alors : $t = pi/2$ si $y > 0$; $t = - pi/2$ si $y < 0$; t n'existe pas si $y = 0$.
- Sinon $t = arctg\ (y/x)$ auquel il faut ajouter *pi* si $x<0$.

Traduire l'algorithme en C.

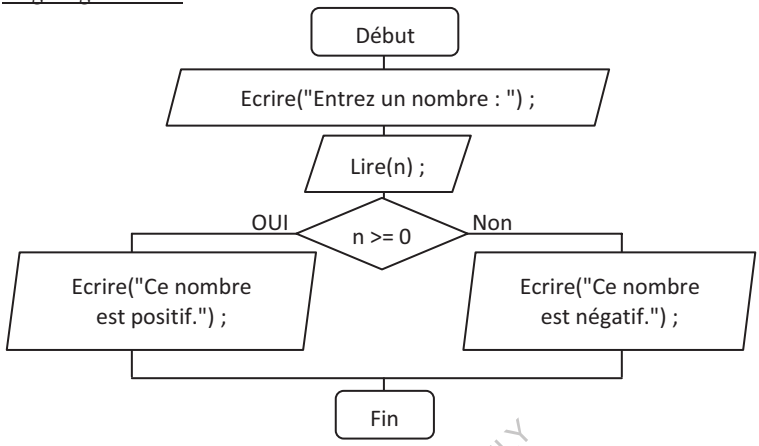
6.2. Corrigés

Solution 1 :
Algorithme :

```
Algorithme positif_negatif
Variables
  Entier    n ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire(n) ;
  Si (n >= 0) Ecrire("Ce nombre est positif.") ;
```

```
  Sinon Ecrire("Ce nombre est négatif.") ;
Fin
```

Organigramme :



Programme C :

```
#include <stdio.h>
main()
{
  int n ;
  printf("Entrez un nombre : ") ;
  scanf("%d", &n) ;
  if (n >= 0)  printf("Ce nombre est positif.");
  else printf("Ce nombre est négatif.") ;
}
```


Solution 2 :

Algorithme :

```
Algorithme valeur_absolue
Variables
  Réel    X, Val_abs ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire(X) ;
  Val_abs ← X ;
  Si (Val_abs < 0) Val_abs ← -Val_abs ;
  Ecrire("La valeur absolue de ", X, "est ", Val_abs) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float X, Val_abs ;
  printf("Entrez un nombre : ") ;
```

```
  scanf("%f", &X) ;
  Val_abs = X ;
  if (Val_abs < 0)  Val_abs = -Val_abs ;
  printf("La valeur absolue de %f est %f", X, Val_abs) ;
}
```

Solution 3 :

Algorithme :

```
Algorithme comparaison
Variables
  Réel    X, Y ;
Début
  Ecrire("Entrez deux nombres X et Y :") ;
  Lire(X, Y) ;
  Si (X = Y) Ecrire(X, " = ", Y) ;
  Sinon Si (X > Y) Ecrire(X, " est supérieur à ", Y) ;
         Sinon Ecrire(X, " est inférieur à ", Y) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float X, Y ;
  printf("Entrez deux nombres X et Y :\n") ;
  scanf("%f%f", &X, &Y) ;
  if (X == Y) printf("%f = %f", X, Y);
  else if (X > Y) printf("%f est supérieur à %f", X, Y) ;
       else printf("%f est inférieur à %f", X, Y) ;
}
```


Solution 4 :

Algorithme :

```
Algorithme pair_impair
Variables
  Entier    X ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire (X) ;
  Si (X % 2 = 0) Ecrire(X, " est un nombre pair.") ;
  Sinon Ecrire(X, " est nombre impair.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int X ;
  printf("Entrez un nombre : ") ;
  scanf ("%d", &X) ;
  if (X % 2 == 0) printf("%d est un nombre pair.", X) ;
  else printf("%d est un nombre impair.", X) ;
```

```
}
```

Solution 5 :

Algorithme :

```
Algorithme evaluation
Variables
  Réel    ne, no, moy ;
Début
  Ecrire("Entrez la note de l'examen écrit : ") ;
  Lire(ne) ;
  Ecrire("Entrez la note de l'examen oral : ") ;
  Lire(no) ;
  moy ← (2 * ne + no)/3 ;
  Si (moy >= 10) Ecrire ("accepté") ;
  Sinon Ecrire("rejeté") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float ne, no, moy ;
  printf("Entrez la note de l'examen écrit : ") ;
  scanf("%f", &ne) ;
  printf("Entrez la note de l'examen oral : ") ;
  scanf("%f", &no) ;
  moy = (2 * ne + no)/3 ;
  if (moy >= 10) printf ("accepté");
  else printf("rejeté") ;
}
```


Solution 6 :

Algorithme :

```
Algorithme plus_grand
Variables
  Réel    x, y, z, pg ;
Début
  Ecrire ("Entrez trois nombres :") ;
  Lire(x, y, z) ;
  Si ((x>= y) et (x>= z)) pg ← x ;
  Sinon Si (y >= z) pg ← y ;
        Sinon pg ← z ;
  Ecrire ("Le plus grand des trois nombres est : ", pg) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float x, y, z, pg ;
  printf ("Entrez trois nombres :\n") ;
  scanf("%f%f%f", &x, &y, &z) ;
```

```
  if ((x>= y) && (x>= z)) pg = x ;
  else if (y >= z) pg = y ;
       else pg = z ;
  printf ("Le plus grand des trois nombres est : %f", pg) ;
}
```

Solution 7 :

Algorithme :

```
Algorithme resultat_produit
Variables
  Entier    m, n ;
Début
  Ecrire("Entrez deux nombres :") ;
  Lire(m, n) ;
  Si ((m=0) OU (n=0))
      Ecrire("Le produit de ", m, " et ",n, " est nul.") ;
  Sinon Si ((m > 0) ET ( n > 0)) OU ((m < 0) ET (n < 0))
         Ecrire("Le produit de ", m, " et ",n, " est positif.") ;
  Sinon Ecrire("Le produit de ", m, " et ", n, " est négatif.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int m, n ;
  printf("Entrez deux nombres :\n") ;
  scanf("%d%d", &m, &n) ;
  if ((m == 0) || ( n == 0))
           printf("Le produit de %d et %d est nul.", m, n) ;
  else if (((m > 0) && ( n > 0)) || ((m < 0) && (n < 0)))
         printf("Le produit de %d et %d est positif.", m, n) ;
    else printf("Le produit de %d et %d est négatif.", m, n) ;
}
```


Solution 8 :

Algorithme :

```
Algorithme Eq_2de_degre
Variables
  Réel    a, b, c, d, x1, x2, x ;
Début
  Ecrire("Entrez les valeurs a, b et c de l'équation :") ;
  Lire(a, b, c) ;
  d ← b * b - 4 * a * c ;
  Si (d>0) début
    x1← (-b - √d) / (2*a) ;
    x2 ← (-b + √d) / (2*a) ;
    Ecrire("L'équation possède deux racines :") ;
    Ecrire("x1= ", x1, " x2= ", x2) ;
  fin
```

```
  Sinon Si (d=0) début
                x ← -b / (2*a) ;
                Ecrire("L'équation possède une racine :") ;
                Ecrire("x= ", x) ;
        fin
       Sinon Ecrire("L'équation ne possède pas de racines.") ;
Fin
```

Programme C :

```
#include <stdio.h>
#include <math.h>
main()
{
  float a, b, c, d, x1, x2, x ;
  printf("Entrez les valeurs a, b et c de l'équation :\n") ;
  scanf("%f%f%f", &a, &b, &c) ;
  d = b * b - 4 * a * c ;
  if (d>0) {
    x1 = (-b - sqrt(d)) / (2*a) ;
    x2 = (-b + sqrt(d)) / (2*a) ;
    printf("L'équation possède deux racines :\n") ;
    printf("x1= %f, x2= %f", x1, x2) ;
  }
  else if (d=0) {
            x = -b / (2*a) ;
            printf("L'équation possède une racine :\n") ;
            printf("x= %f", x) ;
        }
        else printf("L'équation ne possède pas de racines.") ;
}
```


Solution 9 :

Algorithme :

```
Algorithme jour
Variables
  Entier    no ;
Début
  Ecrire("Entrez le numéro du jour : 1.Samedi, 2.Dimanche… : ") ;
  Lire(no) ;
  Selon (no)
    début
        cas 1 : Ecrire("Samedi") ;
        cas 2 : Ecrire("Dimanche") ;
        cas 3 : Ecrire("Lundi") ;
        cas 4 : Ecrire("Mardi") ;
        cas 5 : Ecrire("Mercredi") ;
        cas 6 : Ecrire("Jeudi") ;
        cas 7 : Ecrire("Vendredi") ;
        Défaut : Ecrire("Numéro non accepté.") ;
    fin
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int no ;
  printf("Entrez le numéro du jour : 1.Samedi, 2.Dimanche… : ") ;
  scanf("%d", &no) ;
  switch (no)
  {
      case 1 : printf("Samedi") ; break;
      case 2 : printf("Dimanche") ; break;
      case 3 : printf("Lundi") ; break;
      case 4 : printf("Mardi") ; break;
      case 5 : printf("Mercredi") ; break;
      case 6 : printf("Jeudi") ; break;
      case 7 : printf("Vendredi") ; break;
      default : printf("Numéro non accepté.") ;
  }
}
```

Solution 10 :

Algorithme :

```
Algorithme menu
Variables
  Réel    nb1, nb2, nb3 ;
  Entier    Choix ;
Début
  Ecrire("Entrez trois nombres :") ;
  Lire(nb1, nb2, nb3) ;
  /* Affichage du menu et saisie du choix */
  Ecrire("1-pour la multiplication") ;
  Ecrire("2-pour la somme") ;
  Ecrire("3-pour la moyenne") ;
  Ecrire("Votre choix : ") ;
  Lire (choix) ;
  Selon (choix)
    début
      cas 1 : Ecrire("Le produit des trois nombres est : ", nb1 * nb2 * nb3) ;
      cas 2 : Ecrire("La somme des trois nombres est : ", nb1 + nb2 + nb3) ;
      cas 3 : Ecrire("La moyenne des trois nombres est : ", (nb1 + nb2 + nb3)/3) ;
      Défaut : Ecrire("Choix incorrect !") ;
    fin
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float nb1, nb2, nb3 ;
  int Choix ;
```

```
  printf("Entrez trois nombres :\n") ;
  scanf("%f%f%f", &nb1, &nb2, &nb3) ;
  /* Affichage du menu et saisie du choix */
  printf("1-pour la multiplication\n") ;
  printf("2-pour la somme\n") ;
  printf("3-pour la moyenne\n") ;
  printf("Votre choix : ") ;
  scanf("%d", &Choix) ;
  switch (Choix)
  {
   case 1 : printf("Le produit des trois nombres est : %f", nb1 * nb2 * nb3) ; break;
   case 2 : printf("La somme des trois nombres est : %f", nb1 + nb2 + nb3) ; break;
   case 3 : printf("La moyenne des trois nombres est : %f", (nb1 + nb2 + nb3)/3) ;
            break;
   default : printf("Choix incorrect !") ;
  }
}
```

Solution 11 :

Algorithme :

```
Algorithme bissextile
Variables
  Entier    A ;
Début
  Ecrire("Introduisez l'année : ") ;
  Lire(A) ;
  Si NON (A %  4 = 0)
          Ecrire("L'année ", A, " n'est pas bissextile.") ;
  Sinon  Si ((A %  100 = 0) ET NON (A % 400 = 0))
          Ecrire("L'année ", A, " n'est pas bissextile.") ;
         Sinon Ecrire("L'année ", A, " est bissextile.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int A ;
  printf("Introduisez l'année : ") ;
  scanf("%d", &A) ;
  if (! (A %  4 == 0))
               printf("L'année %d n'est pas bissextile.", A);
  else if ((A %  100 == 0) && (! (A % 400 == 0)))
              printf("L'année %d n'est pas bissextile.", A) ;
       else printf("L'année %d est bissextile.", A) ;
}
```

Après test :

On a obtenu : `111 n'est pas bissextile.` ; `1984 est bissextile.` ; `1900 n'est pas bissextile.` ; et `800 est bissextile.`

Solution 12 :

Algorithme :

```
Algorithme Cartesien_Polaire
Constantes
  Pi=3.14 ;
Variables
  Réel    x, y, r, t ;
Début
  Ecrire("Introduisez l'abscisse x : ") ;
  Lire(x) ;
  Ecrire("Introduisez l'ordonnée y : ") ;
  Lire(y) ;
  r ← √(x²+y²) ;
  Si (x=0)
     Si (y>0) Ecrire("r= ", r ," et t= ", pi/2) ;
     Sinon Si (y<0) Ecrire("r= ", r ," et t= ", -pi/2) ;
           Sinon Ecrire("r= ", r ," et t n'existe pas.") ;
  Sinon début
         t ← arctg (y/x) ;
         Si (x<0) t ← t + pi ;
         Ecrire("r= ", r ," et t= ", t) ;
        fin
Fin
```

Programme C :

```
#include <stdio.h>
#include <math.h>
#define Pi 3.14
main()
{
  float x, y, r, t ;
  printf("Introduisez l'abscisse x : ") ;
  scanf("%f", &x) ;
  printf("Introduisez l'ordonnée y : ") ;
  scanf("%f", &y) ;
  r = sqrt(pow(x,2)+pow(y,2));
  if (x==0) if (y>0) printf("r= %f et t= %f", r, Pi/2) ;
            else if (y<0) printf("r= %f et t= %f", r, -Pi/2);
                 else printf("r= %f et t n'existe pas.", r) ;
    else {
          t = atan(y/x) ;
          if (x<0) t = t + Pi ;
          printf("r= %f et t= %f", r, t) ;
         }
}
```

Chapitre 4 : Les boucles

1. Introduction

Lorsqu'on veut répéter une opération ou une suite d'opérations plusieurs fois dans un algorithme, il est possible d'utiliser des structures répétitives (itératives) appelées les boucles. Il existe trois types de boucles : `Tant que`, `Répéter` et `Pour`.

2. La boucle Tant que

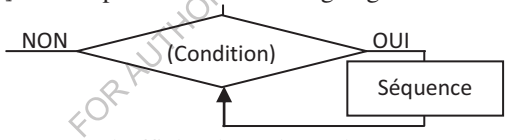
Format général : `Tant que (Condition) <Séquence>`

Cette structure permet la répétition d'une séquence d'opérations tant que la condition est satisfaite (= `VRAI`). Quand la condition devient fausse, la boucle est terminée.

La condition est une expression logique. Il faut alors que cette expression puisse changer de valeur (avoir la valeur `FAUX`) pour sortir de la boucle et éviter le cas de la *boucle infinie*.

La séquence est une ou plusieurs opérations. Pour plusieurs opérations, le `début` et `fin` sont obligatoires. Dans le cas d'une seule opération, le `début` et `fin` sont optionnels.

La boucle `Tant que` est représentée dans un organigramme comme suit :



L'exemple suivant permet d'afficher les valeurs de `1` à `10` :

```
i ← 1 ;
Tant que (i <= 10) début
        Ecrire(i) ;
        i ← i + 1 ;
    fin
```

En C, la boucle `Tant que` s'exprime comme suit :

```
i = 1 ;
while (i <= 10) {
       printf("%d\n", i);
       i = i + 1 ;
     }
```

Remarques :

- En langage C, dans une instruction `while`, la condition doit être délimitée par `(` et `)`. Chaque instruction du bloc doit se terminer par un point-virgule (`;`).

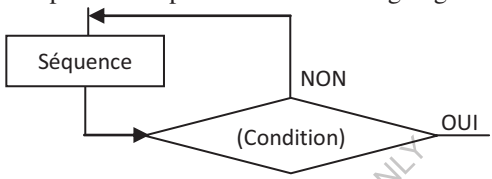
- Si le bloc contient plusieurs instructions, alors elles sont obligatoirement délimitées par `{` et `}`. Si ce bloc contient une seule instruction, les accolades sont facultatives.
- Condition vraie veut dire que la valeur de l'expression est non nulle. Condition fausse veut dire que la valeur de l'expression est `0`.

3. La boucle Répéter

Format général : `Répéter <Séquence> Jusqu'à (Condition)`

Cette structure permet la répétition d'une ou plusieurs opérations jusqu'à ce qu'une condition soit vérifiée (= `VRAI`). Pour éviter la boucle infinie, il faut que la condition puisse changer de valeur (avoir la valeur `VRAI`).

La boucle `Répéter` peut être représentée dans un organigramme comme suit :



Reprenons l'exemple permettant d'afficher les valeurs de `1` à `10` :

```
i ← 1;
Répéter
  début
    Ecrire(i);
    i ← i + 1;
  fin
Jusqu'à (i > 10) ;
```


Le tableau suivant récapitule les différences entre les boucles `Tant que` et `Répéter` :

Boucle	Condition d'exécution	Condition pour quitter	Nombre d'exécutions
Tant que	Condition = VRAI	Condition = FAUX	n'est pas connu d'avance, mais elle peut ne jamais être exécutée si la condition = FAUX dès le départ.
Répéter	Condition = FAUX	Condition = VRAI	n'est pas connu d'avance, mais elle est exécutée au moins une fois.

Remarques :

- Pour la boucle `Tant que` et `Répéter`, on utilise souvent une variable, dite indice, initialisée à une valeur initiale avant d'exécuter la boucle. A l'intérieur de la boucle, l'indice sera incrémenté (en ajoutant une valeur à la valeur courante), ou décrémenté (en diminuant la valeur courante) pour atteindre la condition d'arrêt de la boucle.

- Une itération est la réalisation d'un cycle complet de boucle. Ceci inclut le test de la condition et l'exécution de la séquence. Ceci est valable pour toutes les boucles.
- En C, il n'existe pas une boucle `Répéter` proprement dite. Il existe une instruction qui fait presque le même travail. Il s'agit de l'instruction `do … while…`, mais en réalité, c'est un simple échange de position entre le bloc d'instructions et la condition de la boucle `while` (`Tant que`), permettant ainsi l'exécution du bloc d'instructions au moins une seule fois, sauf que la condition d'arrêt est la même que la boucle `while`, i.e. condition = `Faux`. L'exemple précédent devient ainsi :

```
i = 1 ;
do{
  printf("%d\n",i);
  i = i + 1 ;
} while (i <= 10);
```

4. La boucle Pour

Format général :

```
Pour (Opération de départ ; Condition ; Pas de progression) <Séquence>
```

Avec :

1. Opération de départ : c'est une opération exécutée une seule fois au début de la boucle. On peut mettre plusieurs opérations séparées par des virgules. Il s'agit généralement d'une ou plusieurs initialisations.
2. Condition : c'est une condition testée avant d'exécuter la séquence. Si la condition est vérifiée (=`VRAI`), on exécute la séquence. Si la condition n'est pas vérifiée, on quitte la boucle sans exécuter la séquence.
3. Pas de progression, indiqué par une opération (ou plusieurs opérations séparées par des virgules) exécutée(s) après l'exécution de la séquence. Après avoir exécuté l'opération ou les opérations correspondant au Pas de progression, on revient à l'étape 2 (tester la Condition). Le Pas de progression doit donc nous ramener à une Condition ayant la valeur `FAUX` pour qu'on puisse quitter la boucle. Généralement, il s'agit de la modification de la variable (ou des variables) initialisée(s) dans l'étape 1 (Opération de départ).

Le format le plus utilisé de la boucle `Pour` est le suivant :

```
Pour (compteur ← val_initiale; compteur <= val_finale; compteur ← compteur + incrément) <Séquence>
```

Ou bien :

```
Pour (compteur ← val_initiale; compteur >= val_finale; compteur ← compteur - décrément) <Séquence>
```

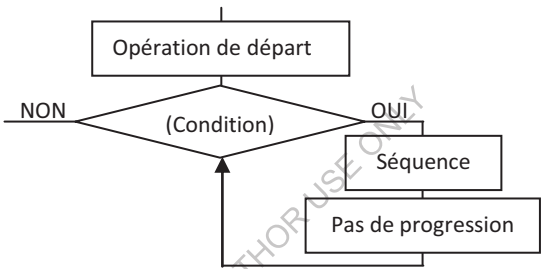
Cette structure permet de répéter l'exécution d'une séquence d'opérations pour toutes les valeurs d'une variable de contrôle (`compteur`) à partir d'une valeur

initiale (`val_initiale`) à une valeur finale (`val_finale`). Après chaque exécution de la séquence, la variable de contrôle est incrémentée d'une unité (`incrément`), ou décrémentée d'une unité (`décrément`). Quand la variable de contrôle dépasse la valeur finale, on quitte la boucle sans exécuter la séquence de la boucle.

Remarques :

- La variable de contrôle est de type `Entier`.
- Dans la séquence d'une boucle `Pour`, le compteur (variable de contrôle) peut être utilisé, mais il ne doit jamais être modifié à l'intérieur de la boucle.
- La valeur initiale, la valeur finale et l'incrément peuvent être des expressions numériques.

La boucle `Pour` peut être représentée dans un organigramme comme suit :



Gardons le même exemple permettant d'afficher les valeurs de `1` à `10` :

```
Pour (i ← 1 ; i<=10 ; i ← i + 1) Ecrire(i) ;
```

En C, la boucle `Pour` s'exprime comme suit :

```
for (i = 1 ; i <=10 ; i = i + 1) printf("%d\n", i) ;
```

Ou bien : `for (i = 1 ; i <=10 ; i+=1) printf("%d\n", i) ;`

Ou même : `for (i = 1 ; i <=10 ; i++) printf("%d\n", i) ;`

Remarques :

- En langage C, si le bloc d'instructions de la boucle `for` contient plusieurs instructions, alors elles sont obligatoirement délimitées par `{` et `}`. Si ce bloc contient une seule instruction, alors les accolades sont facultatives.
- Après avoir quitté la boucle `for(i=1 ; i<=10 ; i++) <Bloc>` la variable `i` aura la valeur `11`.
- En langage C, le format général suivant peut être utilisé :

  `for(<Instruction de départ> ; <Condition >; <Pas de progression>) <Bloc>`

  avec (1) `<Instruction de départ>` (généralement une initialisation) est exécutée une seule fois au début de la boucle ; (2) la `Condition` est évaluée au début de chaque itération ; (3) Une modification indiquant un `Pas de progression` est exécutée à la fin de chaque itération.

– L'instruction `break` sert à interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. Par exemple, la boucle :

```
for (i = 1; i<=4; i++) {
   if (i==2) break;
   printf("%d", i);
}
```

affiche : `1`

On note ici qu'après avoir quitté la boucle, la variable `i` aura la valeur `2`.

– L'instruction `continue` permet de passer au tour de boucle suivant. Par exemple, la boucle :

```
for (i = 1; i<=4; i++) {
   if (i==2) continue;
   printf("%d", i);
}
```

affiche : `134`

– Il est possible d'initialiser/modifier plusieurs variables dans la boucle `for` en même temps. Ces initialisations/modifications sont séparées par des virgules. Par exemple, `for(x=1, y=10; x<y; x++,y--) <Bloc>` permet de répéter le bloc d'instructions cinq fois.

– L'instruction `for(; ;) <Bloc>` est une boucle infinie dont on ne pourra sortir que par une instruction `break`.

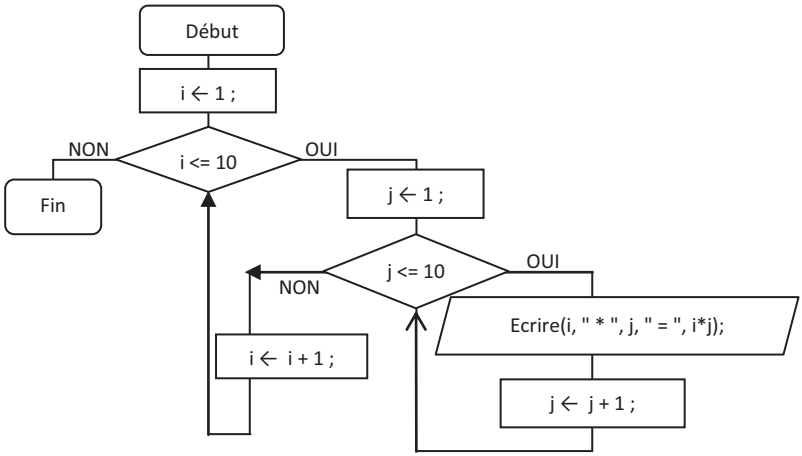
5. Les boucles imbriquées

Les boucles peuvent être imbriquées les unes dans les autres. Une boucle `Tant que` peut contenir une autre boucle `Tant que`, une autre boucle `Répéter`, ou une autre boucle `Pour`, et vice versa. De plus, une boucle peut contenir une autre boucle, qui elle-même peut contenir une autre boucle, et ainsi de suite.

L'algorithme suivant permet d'afficher les tables de multiplication de `1` jusqu'à `10` :

```
Algorithme boucles_imbriquees
Variables
       Entier    i, j ;
Début
  Pour (i ← 1 ; i<=10 ; i ← i+1) début
      j ← 1 ;
      Tant que (j<=10) début
        Ecrire(i, " * ", j, " = ", i*j) ;
        j ← j + 1 ;
      Fin
Fin
```

L'organigramme correspondant à l'algorithme précèdent est le suivant :



Programme C :

```
#include <stdio.h>
main()
{
  int i,j ;
  for (i=1; i<=10; i++) {
      j = 1 ;
      while (j<=10) {
        printf("%d * %d = %d\n", i, j, i*j) ;
        j = j + 1 ;
      }
  }
}
```


6. Exercices corrigés

6.1. Exercices

Exercice 1 :

Ecrire des algorithmes permettant de calculer la somme de la suite des nombres *1, 2, 3, ..., n* (*n* est un entier positif lu à partir du clavier) en utilisant les boucles `Tant que`, `Répéter` et `Pour`. Représentez chaque algorithme par l'organigramme correspondant.

Exercice 2 :

Ecrire un algorithme qui calcule la factorielle d'un nombre entier *n* strictement positif, sachant que : *n! = 1*2*3*...*n*, et que *0! = 1*. Traduire l'algorithme en C.

Exercice 3 :

Ecrire un algorithme qui calcule la $n^{ième}$ puissance entière d'un entier *x* par multiplications successives du nombre par lui-même. Traduire l'algorithme en C.

Exercice 4 :
Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne. Traduire l'algorithme en C.

Exercice 5 :
Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Traduire l'algorithme en C.

Exercice 6 :
Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre. Traduire l'algorithme en C.

Exercice 7 :
Etant donnés deux nombres entiers *m* et *n* positifs non nuls. Ecrire un algorithme permettant de déterminer le PGCD (plus grand diviseur commun) de *m* et *n*. Traduire l'algorithme en C.
Note : Le PGCD peut être calculer en utilisant l'algorithme itératif d'Euclide qui prend d'abord le reste de la division de *m* par *n*, puis le reste de la division de *n* par ce premier reste, etc., jusqu'à ce qu'on trouve un reste nul. Le dernier diviseur utilisé sera le PGCD de *m* et *n*.

Exercice 8 :
Ecrire un algorithme permettant de saisir une série de nombres entiers positifs. On termine la saisie par un nombre négatif qui ne sera pas tenu en compte lors des calculs. Puis, on propose indéfiniment (en boucle) à l'utilisateur, par l'intermédiaire d'une sorte de menu à choix multiple, d'afficher la valeur minimale, la valeur maximale, la somme ou la moyenne des nombres entrés, ou encore de quitter le programme. Traduire l'algorithme en C.

Exercice 9 :
Ecrire un algorithme permettant de lire la suite des prix des achats d'un client (en dinars entiers positifs). La suite se termine par un zéro. L'algorithme doit permettre de calculer la somme qu'il doit, lire la somme qu'il paye, et simuler la remise de la monnaie en affichant les textes « 10 DA », « 5 DA » et « 1 DA » autant de fois qu'il y a de coupures de chaque sorte à rendre. Traduire l'algorithme en C.

Exercice 10 :
Ecrire un algorithme qui calcule itérativement le $n^{ième}$ terme de la suite de Fibonacci définie comme suit : Si $n = 0$ alors $F_n = 0$; Si $n = 1$ alors $F_n = 1$; Si $n > 1$ alors $F_n = F_{n-1}+F_{n-2}$.
Traduire l'algorithme en C.

Exercice 11 :
Ecrire un programme C permettant d'afficher un triangle rempli d'étoiles, s'étendant sur un nombre de lignes lu à partir du clavier, et se présentant comme dans cet exemple :

```
Donnez le nombre de lignes : 4
*
**
***
****
```

Exercice 12 :

1. Que fait le programme C suivant ?

```
#include <stdio.h>
main()
{
   int x;
    while(1){
          scanf("%d", &x);
          if(x==1) break;
    }
}
```

2. Que devient-il si on enlève l'instruction `break` ?
3. Que se passe t-il si on remplace `while(1)` par `while(0)` ?

6.2. Corrigés

Solution 1 :

Algorithmes :

Algorithme Tant que :

```
Algorithme somme_suite1
Variables
  Entier    i, n, somme ;
Début
  Ecrire ("Donnez un nombre :") ;
  Lire (n) ;
  Si (n > 0) début
    somme ← 0 ;
    i ← 1 ;
    Tant que (i <= n) début
      somme ← somme + i ;
      i ← i + 1 ;
    fin
    Ecrire ("La somme de la suite = ", somme) ;
  fin
  Sinon Ecrire ("Le nombre doit être positif.") ;
Fin
```

Algorithme Répéter :

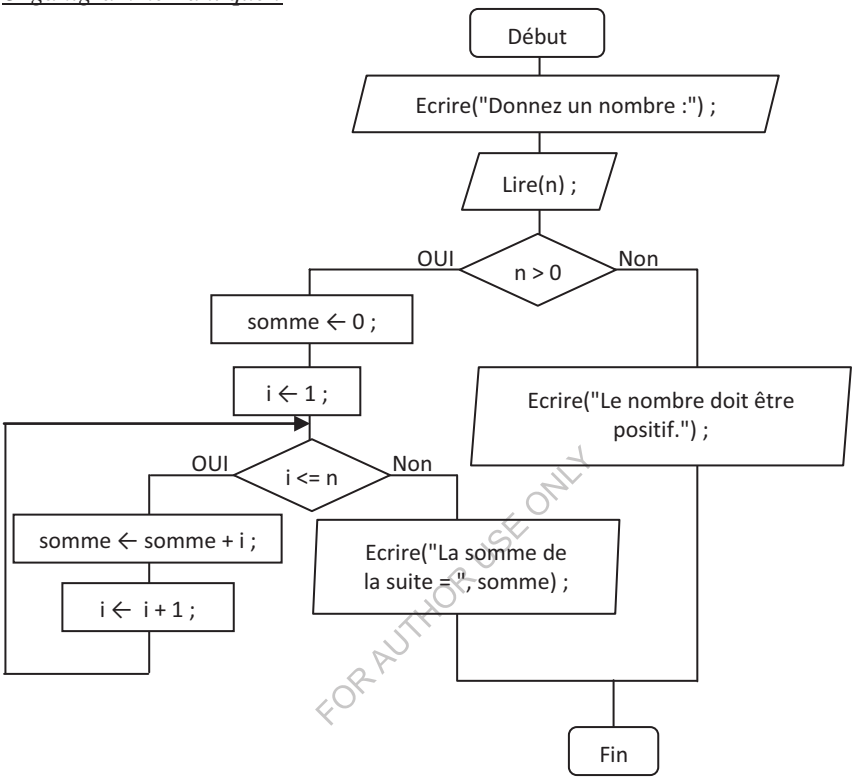
```
Algorithme somme_suite2
Variables
  Entier    i, n, somme ;
Début
  Ecrire ("Donnez un nombre :") ;
  Lire (n) ;
```

```
  Si (n > 0) début
    somme ← 0 ;
    i ← 1 ;
    Répéter
      début
        somme ← somme + i ;
        i ← i + 1 ;
      fin
    jusqu'à (i > n) ;
    Ecrire ("La somme de la suite = ", somme) ;
  Fin
  Sinon Ecrire ("Le nombre doit être positif.") ;
 Fin
```

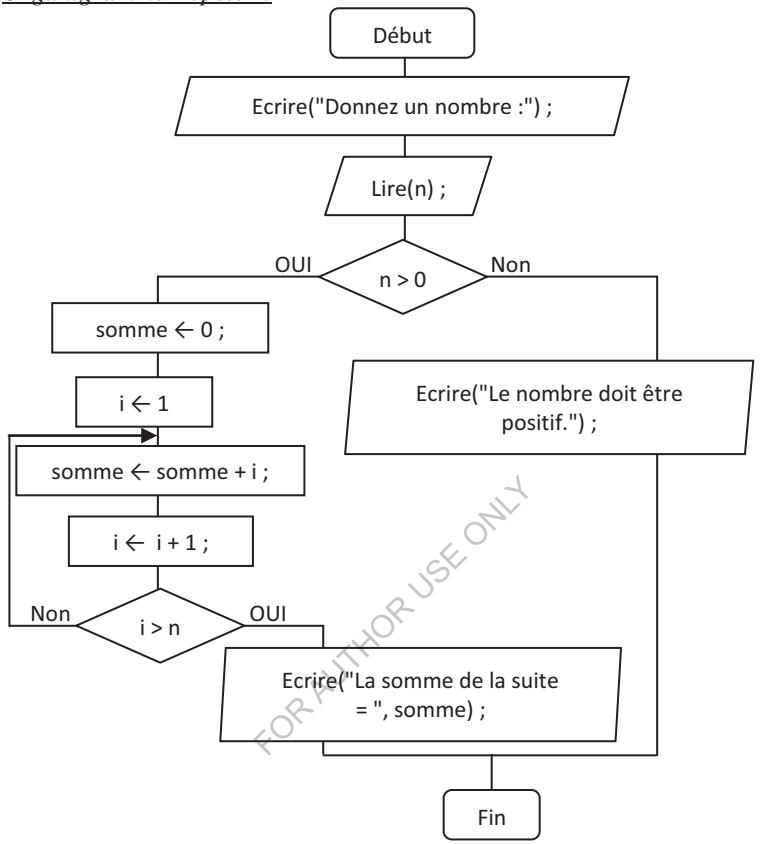
Algorithme Pour :

```
Algorithme somme_suite3
Variables
  Entier    i, n, somme ;
Début
  Ecrire("Donnez un nombre :") ;
  Lire (n) ;
  Si (n > 0) début
    somme ← 0 ;
    Pour (i ← 1 ; i<= n ; i ← i + 1) somme ← somme + i ;
    Ecrire("La somme de la suite = ", somme) ;
  fin
  Sinon Ecrire("Le nombre doit être positif.") ;
Fin
```

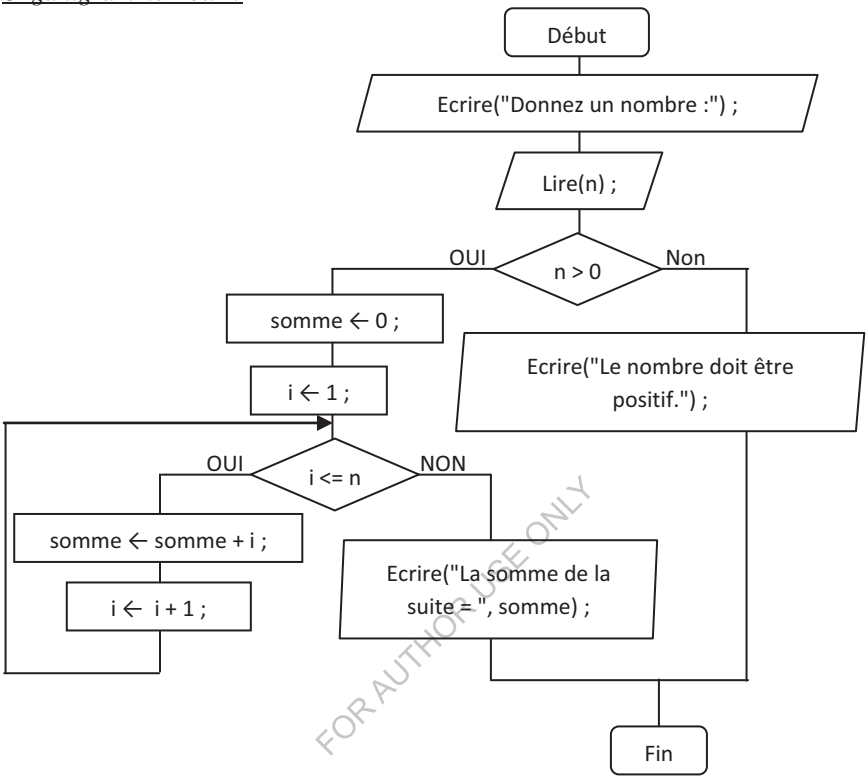
Organigrammes :
Organigramme Tant que :



Organigramme Répéter :



Organigramme Pour :



Solution 2 :

Algorithme :

```
Algorithme factorielle
Variables
  Entier    i, n, factorielle ;
Début
  Ecrire("Entez un nombre : ") ;
  Lire (n) ;
  Si (n < 0)
    Ecrire("La factorielle d'un nombre négatif n'existe pas.")
  Sinon début
   factorielle ← 1 ;
   Pour (i ← 1; i<= n ; i ← i + 1) factorielle ← factorielle * i ;
   Ecrire("La factorielle de ", n, " = ", factorielle) ;
  fin
Fin
```

Programme C :

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

```
main()
{
  int i, n, factorielle ;
  printf ("Entez un nombre : ") ;
  scanf("%d", &n) ;
  if (n < 0)
   printf ("La factorielle d'un nombre négatif n'existe pas.");
  else {
    factorielle = 1 ;
    for (i=1; i<=n; i++)  factorielle = factorielle * i ;
    printf ("La factorielle de %d = %d", n, factorielle) ;
   }
}
```

Solution 3 :

Algorithme :

```
Algorithme puissance
Variables
  Entier    n, i ;
  Réel    x, puiss ;
Début
  Ecrire("Calcul de la puissance n ième du réel x par multiplications successives.") ;
  Ecrire("Introduisez le nombre réel x : ") ;
  Lire(x) ;
  Ecrire("Introduisez l'exposant de x (entier positif) : ") ;
  Lire(n) ;
  puiss ← 1 ;
  i ← 1 ;
  Tant que (i <= n) début
      puiss ← puiss * x ;
      i ← i + 1 ;
  fin
  Ecrire("La puissance ", n," ième de ", x," est ",puiss) ;
Fin
```


Programme C :

```
#include <stdio.h>
main()
{
  int n, i ;
  float x, puiss ;
  printf("Calcul de la puissance n ième du réel x par multiplications successives.\n") ;
  printf("Introduisez le nombre réel x : ") ;
  scanf("%f", &x) ;
  printf("Introduisez l'exposant de x (entier positif) : ") ;
  scanf("%d", &n) ;
  puiss = 1 ;
  i = 1 ;
  while (i <= n) {
      puiss = puiss * x ;
      i = i + 1 ;
```

```
  }
  printf("La puissance %d ième de %f est %f", n, x, puiss) ;
}
```

Solution 4 :

Algorithme :

```
Algorithme valeur_1_3
Variables
  Entier    n ;
Début
  Répéter
    début
      Ecrire("Entrez un nombre entre 1 et 3 :") ;
      Lire(n) ;
      Si ((n < 1) OU (n > 3))
        Ecrire("Saisie erronée. Recommencez !") ;
      Sinon Ecrire("Nombre accepté.") ;
    fin
  Jusqu'à ((n >= 1) ET (n <= 3));
Fin
```

Programme C :

Comme il n'existe pas une structure qui correspond exactement à la boucle `Répéter` en C, nous allons utiliser l'instruction `do… while…` comme suit :

```
#include <stdio.h>
main()
{
  int n  ;
  do{
    printf("Entrez un nombre entre 1 et 3 :\n") ;
    scanf("%d", &n) ;
    if ((n < 1) || (n > 3))
      printf("Saisie erronée. Recommencez !\n");
    else printf("Nombre accepté.") ;
  }
  while ((n < 1) || (n > 3));
}
```


Solution 5 :

Algorithme :

```
Algorithme afficher_10_suivants
Variables
  Entier    n, i ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire(n) ;
  Ecrire("Les 10 nombres suivants sont :") ;
  Pour (i ← (n + 1) ; i <= (n + 10) ; i ← i +1) Ecrire(i) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int n, i ;
  printf("Entrez un nombre : ") ;
  scanf("%d", &n) ;
  printf("Les 10 nombres suivants sont :\n") ;
  for(i=(n+1); i<= (n+10); i++) printf("%d\n", i) ;
}
```

Solution 6 :

Algorithme :

```
Algorithme table_multiplication
Variables
  Entier    n, i ;
Début
  Ecrire("Entrez un nombre : ") ;
  Lire(n) ;
  Ecrire("La table de multiplication de ",n , " est :") ;
  Pour (i ← 1 ; i <= 10; i ← i+1) Ecrire(n,' x ',i, ' = ', n*i ) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int n, i ;
  printf("Entrez un nombre : ") ;
  scanf("%d", &n) ;
  printf("La table de multiplication de %d est :\n", n) ;
  for(i=1; i<=10; i++) printf("%d x %d = %d\n", n, i, n*i ) ;
}
```


Solution 7 :

Algorithme :

```
Algorithme Plus_Grand_Diviseur_Commun
Variables
  Entier    m, n, a, b, r, PGCD ;
Début
  Ecrire("Nous allons calculer le PGCD de deux nombres entiers.")  ;
  Ecrire("Introduisez le premier nombre : ") ;
  Lire(m) ;
  Ecrire("Introduisez le deuxième nombre : ") ;
  Lire(n) ;
  a ← m ;
  b ← n ;
  r ← a % b ;
  Tant que (NON(r = 0)) début
    a ← b ;
    b ← r ;
    r ← a % b;
```

```
  fin
  PGCD ← b ;
  Ecrire("Le PGCD de ",m," et ",n," est ",PGCD) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int m, n, a, b, r, PGCD ;
  printf("Nous allons calculer le PGCD de deux nombres entiers.\n")  ;
  printf("Introduisez le premier nombre : ") ;
  scanf("%d", &m) ;
  printf("Introduisez le deuxième nombre : ") ;
  scanf("%d", &n) ;
  a = m ;
  b = n ;
  r = a % b ;
  while (!(r == 0)) {
    a = b ;
    b = r ;
    r = a % b;
  }
  PGCD = b ;
  printf("Le PGCD de %d et %d est %d", m, n, PGCD) ;
}
```


Solution 8 :

Algorithme :

```
Algorithme calculs_menu
Variables
  Entier    i, choix ;
  Réel    s, n, moyenne, minimum, maximum ;
Début
  Ecrire("Entrez successivement des nombres positifs.") ;
  Ecrire("Entrez un nombre négatif pour finir.") ;
  Lire (n) ;
  Si (n >= 0) début
    s ← n ;
    minimum ← n ;
    maximum ← n ;
    i ← 1 ;
    Répéter
      début
        Lire (n);
        Si (n >= 0) début
          Si (n < minimum) minimum ← n ;
          Si (n > maximum) maximum ← n ;
          s ← s + n ;
          i ← i + 1 ;
         fin
```

```
      fin
    jusqu'à (n < 0) ;
    moyenne ← s/i ;
    Répéter
     début
      Ecrire("Choisissez entre :") ;
      Ecrire("1 : minimum") ;
      Ecrire("2 : maximum") ;
      Ecrire("3 : somme") ;
      Ecrire("4 : moyenne") ;
      Ecrire("0 : stop") ;
      Ecrire("Entrez votre choix : ") ;
      Lire(choix) ;
      Cas (choix)
        début
          1 : Ecrire("Le minimum est : ", minimum) ;
          2 : Ecrire("Le maximum est : ", maximum) ;
          3 : Ecrire("La somme est : ", s) ;
          4 : Ecrire("La moyenne est : ", moyenne) ;
          0 :
          Défaut : Ecrire("Choix non accepté.") ;
        fin
     fin
    Jusqu'à (choix = 0) ;
  fin
Fin
```


Programme C :

```
#include <stdio.h>
main()
{
  int i, choix ;
  float s, n, moyenne, minimum, maximum ;
  printf("Entrez successivement des nombres positifs.\n") ;
  printf("Entrez un nombre négatif pour finir.\n") ;
  scanf("%f", &n) ;
  if (n >= 0) {
    s = n ;
    minimum = n ;
    maximum = n ;
    i = 1 ;
    do
      {
        scanf("%f", &n);
        if (n >= 0) {
          if (n < minimum) minimum = n ;
          if (n > maximum) maximum = n ;
          s = s + n ;
          i = i + 1 ;
                                      }
```

```
      }
    while (n >= 0) ;
    moyenne = s/i ;
    do
     {
      printf("Choisissez entre :\n") ;
      printf("1 : minimum\n") ;
      printf("2 : maximum\n") ;
      printf("3 : somme\n") ;
      printf("4 : moyenne\n") ;
      printf("0 : stop\n") ;
      printf("Entrez votre choix : ") ;
      scanf("%d", &choix) ;
      switch (choix)
        {
          case 1 : printf("Le minimum est : %f\n", minimum) ;
                   break;
          case 2 : printf("Le maximum est : %f\n", maximum) ;
                   break;
          case 3 : printf("La somme est : %f\n", s) ; break;
          case 4 : printf("La moyenne est : %f\n", moyenne) ;
                   break ;
          case 0 : break ;
          default : printf("Choix non accepté.\n") ;
        }
     }
    while (choix != 0) ;
  }
}
```


Solution 9 :

Algorithme :

```
Algorithme Achat
Variables
  Entier    prix, somme, M, Reste, Nb10D, Nb5D, Nb1D ;
Début
  somme ← 0 ;
  Ecrire("Donnez les prix, et terminez par un 0 :") ;
  Répéter
   début
     Ecrire("Entrez un prix : ") ;
     Lire(prix) ;
     Si (prix > 0) somme ← somme + prix ;
   fin
  Jusqu'à (prix = 0) ;
  Ecrire("Vous devez : ", somme, " DA") ;
  Ecrire("Montant versé : ") ;
  Lire(M) ;
  Reste ← M – somme ;
```

```
  Nb10D ← Reste / 10 ;
  Nb5D ← (Reste % 10) / 5 ;
  Nb1D ← Reste % 5;
  Ecrire("Remise de la monnaie :") ;
  Ecrire("Pièces de 10 DA : ", Nb10D) ;
  Ecrire("Pièces de 5 DA : ", Nb5D) ;
  Ecrire("Pièces de 1 D : ",Nb1D) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int prix, somme, M, Reste, Nb10D, Nb5D, Nb1D ;
  somme = 0 ;
  printf("Donnez les prix, et terminez par un 0 :\n");
  do
   {
     printf("Entrez un prix : ") ;
     scanf("%d", &prix) ;
     if (prix > 0) somme = somme + prix ;
   }
  while (prix != 0) ;
  printf("Vous devez : %d DA\n", somme) ;
  printf("Montant versé : ") ;
  scanf("%d", &M) ;
  Reste = M - somme ;
  Nb10D = Reste / 10 ;
  Nb5D = (Reste % 10) / 5 ;
  Nb1D = Reste % 5;
  printf("Remise de la monnaie :\n") ;
  printf("Pièces de 10 DA : %d\n", Nb10D) ;
  printf("Pièces de 5 DA : %d\n", Nb5D) ;
  printf("Pièces de 1 D : %d\n",Nb1D) ;
}
```


Solution 10 :

Algorithme :

```
Algorithme fibonacci
Variables
  Entier    n, fibo, pred0, pred1, i ;
Début
  Ecrire("Donnez un entier : ") ;
  Lire(n) ;
  Si (n = 0) fibo ← 0 ;
    Sinon Si (n = 1) fibo ← 1 ;
      Sinon Si (n > 1) début
                pred0 ← 0 ;
                pred1 ← 1 ;
                Pour (i ← 2 ; i<= n ; i ← i+1) début
                  fibo ← pred0 + pred1 ;
```

```
                  pred0 ← pred1 ;
                  pred1 ← fibo  ;
                fin
              fin
  Ecrire("Fibonacci(", n, ") = ", fibo) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int n, fibo, pred0, pred1, i ;
  printf("Donnez un entier : ") ;
  scanf("%d", &n) ;
  if (n == 0) fibo = 0 ;
    else if (n == 1) fibo = 1 ;
      else if (n > 1) {
                pred0 = 0 ;
                pred1 = 1 ;
                for(i=2; i<=n; i++){
                  fibo = pred0 + pred1 ;
                  pred0 = pred1 ;
                  pred1 = fibo  ;
                }
              }
  printf("Fibonacci(%d) = %d", n, fibo) ;
}
```

Solution 11 :

Programme C :

```
#include <stdio.h>
main()
{
  int n, i, j;
  printf("Donnez le nombre de lignes : ") ;
  scanf("%d", &n) ;
  for (i=1; i<=n; i++){
    for (j=1; j<=i; j++) printf("*");
    printf("\n") ;
 }
}
```

Solution 12 :

1. Le programme permet de lire une série de valeurs, jusqu'à introduire la valeur 1.
2. Boucle infinie.
3. Le bloc n'est jamais exécuté.

Chapitre 5 : Les tableaux et les chaînes de caractères

1. Introduction

Les types présentés jusqu'à maintenant sont des types simples. Il existe d'autres types dits structurés (complexes). Un type structuré est tout type défini à base d'autres types. Commençant par le type qu'on juge le plus important des types structurés, à savoir le type tableau.

2. Le type tableau

Un tableau est une structure de données homogène composée d'un ensemble d'éléments de même type de données.

Format général : `type_données  Nom_tab[taille_tab];`

Un tableau possède un nom (`Nom_tab`). Il est aussi caractérisé par sa taille, i.e. le nombre d'éléments ou encore le nombre de cases (`taille_tab`). Le type des éléments du tableau est indiqué par `type_données`.

Par exemple, pour déclarer un tableau `Note` de dix éléments réels, on met :

```
Réel  Note[10];
```

Cet exemple nous a permis de substituer la déclaration de 10 variables `Note0`, `Note1`, …, `Note9` de type `Réel` par une seule structure, à savoir le tableau `Note`.

En C, on écrit : `float Note[10];`

La capacité d'un tableau (le nombre d'éléments que peut contenir un tableau) ne peut pas être changée au cours d'exécution d'un programme. Il faut donc que la taille du tableau soit suffisamment grande pour la manipulation. Pour une souplesse de programmation, on peut utiliser la déclaration suivante :

```
#define taille_tab 10
main()
{
  float Note[taille_tab];
  ….
```

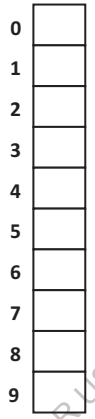
Remarques :

- Les cases d'un tableau sont numérotées de `0` à `taille_tab-1` en langage C.
- Le nom du tableau désigne l'adresse où débute le tableau en mémoire, i.e. `Note` est équivalent à `&Note[0]`.
- Le nom du tableau ne peut pas figurer à gauche d'une affectation (*lvalue*).
- Si vous utilisez le tableau hors de ses limites, il n'y aura pas d'erreur de compilation. On appelle ça, débordement ou dépassement de tableau. C'est l'un des bugs de la programmation en C.
- Pour la manipulation des tableaux, on utilise fréquemment les boucles.

- Il est possible d'omettre la dimension du tableau. Celle-ci étant déterminée par le nombre de valeurs énumérées dans l'initialisation. Par exemple, `float Note[] = {10.5, 11}` indique que le tableau contient deux éléments.

2.1. Manipulation d'un tableau

Un tableau peut être représenté par un ensemble de cases contigües. Le tableau `Note`, avec 10 éléments, peut être représenté comme suit :



Un élément du tableau est accessible par sa position (indice) dans le tableau. Le tableau `Note` est un tableau à une seule dimension, là où chaque élément est accessible par un seul indice. Par exemple, `Note[0] ← 0.25` permet d'affecter la valeur `0.25` à la première case du tableau. Le tableau `Note` devient:

L'indice peut être exprimé directement comme un nombre en clair. Il peut aussi correspondre à une valeur d'une variable ou à une expression calculée.

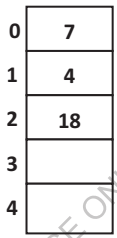
Soit par exemple :

```
Algorithme exe_tab
Variables
  Entier T[5];
  Entier  X ;
Début
  X ← 2;
  T[0] ← 7;
  T[X] ← 18;
  T[X-1] ← 4;
Fin
```

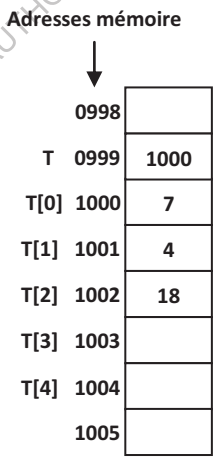
En C :

```
main()
{
  int T[5] ;
  int X ;
  X = 2;
  T[0] = 7;
  T[X] = 18;
  T[X-1] = 4;
}
```

Après l'exécution des opérations de l'algorithme, le tableau `T` peut être représenté comme suit :



Une représentation plus proche à la réalité est la suivante : (Voir encore la section 2 du chapitre 2)



Avec, le contenu de la case `T` correspond à `&T[0]` qui est égale à `1000`; le nom du tableau `T` est donc un pointeur vers le premier élément (contient son adresse).

Remarques :

- En C, l'indice peut être aussi de type caractère, compte tenu des règles de conversion implicite. Par exemple, `T['A'-64]` est équivalent à `T[1]`.
- Lorsqu'on applique `sizeof` à une variable de type tableau, l'opérateur renvoie l'espace mémoire total que ce tableau occupe, i.e. son nombre d'éléments multiplié par l'espace mémoire occupé par un seul élément.

Exercice 1 : (Création, initialisation et édition d'un tableau)
Problème : Ecrire un algorithme permettant de créer un tableau de dix entiers, d'initialiser ses éléments à `0`, ensuite de les afficher. Traduire l'algorithme en C.

Solution :

```
Algorithme init_tab
Variables
  Entier T[10] ;
  Entier i ;
Début
  Pour (i ← 0 ; i <= 9 ; i ← i+1) T[i] ← 0 ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Ecrire(T[i]) ;
Fin
```

Le programme C :

```
#include <stdio.h>
main()
{
  int T[10] ;
  int i ;
  for (i = 0; i<=9; i++) T[i] = 0 ;
  for (i = 0; i<=9; i++) printf("%d\n", T[i]) ;
}
```

Ou bien :

```
#include <stdio.h>
main()
{
  int T[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, i;
  for (i = 0; i<=9; i++) printf("%d\n", T[i]) ;
}
```

Exercice 2 : (Création, lecture et affichage d'un tableau)
Problème : Ecrire un algorithme permettant de créer un tableau de dix entiers, de lire ses éléments à partir du clavier, ensuite de les afficher. Traduire l'algorithme en C.

Solution :

```
Algorithme lire_tab
Variables
  Entier  T[10] ;
  Entier  i ;
Début
```

```
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(T[i]) ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Ecrire(T[i]) ;
Fin
```

Le programme C :

```
#include <stdio.h>
main()
{
  int T[10] ;
  int i ;
  for (i = 0; i<=9; i++) scanf("%d", &T[i]) ;
  for (i = 0; i<=9; i++) printf("%d\n", T[i]) ;
}
```

Exercice 3 : (Recherche dans un tableau)

Problème : Ecrire un algorithme permettant la recherche d'un nombre lu à partir du clavier dans un tableau de dix réels. Traduire l'algorithme en C.

Solution : Il existe plusieurs stratégies pour la recherche dans un tableau. Nous en verrons deux : la recherche en utilisant la technique de Flag, et la recherche dichotomique.

Technique de Flag : Le Flag, en anglais, veut dire drapeau. Dans cette première méthode, on a une variable booléenne initialisée à `FAUX` (drapeau baissé), et qui aura la valeur `VRAI` dès qu'un événement attendu se produit (drapeau levé). La technique de Flag peut être utilisée dans le problème de recherche et dans n'importe quel problème pareil.

La recherche dichotomique : La dichotomie (« couper en deux » en grec). La recherche dichotomique exige que les éléments du tableau soient ordonnés préalablement. Elle consiste à comparer le nombre à rechercher avec le nombre qui se trouve au milieu du tableau. S'ils sont égaux, on arrête la recherche et on déclare que l'élément existe. Sinon, si le nombre recherché est inférieur, on devra continuer la recherche dorénavant dans la première moitié. Sinon, on devra le rechercher dans la deuxième moitié. A partir de là, on prend la moitié du tableau qui nous reste, et on recommence la recherche. La recherche s'arrête quand il ne reste plus de place là où rechercher, et on déclare que le nombre recherché n'existe pas dans le tableau.

Algorithme de la recherche en utilisant la technique de Flag :

```
Algorithme Recherche_Flag
Variables
  Réel  NOMBRES[10] ;
  Réel  NBR ;
  Booléen  Flag ;
  Entier  i ;
Début
  Ecrire("Entrez les éléments du tableau :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(NOMBRES[i]) ;
  Ecrire("Entrez le nombre à rechercher : ") ;
```

```
  Lire(NBR) ;
  Flag ← FAUX ;
  Pour (i ← 0 ; i <= 9; i ← i+1) Si (NOMBRES[i]=NBR) Flag ← VRAI ;
  Si (Flag) Ecrire(NBR," fait partie du tableau.") ;
  Sinon Ecrire(NBR," ne fait pas partie du tableau.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float NOMBRES[10] ;
  float NBR ;
  int Flag ;
  int i ;
  printf("Entrez les éléments du tableau :\n");
  for(i=0; i<=9; i++) scanf("%f", &NOMBRES[i]) ;
  printf("Entrez le nombre à rechercher : ") ;
  scanf("%f", &NBR) ;
  Flag = 0 ;
  for(i=0; i<=9; i++) if (NOMBRES[i]==NBR) Flag = 1 ;
  if (Flag) printf("%f fait partie du tableau.", NBR) ;
  else printf("%f ne fait pas partie du tableau.", NBR) ;
}
```

L'inconvénient du programme précédent est que la recherche continue jusqu'au dernier élément, même si l'élément recherché a été trouvé auparavant. Un programme optimisé est donc le suivant :


```
#include <stdio.h>
main()
{
  float NOMBRES[10] ;
  float NBR ;
  int Flag ;
  int i ;
  printf("Entrez les éléments du tableau :\n");
  for(i=0; i<=9; i++) scanf("%f", &NOMBRES[i]) ;
  printf("Entrez le nombre à rechercher : ") ;
  scanf("%f", &NBR) ;
  Flag = 0 ;
  i = 0;
  while ((i <= 9)&& (! Flag))
    if (NOMBRES[i]==NBR) Flag = 1 ;
    else i++;
  if (Flag) printf("%f fait partie du tableau.", NBR) ;
  else printf("%f ne fait pas partie du tableau.", NBR) ;
}
```

Algorithme de la recherche dichotomique :

```
Algorithme Recherche_dichotomique
Variables
```

```
  Réel  NOMBRES[10] ;
  Réel  NBR ;
  Booléen  TROUVE ;
  Entier  i, Sup, Inf, Milieu ;
Début
  Ecrire("Entrez les éléments triés du tableau :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(NOMBRES[i]) ;
  Ecrire("Entrez le nombre à rechercher : ") ;
  Lire(NBR) ;
  Sup ← 9 ;
  Inf ← 0 ;
  TROUVE ← FAUX ;
  Tant que ((NON TROUVE) ET (Sup >= Inf)) début
      Milieu ← (Sup + Inf) / 2 ;
      Si (NBR = NOMBRES[Milieu]) TROUVE ← VRAI ;
           Sinon Si (NBR < NOMBRES[Milieu]) Sup ← Milieu – 1 ;
                          Sinon Inf ← Milieu + 1 ;
  Fin
  Si (TROUVE) Ecrire(NBR, " existe dans le tableau.") ;
  Sinon Ecrire(NBR, " n'existe pas dans le tableau.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float NOMBRES[10] ;
  float NBR ;
  int TROUVE ;
  int i, Sup, Inf, Milieu ;
  printf("Entrez les éléments triés du tableau :\n") ;
  for (i=0; i<=9; i++) scanf("%f", &NOMBRES[i]) ;
  printf("Entrez le nombre à rechercher : ") ;
  scanf("%f", &NBR) ;
  Sup = 9 ;
  Inf = 0 ;
  TROUVE = 0 ;
  while ((! TROUVE) && (Sup >= Inf)) {
      Milieu = (Sup + Inf) / 2 ;
      if (NBR == NOMBRES[Milieu]) TROUVE = 1 ;
           else if (NBR < NOMBRES[Milieu]) Sup = Milieu - 1 ;
                          else Inf = Milieu + 1 ;
  }
  if (TROUVE) printf("%f existe dans le tableau.", NBR) ;
      else printf("%f n'existe pas dans le tableau.", NBR) ;
}
```


Remarque : On note que la recherche dichotomique contient la technique de Flag, car on a besoin d'un test d'arrêt au cas où l'élément recherché est trouvé.

2.2. Tri d'un tableau

Problème : Soit un tableau de N entiers rangés n'importe comment. On cherche à modifier le tableau de telle manière que les nombres y soient rangés par ordre croissant.

Plusieurs stratégies peuvent être utilisées pour résoudre le problème de tri :

- Tri par sélection.
- Tri à bulles.
- Tri par insertion.
- Tri rapide (*quick sort*).
- Tri fusion (*merge sort*).
- Tri par tas (*heap sort*).

Dans ce qui suit, nous en verrons deux stratégies : le tri par sélection et le tri à bulles.

Algorithme de tri par sélection :

Dans ce cas, le tri d'un tableau consiste à mettre en bonne position le premier élément, c'est-à-dire le plus petit, puis, on met en bonne position l'élément suivant, et ainsi de suite, jusqu'au dernier.

```
Algorithme Tri_tab_par_selection
Variables
  Entier  T[10] ;
  Entier  x, i, j ;
Début
  Ecrire("Donnez les éléments du tableau :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(T[i]) ;
  Pour (i ← 0 ; i <= 8 ; i ← i+1)
      Pour (j ← i+1 ; j <= 9 ; j ← j+1) Si (T[i] > T[j]) début
                                 x ← T[i] ;
                                 T[i] ← T[j] ;
                                 T[j] ← x ;
                               fin
  Ecrire("Voici les éléments du tableau trié :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Ecrire(T[i]) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{ int  T[10] ;
  int  x, i, j ;
  printf("Donnez les éléments du tableau :\n") ;
  for (i=0; i<=9; i++) scanf("%d", &T[i]) ;
  for (i=0; i<=8; i++)
    for (j=i+1; j<=9; j++) if (T[i] > T[j]) {
                             x = T[i] ;
                             T[i] = T[j] ;
                             T[j] = x ;
                           }
```

```
  printf("Voici les éléments du tableau trié :\n") ;
  for (i=0; i<=9; i++) printf("%d\n", T[i]) ;
}
```

Algorithme de tri à bulles :

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant est un tableau dans lequel tout élément est plus petit que celui qui le suit.

```
Algorithme Tri_tab_bulles
Variables
  Entier  T[10] ;
  Entier  i, x ;
  Booléen  Permut ;
Début
  Ecrire("Donnez les éléments du tableau :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(T[i]) ;
  Répéter
    début
      Permut ← FAUX ;
      Pour (i ← 0 ; i <= 8 ; i ← i+1) Si (T[i] > T[i+1]) début
                                        x ← T[i] ;
                                        T[i] ← T[i+1] ;
                                        T[i+1] ← x ;
                                        Permut ← VRAI ;
                                      fin
      fin
    Jusqu'à (Non Permut) ;
  Ecrire("Voici les éléments du tableau trié :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Ecrire(T[i]);
Fin
```


Programme C :

```
#include <stdio.h>
main()
{
  int  T[10] ;
  int  i, x ;
  int  Permut ;
  printf("Donnez les éléments du tableau :\n") ;
  for(i=0; i<=9; i++) scanf("%d", &T[i]) ;
  do
    {
      Permut = 0 ;
      for(i=0; i<=8; i++) if (T[i] > T[i+1]) {
                                        x = T[i] ;
                                        T[i] = T[i+1] ;
                                        T[i+1] = x ;
                                        Permut = 1 ;
                                      }
      }
```

```
    while (Permut) ;
    printf("Voici les éléments du tableau trié :\n") ;
    for(i=0; i<=9; i++) printf("%d\n", T[i]);
}
```

2.3. Tableau à deux dimensions

Les tableaux vus précédemment sont à une dimension. Un tableau à deux dimensions peut être déclaré de la manière suivante :

```
type_données  Nom_tab[taille_dim1][taille_dim2];
```

Où `taille_dim1` et `taille_dim2` représentent respectivement la taille de la première et la deuxième dimension.

L'exemple suivant permet de déclarer un tableau de réels nommé `Note` à deux dimensions 3 * 5 :

```
Réel  Note[3][5];
```

En C, on écrit : `float Note[3][5];`

Un tableau à deux dimensions peut être représenté par un ensemble de cases organisées en lignes et colonnes et schématisées comme suit :

	0	1	2	3	4
0					
1					
2					

Représentation du tableau `Note` à deux dimensions

Dans ce cas-là, un élément du tableau est accessible par deux indices. Par exemple, si on met `Note[0][0]←0.25` et `Note[1][4]←7.57`, le tableau `Note` devient :

	0	1	2	3	4
0	0.25				
1					7.57
2					

Remarques :

- On peut déclarer et initialiser au même temps un tableau à deux dimensions comme suit :
  ```
  float Note[3][5] = {{0.25, 11, 55, 6, 7.5}, {7, 0, 4, 9.6, 7.57}, {0.62, 61, 0, 2.6, 5}};
  ```
  ou bien :
  ```
  float Note[3][5] = {0.25, 11, 55, 6, 7.5, 7, 0, 4, 9.6, 7.57, 0.62, 61, 0, 2.6, 5};
  ```
- Les tableaux les plus utilisés sont à une ou deux dimensions, mais il est possible de définir des tableaux à trois ou à quatre dimensions, voire plus. Par exemple, `float Note[3][5][2][4];`
- On utilise parfois le terme vecteur pour désigner un tableau à une dimension, et le terme matrice pour désigner un tableau à deux dimensions.
- `Note` contient l'adresse du premier élément, i.e., `Note` est équivalent à `&Note[0][0]`. En plus, les notations suivantes sont équivalentes : `Note[0]` et `&Note[0][0]`, `Note[1]` et `&Note[1][0]`, `Note[2]` et `&Note[2][0]`.

3. Les chaînes de caractères

Une chaîne de caractères est une suite de caractères. Elle est équivalente à un tableau de caractères avec des opérations supplémentaires.

3.1. Déclaration d'une chaîne de caractères

En C, une chaîne de caractères se déclare sous forme d'un tableau de caractères de longueur fixe. Dans ce tableau, le caractère `'\0'`, correspondant à la valeur `0`, indique la fin de la chaîne de caractères.

Ainsi : `char ch[15]` permet d'enregistrer une chaîne de 14 caractères maximum (15- 1 pour le 0 indiquant la fin de la chaîne).

En C, avec `char ch[15] = "bonjour"`, la chaîne de caractères `ch` peut être représentée de la manière suivante :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	o	n	j	o	u	R	\0							

Ceci est équivalent à : `char ch[15] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};`

Remarques :

- Dans d'autres langages, tels que Pascal et Java, une chaîne de caractères est déclarée par le mot clé `String`.
- En C, une chaîne constante doit être mise entre deux guillemets doubles pour la distinguer d'un identificateur de variable, comme déjà vu pour la chaîne `"bonjour"`.
- Comme signalé auparavant, si vous dépassez la longueur fixée du tableau, vous écrivez dans des cases qui suivent le tableau, et qui peuvent être réservées pour d'autres variables. Si vous affectez une valeur qui dépasse la longueur prévue de la chaîne de caractères, le compilateur ne déclare pas une erreur de compilation, mais vous aurez pu avoir un résultat inattendu. Donc, évitez cette situation.
- Il est possible de déclarer une chaîne de caractères sans spécifier la longueur de départ de la façon suivante : `char ch2[] = "Ahmed"`. De cette façon, la chaîne fera exactement la longueur nécessaire pour stocker `Ahmed` et le `0` final. Elle aura donc la taille 5+1=6 octets.

3.2. Manipulation des chaînes de caractères

Affichage :

Une chaîne de caractères s'affiche grâce à la fonction `printf` et le format `%s`.

Ainsi, `printf("%s", ch)` affiche : `bonjour`

Remarques :

- `printf` appartient à la bibliothèque `stdio.h`. Si on utilise `printf` sans spécifier la bibliothèque `stdio.h`, le compilateur C signale un [Warning].
- Rappelons que `puts(ch)` est équivalente à `printf("%s\n", ch)`.

- La fonction `puts`, elle aussi, appartient à la bibliothèque `stdio.h`.

Accès aux caractères :

Chaque caractère peut être manipulé en utilisant son indice. Les indices valides pour une chaîne de caractères sont des entiers compris entre 0 et la longueur de la chaîne-1.

Ainsi, `printf("%c",ch[2])` affiche : `n`

Longueur d'une chaîne de caractères :

La longueur d'une chaîne de caractères s'obtient par la fonction `strlen` (appartenant à la bibliothèque `string.h`). Le `0` de fin de chaîne n'est pas compté dans cette longueur.

Ainsi, `printf("%d", strlen(ch))` affiche : `7`

Affectation :

Pour affecter une valeur à une variable chaîne de caractères, il est possible d'utiliser la fonction `strcpy(<s>,<t>)`. Cette fonction nous impose d'ajouter le fichier d'en-tête `string.h` :

Ainsi, `strcpy(ch, "Ali")` affecte la valeur `"Ali"` à la chaîne `ch`.

Remarques :

- La chaîne `<s>` doit être suffisamment grande pour accueillir `<t>`. Si on ne respecte pas cette règle, la réaction du programme dépond du compilateur C utilisé.
- L'affectation `ch="Ali"` n'est pas acceptée par le compilateur C, car le membre de gauche désigne une adresse alors que le membre de droite désigne une chaîne de caractères.

La fonction strncpy :

La fonction `strncpy(<s>,<t>,n)` fait le même travail que `strcpy`, sauf qu'elle se limite au `n` premiers caractères de `<t>`. Si `n` dépasse la longueur de `<t>`, alors le `'\0'` de `<t>` sera aussi copier dans `<s>`, et `strncpy` devient équivalente à `strcpy`.

Ainsi, `strncpy(ch,st,2)` remplace les deux premiers caractères de `ch` par les deux premiers de `st`.

Lecture :

Si l'on souhaite lire une chaîne directement au clavier, on peut utiliser la fonction `scanf` :

Ainsi, `scanf("%s", ch)` permet de lire la chaîne de caractères `ch` à partir du clavier.

Remarques :

- Nous n'avons pas mis `&` avant `ch` dans `scanf("%s", ch)`, car `ch` lui-même représente l'adresse du premier caractère de la chaîne.

- `scanf("%s",&ch[0])` ferait la même chose que `scanf("%s", ch)`.
- `scanf` appartient à la bibliothèque `stdio.h`. Si on utilise `scanf` sans spécifier la bibliothèque `stdio.h`, le compilateur C signale un [Warning].
- En utilisant `scanf`, la chaîne de caractères saisie ne peut comporter ni espaces, ni tabulations.

La fonction gets :

La fonction `gets` permet de lire une chaîne de caractères validée par la touche *Entrée*. Attention ! La touche *Entrée* elle-même n'est pas enregistrée dans le tableau de caractères.

Ainsi, `gets(ch)` permet de lire la chaîne de caractères `ch` à partir du clavier.

Remarques :

- A la différence de `scanf`, la fonction `gets` permet de lire une chaîne de caractères contenant des espaces et des tabulations. Elle permet même de lire une chaîne vide. Il est donc recommandé d'utiliser `gets` au lieu de `scanf` pour lire les chaînes de caractères à partir du clavier.
- La fonction `gets`, elle aussi, appartient à la bibliothèque `stdio.h`.

La fonction strcat :

La fonction `strcat(<s>,<t>)` de la bibliothèque `string.h` ajoute la chaîne `<t>` à la chaîne `<s>` (on appelle cela une concaténation). La taille de `<s>` doit être suffisamment grande pour contenir aussi le contenu de `<t>`.

Ainsi, `strcat(ch,st)` ajoute `st` à `ch`.

On peut dire aussi que `strcat(ch,st)` retourne la concaténation des deux chaînes `ch` et `st`.

La fonction strncat :

La fonction `strncat(<s>,<t>,n)` ajoute les `n` premiers caractères de la chaîne `<t>` à la fin de la chaîne `<s>`.

Ainsi, `strncat(ch,st,2)` ajoute les deux premiers caractères de `st` à la fin de la chaîne `ch`.

La fonction strcmp :

La fonction `strcmp(<s>,<t>)` de la bibliothèque `string.h` compare les chaînes de caractères `<s>` et `<t>` de manière lexicographique et fournit un résultat :

- nul (`0`) si `<s>` est égale à `<t>`.
- négatif (`-1`) si `<s>` précède `<t>`.
- positif (`+1`) si `<s>` suit `<t>`.

Ainsi, `strcmp("AA","AA")` renvoit `0`, `strcmp("AA","BB")` renvoit `-1`, et `strcmp("BB","AA")` renvoit `+1`.

Remarques :

- `strncmp(<s>,<t>, n)` limite la comparaison aux `n` premiers caractères des deux chaînes.
- `stricmp(<s>,<t>)` compare les deux chaînes sans distinction entre majuscules et minuscules. Les deux chaînes sont comparées comme si elles étaient introduites en minuscules.
- `strnicmp(<s>,<t>, n)` combine les deux fonctions `strncmp` et `stricmp`.

Les fonctions sprintf et sscanf :

La fonction :
`sprintf(<chaîne cible>,<chaîne de formatage>,<expr1>,<expr2>,…)`
convertit un ou plusieurs nombres en une chaîne de caractères.

Ainsi, `int i = sprintf(ch,"%d",15)` permet de convertir l'entier `15` en chaîne de caractères et la stocker dans la variable `ch`. À l'arrivée, `ch` contiendra la chaîne de caractères `"15"`, et `i` contiendra la longueur de cette chaîne (`2`).

La fonction `sscanf` fait le contraire.

Ainsi, pour `int i = sscanf("12.5 12.3 11.6","%f%f%f",&a,&b,&c)`, les variables réelles `a`, `b` et `c` contiendront respectivement `12.5`, `12.3` et `11.6`, et `i` contiendra `3` (le nombre de variables remplies avec succès).

Exemple :

```
#include <stdio.h>
#include<string.h>
main()
{
  char ch[15] = "bonjour";
  char st[] = "Ahmed";
  printf("%s\n", ch);
  printf("%c\n", ch[4]);
  ch[4]='\0';
  printf("%s\n", ch);
  printf("%d\n", strlen(ch));
  strcpy(ch, "Ali");
  printf("%s\n", ch);
  strcat(ch,st);
  printf("%s\n", ch);
  strncpy(ch,st,2);
  printf("%s\n", ch);
  strncat(ch,st,2);
  printf("%s\n", ch);
  printf("%d\n", strcmp(ch,st));
  int i = sprintf(ch,"%d",15);
  printf("%s ** %d\n", ch, i);
  float a,b,c;
  i = sscanf("12.5 12.3 11.6","%f%f%f",&a, &b, &c);
  printf("%f ** %f ** %f\n", a, b, c);
}
```

Le programme C précédent affiche :

```
bonjour
o
bonj
4
Ali
AliAhmed
AhiAhmed
AhiAhmedAh
-1
15 ** 2
12.500000 ** 12.300000 ** 11.600000
```

3.3. Tableau de chaînes de caractères

On peut aussi déclarer un tableau de chaînes de caractères qui est un tableau de caractères à deux dimensions. Pour déclarer, par exemple, un tableau de 5 chaînes de 19 caractères maximum, on écrit : `char tab[5][20]`. On accède à chacune des chaînes en utilisant l'élément `tab[i]` du tableau, et on accède à n'importe quel caractère *j* de la chaîne *i* par `tab[i][j]`.

4. Exercices corrigés

4.1. Exercices

Exercice 1 :

Soit l'algorithme :

```
Algorithme X
Variables
  Entier Suite[7] ;   Entier i ;
Début
  Suite[0] ← 0 ;  Suite[1] ← 1 ;
  Pour (i ← 2 ; i<= 6 ; i ← i+1)
        Suite[i] ← Suite[i-1] + Suite[i-2] ;
Fin
```

Représentez le tableau `Suite` après l'exécution des opérations de l'algorithme `X`.

Exercice 2 :

Ecrire un algorithme, ensuite le programme C, permettant de calculer la somme, le produit et la moyenne d'un tableau de dix réels.

Exercice 3 :

Ecrire un algorithme qui lit un tableau de 10 entiers, et affiche ensuite les éléments d'indice impair. Traduire l'algorithme en C.

Exercice 4 :

Ecrire un algorithme, ensuite le programme C, permettant de déterminer la valeur maximale et la valeur minimale dans un tableau de dix entiers, avec leurs positions dans le tableau.

Exercice 5 :
Ecrire un algorithme permettant l'affichage des deux plus petits éléments d'un tableau de dix entiers. Traduire l'algorithme en C.

Exercice 6 :
Ecrire un algorithme permettant de remplir deux tableaux, chacun contient 10 entiers, de stocker la somme de ces deux tableaux dans un troisième tableau, et d'afficher ce dernier. Traduire l'algorithme en C.

Note : Chaque élément du troisième tableau est calculé par l'addition des deux éléments des tableaux 1 et 2 ayant la même position, comme le montre l'exemple suivant :
Si Tableau 1 contient : `9, 2, 0, 7, 4, -1, 7, 4, 9, 1,`
et Tableau 2 contient : `4, 2, 8, 9, -4, 1, 6, 6, 3, 7,`
alors Tableau 3 va contenir : `13, 4, 8, 16, 0, 0, 13, 10, 12, 8.`

Exercice 7 :
Ecrire un algorithme permettant de lire un tableau de 7 entiers et de dire si les éléments du tableau sont tous consécutifs ou non. Traduire l'algorithme en C.

Note : Les nombres consécutifs sont des nombres entiers qui se suivent, comme par exemple `21, 22, 23, 24, 25, 26` et `27`.

Exercice 8 :
Ecrire un algorithme qui calcule itérativement le $n^{ième}$ terme de la suite de Fibonacci définie comme suit : Si $n = 0$ alors $F_n = 0$; Si $n = 1$ alors $F_n = 1$; Si $n > 1$ alors $F_n = F_{n-1}+F_{n-2}$. Utilisez cette fois-ci un tableau.
Traduire ensuite l'algorithme en C.

Exercice 9 :
Ecrire un algorithme qui supprime un élément dont la position est lue à partir du clavier dans un tableau de dix entiers. L'élément supprimé sera substitué par un zéro ajouté à la fin du tableau. Traduire l'algorithme en C.

Exercice 10 :
Ecrire un algorithme déterminant les *k* premiers *nombres premiers*. La valeur de *k* étant une constante. On conservera les *nombres premiers* dans un tableau, au fur et à mesure de leur découverte. Enfin, on affiche les *k* premiers *nombres premiers* à partir du tableau. Traduire l'algorithme en C.
Note : Un *nombre premier* est un entier positif qui admet exactement deux diviseurs distincts entiers et positifs. Ces deux diviseurs sont 1 et le nombre considéré. Les nombres 0 et 1 ne sont pas premiers. Voici, par exemple, les dix premiers *nombres premiers* : `2, 3, 5, 7, 11, 13, 17, 19, 23` et `29`.

Exercice 11 :
Ecrire un algorithme permettant de créer un tableau d'entiers à deux dimensions (3*5), d'initialiser ses éléments à zéro, ensuite de les afficher. Traduire l'algorithme en C.

Exercice 12 :

Soit l'algorithme :

```
Algorithme XX
Variables
  Entier T[4][2] ;
  Entier k,m ;
Début
  Pour (k ← 0 ; k<= 3 ; k ← k +1)
       Pour (m ← 0 ; m <= 1 ; m ← m+1) T[k][m] ← k+m ;
Fin
```

Représentez le tableau `T` après l'exécution des opérations de l'algorithme `XX`.

Exercice 13 :

Ecrire un algorithme permettant de remplir un tableau d'entiers à deux dimensions (3*5) à partir du clavier, et de déterminer le nombre d'apparition d'une certaine valeur saisie par l'utilisateur. Traduire l'algorithme en C.

Exercice 14 :

Ecrire un algorithme qui calcule la transposée d'une matrice d'entiers (3*3). Traduire l'algorithme en C.

Note : La transposée d'une matrice est une nouvelle matrice dont les lignes sont les colonnes de la matrice initiale et les colonnes sont les lignes de la matrice initiale, comme le montre l'exemple suivant :

La transposée de la matrice :

7	2	-1
4	14	6
0	1	2

est :

7	4	0
2	14	1
-1	6	2

Exercice 15 :

Ecrire un algorithme qui calcule le produit matriciel de deux matrices (3*3). Le résultat sera rangé dans une troisième matrice. Traduire l'algorithme en C.

Note : Le produit matriciel de deux matrices est une troisième matrice là où chaque élément *(x,y)* est calculer par la multiplication de la x^{ieme} ligne de la première matrice par la $y^{ième}$ colonne de la deuxième matrice, comme le montre l'exemple suivant :

1	6	-1	*	2	0	7	=	13	6	11
2	1	4		2	1	1		10	1	23
0	0	1		1	0	2		1	0	2

Exercices 16 :

En utilisant une boucle `for`, écrire un programme C qui remplit un tableau de 10+1 caractères avec les lettres de l'alphabet en commençant par `'A'` (code ASCII `65`). Le tableau devra donc contenir ceci :

0	1	2	3	4	5	6	7	8	9	10
'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	0

Dans le même programme :

- Faites afficher la chaîne de caractères ainsi obtenue ;

- Faites afficher chaque caractère du tableau sous la forme :
  ```
  Caractère numéro 0 : A.
  Caractère numéro 1 : B.
  …
  ```

Exercice 17 :

Qu'affiche le programme C suivant si vous saisissez au clavier les deux mots : `"science"` et `"informatique"` ?

```
#include <stdio.h>
#include <string.h>
main () {
   char chaine1[81]; /* 80 caractères + '\0' */
   char chaine2[81];
   printf("Veuillez entrer votre chaîne de caractères : ");
   gets(chaine1) ; /* identique à scanf("%s",chaine1);
                      ou scanf("%s",&chaine1[0]); */
   strcpy(chaine2,chaine1);  /* attention à l'ordre !!! */
   printf("chaîne2 vaut : %s\n",chaine2);
   strcpy(chaine1,"");   /* et non pas chaine1="";  !!!!!!! */
   strcat(chaine1,"Ahmed ");
   printf("Veuillez entrer votre chaîne de caractères : ");
   gets(chaine2);
   strcat(chaine1,chaine2);
   strcat(chaine1," Mohamed Ali...");
   printf("chaîne1 vaut : %s \n",chaine1);
}
```


Exercice 18 :

Ecrire un programme C permettant de lire deux chaînes de caractères et de les afficher par ordre alphabétique.

Exercice 19 :

Ecrire un programme C permettant de lire une chaîne de 19 caractères maximum et de calculer le nombre d'apparition d'un caractère lu à partir du clavier.

Exercice 20 :

Ecrire un programme C qui supprime toutes les lettres `'e'` d'une chaîne de caractères. La chaîne ainsi modifiée doit être affichée.

Exercice 21 :

Ecrire un programme C qui lit au clavier une chaîne d'au plus 29 caractères et qui l'affiche à l'envers.

Exercice 22 :

Ecrire un programme C qui calcule la somme des codes ASCII des caractères d'une chaîne lue à partir du clavier.

Exercice 23 :

Ecrire un programme C qui lit un verbe du premier groupe et qui en affiche la conjugaison au présent de l'indicatif. On admettra que l'utilisateur ne fournit

que des verbes réguliers (se terminant par `"er"`), et qu'il ne fournira pas des verbes tels que `"manger"`.
Tester votre programme avec le verbe `"sauter"`.

Exercice 24 :
Ecrire un programme C qui lit un tableau de 5 mots de 19 caractères maximum, puis affiche à nouveau le tableau avec son contenu en majuscules, et trié par ordre alphabétique.

Note : En langage C, la fonction de conversion d'un caractère en majuscule est `toupper(code ASCII du caractère)`. La fonction de conversion d'un caractère en minuscule est `tolower(code ASCII du caractère)`. Toutes les deux appartiennent à la bibliothèque `ctype.h`.

Exercice 25 :
Ecrire un programme C permettant de lire un tableau de 10 chaînes de 4 caractères maximum, puis il affiche le dernier caractère de chaque chaîne.

4.2. Corrigés

Solution 1 :
Cet algorithme crée un tableau de 7 éléments de type entier. Ensuite, il remplit le tableau par 7 valeurs : `0`, `1`, `1`, `2`, `3`, `5` et `8`.

Le tableau `Suite` peut être représenté donc comme suit :

0	0
1	1
2	1
3	2
4	3
5	5
6	8

Solution 2 :
```
Algorithme Som_Moy_Produit_tab
Variables
  Réel  T[10] ;
  Entier  i ;
  Réel  somme, produit, moyenne ;
Début
  Ecrire("Donnez les éléments du tableau :");
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(T[i]) ;
  somme ← 0;
  produit ← 1 ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) début
    somme ← somme + T[i] ;
```

```
    produit ← produit * T[i] ;
  fin
  moyenne ← somme / 10 ;
  Ecrire("La somme = ", somme) ;
  Ecrire("Le produit = ", produit) ;
  Ecrire("La moyenne = ", moyenne) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  float T [10] ;
  int i ;
  float somme, produit, moyenne ;
  printf("Donnez les éléments du tableau :\n");
  for (i=0; i<=9; i++) scanf("%f", &T[i]) ;
  somme = 0;
  produit = 1 ;
  for (i=0; i<=9; i++) {
    somme = somme + T[i] ;
    produit = produit * T[i] ;
  }
  moyenne = somme / 10 ;
  printf("La somme = %f\n", somme) ;
  printf("Le produit = %f\n", produit) ;
  printf("La moyenne = %f", moyenne) ;
}
```


Solution 3 :

Algorithme :

```
Algorithme elements_impairs
Variables
  Entier T[10] ;
  Entier i ;
Début
  Ecrire("Donnez les éléments du tableau :") ;
  Pour (i ← 0; i<= 9; i ← i+1) Lire(T[i]) ;
  Ecrire("Voici les éléments dont l'indice est impair :") ;
  Pour (i ← 0; i<= 9; i ← i+1) Si (i%2 = 1) Ecrire(T[i]) ;
Fin
```

Programme C :

```
#include <stdio.h>
main () {
  int T[10] ;
  int i ;
  puts("Donnez les éléments du tableau :") ;
  for(i=0;i<=9;i++) scanf("%d", &T[i]) ;
  puts("Voici les éléments dont l'indice est impair :") ;
  for(i=0;i<=9;i++) if (i%2 ==1) printf("%d\n", T[i]) ;
```

```
}
```

Solution 4 :

```
Algorithme MaxMin_tab
Variables
  Entier  T[10] ;
  Entier  max, min, pos_min, pos_max, i ;
Début
  Ecrire("Donnez les éléments du tableau :") ;
  Pour (i ← 0 ; i <= 9 ; i ← i+1) Lire(T[i]) ;
  max ← T[0] ;
  pos_max ← 0 ;
  Pour (i ← 1 ; i <= 9 ; i ← i+1) Si (max < T[i]) début
                                        max ← T[i] ;
                                        pos_max ← i ;
                                  fin
  min ← T[0] ;
  pos_min ← 0 ;
  Pour (i ← 1 ; i <= 9 ; i ← i+1) Si (min > T[i]) début
                                        min ← T[i] ;
                                        pos_min ← i ;
                                  fin
  Ecrire("La valeur maximale = ", max, " occupant la position ", pos_max) ;
  Ecrire("La valeur minimale = ", min, " occupant la position ", pos_min) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[10] ;
  int max, min, pos_min, pos_max, i ;
  printf("Donnez les éléments du tableau :\n") ;
  for (i=0; i<=9; i++) scanf("%d", &T[i]) ;
  max = T[0] ;
  pos_max = 0 ;
  for(i=1; i<=9; i++) if (max < T[i]) {
                                       max = T[i] ;
                                       pos_max = i ;
                                      }
  min = T[0] ;
  pos_min = 0 ;
  for (i=1; i<=9; i++ ) if (min > T[i]) {
                                         min = T[i] ;
                                         pos_min = i ;
                                        }
  printf("La valeur maximale = %d occupant la position %d\n", max, pos_max) ;
  printf("La valeur minimale = %d occupant la position %d", min, pos_min) ;
}
```

Solution 5 :

Algorithme :

```
Algorithme Min2_tab
Variables
  Entier T[10];
  Entier min1, min2, i ;
Début
  Ecrire("Donnez les éléments du tableau :") ;
  Pour (i ← 0; i<= 9; i ← i+1) Lire(T[i]) ;
  min1 ← T[0] ;
  min2 ← T[1] ;
  Pour (i ← 2 ; i<= 9 ; i ← i+1)
    Si (min1 > min2) début Si (min1 > T[i]) min1 ← T[i];  fin
      Sinon Si (min2 > T[i]) min2 ← T[i] ;
  Ecrire("Les deux valeurs minimales sont :") ;
  Si (min1 <min2) Ecrire(min1, "  ", min2) ;
    Sinon Ecrire(min2, "  ", min1) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[10];
  int min1, min2, i ;
  printf("Donnez les éléments du tableau :\n") ;
  for(i=0; i<=9; i++) scanf("%d", &T[i]) ;
  min1 = T[0] ;
  min2 = T[1] ;
  for(i=2; i<=9; i++)
    if (min1 > min2) { if (min1 > T[i]) min1 = T[i];  }
      else if (min2 > T[i]) min2 = T[i] ;
  printf("Les deux valeurs minimales sont :\n") ;
  if (min1 <min2) printf("%d  %d", min1, min2) ;
    else printf("%d  %d", min2, min1) ;
}
```


Solution 6 :

Algorithme :

```
Algorithme Somme_tab
Variables
  Entier T1[10], T2[10], T3[10] ;
  Entier i ;
Début
  Ecrire("Donnez les éléments du 1ier tableau :") ;
  Pour (i ← 0; i<= 9; i ← i+1) Lire (T1[i]) ;
  Ecrire("Donnez les éléments du 2ième tableau :") ;
  Pour (i ← 0; i<= 9; i ← i+1) Lire (T2[i]) ;
  Pour (i ← 0; i<= 9; i ← i+1) T3[i] ← T1[i] + T2[i] ;
  Ecrire("Voici le tableau Somme :") ;
```

```
  Pour (i ← 0; i<= 9; i ← i+1) Ecrire(T3[i]) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T1[10], T2[10], T3[10] ;
  int i ;
  printf("Donnez les éléments du 1ier tableau :\n") ;
  for (i=0; i<=9; i++) scanf("%d", &T1[i]) ;
  printf("Donnez les éléments du 2ième tableau :\n") ;
  for (i=0; i<=9; i++) scanf("%d", &T2[i]) ;
  for (i=0; i<=9; i++) T3[i] = T1[i] + T2[i] ;
  printf("Voici le tableau Somme :\n") ;
  for (i=0; i<=9; i++) printf("%d\n", T3[i]) ;
}
```

Solution 7 :

Algorithme :

```
Algorithme Nombres_consecutifs
Variables
  Entier T[7] ;
  Entier i ;
  Booléen Cons ;
Début
  Ecrire("Entrez les valeurs :") ;
  Pour (i ← 0 ; i<= 6 ; i ← i+1) début
    Ecrire("Entrez le nombre n° ", i, " : ") ;
    Lire(T[i]) ;
  fin
  Cons ← VRAI ;
  Pour (i ← 0 ; i<=5 ; i ← i+1)
      Si (T[i] <> (T[i + 1] - 1)) Cons ← FAUX ;
  Si (Cons) Ecrire("Les nombres sont consécutifs.") ;
    Sinon Ecrire("Les nombres ne sont pas consécutifs.") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[7] ;
  int i ;
  int Cons ;
  printf("Entrez les valeurs :\n") ;
  for(i=0; i<=6; i++) {
    printf("Entrez le nombre n° %d : ", i) ;
    scanf("%d", &T[i]) ;
  }
  Cons = 1 ;
```

```
  for(i=0; i<=5; i++)
      if (T[i] != (T[i + 1] - 1)) Cons = 0 ;
  if (Cons) printf("Les nombres sont consécutifs.") ;
    else printf("Les nombres ne sont pas consécutifs.") ;
}
```

Cette solution est sans doute la plus spontanée, mais elle présente le défaut d'examiner la totalité du tableau, même lorsqu'on découvre dès le départ deux éléments non consécutifs. Aussi, dans le cas d'un grand tableau, cette solution est coûteuse en temps de traitement. Une autre manière de procéder serait de sortir de la boucle dès que deux éléments non consécutifs sont détectés, et ceci en remplaçant la deuxième boucle `Pour` par la boucle `Tant que` suivante :

```
  i ← 0 ;
  Tant que ((Cons) ET (i <=5))
    Si (T[i] <> T[i + 1] - 1) Cons ← FAUX ;
    Sinon i ← i + 1 ;
```

En C :

```
  i = 0 ;
  while ((Cons) && (i <=5))
    if (T[i] != T[i + 1] - 1) Cons = 0 ;
    else i = i + 1 ;
```

Solution 8 :

Algorithme :

```
Algorithme Fibonacci
Constantes
  max = 10 ;
Variables
  Entier fibo[max] ;
  Entier n, i ;
Début
  Ecrire("Donnez un entier : ") ;
  Lire(n) ;
  Si (n<=max) début
    fibo[0] ← 0;
    fibo[1] ← 1;
    Pour (i ← 2; i<= n; i ← i+1)
         fibo[i] ← fibo[i-1] + fibo[i-2];
    Ecrire("Fibonacci(", n, ") = ", fibo[n]) ;
  fin
Fin
```

Programme C :

```
#include <stdio.h>
#define max 10
main()
{
  int fibo[max] ;
  int n, i ;
  printf("Donnez un entier : ") ;
```

```
  scanf("%d", &n) ;
  if (n<=max) {
    fibo[0] = 0;
    fibo[1] = 1;
    for (i=2; i<=max-1; i++) fibo[i] = fibo[i-1] + fibo[i-2];
    printf("Fibonacci(%d) = %d", n, fibo[n]) ;
  }
}
```

Solution 9 :

Algorithme :

```
Algorithme Supprimer_element_tab
Variables
  Entier T[10] ;
  Entier i, j ;
Début
  Ecrire("Saisir les éléments d'un tableau de dix entiers :");
  Pour (i ← 0; i<= 9; i ← i+1) Lire(T[i]);
  Ecrire("Donnez la position de l'élément à supprimer (prise entre 0 et 9) : ");
  Lire(j);
  Pour (i ← j ; i<= 9 ; i ← i+1) T[i] ← T[i+1];
  T[9] ← 0;
  Ecrire("Les éléments du tableau après la suppression de l'élément num ", j);
  Pour (i ← 0; i<= 9; i ← i+1) Ecrire(T[i]);
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[10] ;
  int i, j ;
  printf("Saisir les éléments d'un tableau de dix entiers :\n");
  for(i=0;i<=9;i++) scanf("%d", &T[i]);
  printf("Donnez la position de l'élément à supprimer (prise entre 0 et 9) : ");
  scanf("%d", &j);
  for(i=j;i<=8;i++) T[i] = T[i+1];
  T[9] = 0;
  printf("Les éléments du tableau après la suppression de l'élément num %d\n", j);
  for(i=0;i<=9;i++) printf("%d\n", T[i]);
}
```

Solution 10 :

Algorithme :

```
Algorithme k_premiers
Constantes k = 10 ;
Variables
  Entier T[k] ;
  Entier i, nb, ind;
  Booléen Premier ;
Début
```

```
  ind ← 0 ;
  nb ← 2 ;
  Tant que (ind < k) début
    Premier ← VRAI ;
    i ← 2 ;
    Tant que ((i<nb) ET Premier)
        Si (nb % i <> 0) Premier ← FAUX ;
          Sinon i ← i + 1 ;
    Si (Premier) début
                T[ind] ← nb ;
                ind ← ind + 1 ;
    fin
    nb ← nb + 1 ;
  fin
  Ecrire("Les ", k, " premiers nombres premiers sont :") ;
  Pour (i ← 0 ; i<= k-1 ; i ← i+1) Ecrire(T[i]) ;
Fin
```

Programme C :

```
#include <stdio.h>
#define k 10
main () {
  int T[k] ;
  int i, nb, ind;
  int Premier ;
  ind = 0 ;
  nb = 2 ;
  while (ind < k) {
    Premier = 1 ;
    i = 2 ;
    while ((i<nb) && Premier)
        if (nb % i == 0) Premier = 0 ;
          else i = i + 1 ;
    if (Premier) {
                T[ind] = nb ;
                ind = ind + 1 ;
                }
    nb = nb + 1 ;
  }
  printf("Les %d premiers nombres premiers sont :\n", k) ;
  for (i=0; i<=k-1; i++) printf("%d\n", T[i]) ;
}
```


Solution 11 :

Algorithme :

```
Algorithme Init_matrice
Variables
  Entier T[3][5] ;
  Entier i, j ;
Début
  Pour (i ← 0 ; i<= 2 ; i ← i+1)
```

```
      Pour (j ← 0 ; j<= 4 ; j ← j+1) T[i][j]← 0 ;
  Ecrire("Voici la matrice initialisée :");
  Pour (i ← 0 ; i<= 2 ; i ← i+1)
      Pour (j ← 0 ; j<= 4 ; j ← j+1) Ecrire(T[i][j], " ") ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[3][5] ;
  int i, j ;
  for(i=0;i<=2;i++)
      for(j=0;j<=4;j++) T[i][j]= 0 ;
  puts("Voici la matrice initialisée :");
  for(i=0;i<=2;i++) {
      for(j=0;j<=4;j++)  printf("%d  ", T[i][j]) ;
      printf("\n");
  }
}
```

Solution 12 :

Cet algorithme remplit un tableau d'entiers à deux dimensions 4*2, de la manière suivante :

```
T[0,0] = 0 ; T[0,1] = 1 ; T[1,0] = 1 ; T[1,1] = 2 ; T[2,0] =
2 ; T[2,1] = 3 ; T[3,0] = 3 ; T[3,1] = 4 ;
```

Le tableau `T` peut être représenté donc comme suit :

	0	1
0	0	1
1	1	2
2	2	3
3	3	4

Solution 13 :

Algorithme :

```
Algorithme Nbr_app
Variables
  Entier T[3][5] ;
  Entier i, j, x, nb ;
Début
  Ecrire("Donnez les valeurs de la matrice :") ;
  Pour (i ← 0 ; i<=2 ; i ← i+1)
       Pour (j ← 0 ; j<=4 ; j ← j+1) Lire(T[i][j]) ;
  Ecrire ("Donnez la valeur dont vous désirez calculer le nombre d'apparition : ") ;
  Lire (x) ;
  nb ← 0 ;
  Pour (i ← 0 ; i<=2 ; i ← i+1)
       Pour (j ← 0 ; j<=4 ; j ← j+1)
            Si (x = T[i][j]) Alors nb ← nb + 1 ;
  Ecrire("Le nombre d'apparition de la valeur ", x, " dans la matrice est = ", nb) ;
```

```
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[3][5] ;
  int i, j, x, nb ;
  printf("Donnez les valeurs de la matrice :\n") ;
  for(i=0;i<=2;i++) for(j=0;j<=4;j++) scanf("%d", &T[i][j]) ;
  printf("Donnez la valeur dont vous désirez calculer le nombre d'apparition : ") ;
  scanf("%d", &x) ;
  nb = 0 ;
  for(i=0;i<=2;i++)
     for(j=0;j<=4;j++) if (x == T[i][j]) nb = nb + 1 ;
  printf("Le nombre d'apparition de la valeur %d dans la matrice est = %d", x, nb) ;
}
```

Solution 14 :

Algorithme :

```
Algorithme Transposee_matrice
Variables
  Entier T[3][3] ;
  Entier i, j, x ;
Début
 Ecrire("Donnez les éléments de la matrice :") ;
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Lire(T[i][j]) ;
 Ecrire("Matrice initiale :");
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Ecrire(T[i][j]);
 Pour (i ← 0; i<= 1; i ← i+1)
   Pour (j ← i+1; j<= 2; j ← j+1) début
      x ← T[i][j] ;
      T[i][j] ← T[j][i] ;
      T[j][i] ← x ;
   fin
 Ecrire("Matrice transposée :");
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Ecrire(T[i][j]) ;
Fin
```

Programme C :

```
#include <stdio.h>
main()
{
  int T[3][3] ;
  int i, j, x ;
  printf("Donnez les éléments de la matrice :\n") ;
  for(i=0;i<=2;i++) for(j=0;j<=2;j++) scanf("%d", &T[i][j]) ;
  printf("Matrice initiale : \n");
```

```
  for(i=0;i<=2;i++){
  for(j=0;j<=2;j++) printf("%d  ", T[i][j]);
  printf("\n");
  }
  for(i=0;i<=1;i++) for(j=i+1;j<=2;j++)
  {
      x = T[i][j] ;
      T[i][j] = T[j][i] ;
      T[j][i] = x ;
  }
  printf("Matrice transposée : \n");
  for(i=0;i<=2;i++){
  for(j=0;j<=2;j++) printf("%d  ", T[i][j]);
  printf("\n");
  }
}
```

Solution 15 :

Algorithme :

```
Algorithme produit_matriciel
Variables
  Entier mat1, mat2, produit[3][3] ;
  Entier i, j, k, x, s ;
Début
 Ecrire("Donnez les éléments de la première matrice :") ;
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Lire(mat1[i][j]);
 Ecrire("Donnez les éléments de la deuxième matrice :") ;
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Lire(mat2[i][j]);
 Pour (i ← 0; i<= 2; i ← i+1)
   Pour (j ← 0; j<= 2; j ← j+1) début
    s ← 0 ;
    Pour (k ← 0; k<=2; k ← k+1) s ← s+mat1[i][k] * mat2[k][j];
    produit[i][j] ← s ;
   fin
 Ecrire("Résultat du produit matriciel :") ;
 Pour (i ← 0; i<= 2; i ← i+1)
      Pour (j ← 0; j<= 2; j ← j+1) Ecrire(produit[i][j]) ;
Fin.
```

Programme C :

```
#include <stdio.h>
main()
{
  int mat1[3][3], mat2[3][3], produit[3][3] ;
  int i, j, k, x, s ;
  printf("Donnez les éléments de la première matrice :\n") ;
  for(i=0;i<=2;i++)
     for(j=0;j<=2;j++) scanf("%d", &mat1[i][j]);
  printf("Donnez les éléments de la deuxième matrice :\n") ;
```

```
  for(i=0;i<=2;i++)
     for(j=0;j<=2;j++) scanf("%d", &mat2[i][j]);
  for(i=0;i<=2;i++) for(j=0;j<=2;j++) {
      s = 0 ;
      for(k=0;k<=2;k++) s = s + mat1[i][k] * mat2[k][j] ;
      produit[i][j] = s ;
    }
  printf("Résultat du produit matriciel :\n") ;
  for(i=0;i<=2;i++)
     for(j=0;j<=2;j++) printf("%d\n", produit[i][j]) ;
}
```

Solution 16 :

Programme C :

```
#include <stdio.h>
#include <stdlib.h>
main () {
   /* 10 caractères + 0 de fin de chaîne */
   char tab [11];
   int i=0; /* compteur */
   /* Remplissage du tableau avec les caractères */
   for (i=0; i<=9; i++)
      tab[i] = 65 + i; // ou tab[i]='A'+i;
   /* Ajout du 0 de fin de chaîne */
   tab[10] = 0;
   /* Affichage de la chaîne */
   printf("tab : %s\n",tab);
   /* Saut d'une autre ligne */
   printf("\n");
   /* Affichage de chacun des caractères */
   for (i=0; i<10; i++)
       printf("Caractère numéro %d : %c\n",i,tab [i]);
}
```


Solution 17 :

Le programme affiche :

```
Veuillez entrer votre chaîne de caractères : science
chaîne2 vaut : science
Veuillez entrer votre chaîne de caractères : informatique
chaîne1 vaut : Ahmed informatique Mohamed Ali...
```

Solution 18 :

Programme C :

```
#include <stdio.h>
#include <string.h>
main()
{
  char CH1[15], CH2[15];
  printf("Donnez la première chaîne : ") ;
  gets(CH1) ;
```

```
  printf("Donnez la deuxième chaîne : ") ;
  gets(CH2) ;
  if (strcmp(CH1,CH2) == -1) printf("%s  %s", CH1,CH2) ;
  else printf("%s  %s", CH2,CH1) ;
}
```

Solution 19 :

Programme C :

```
#include <stdio.h>
#include <string.h>
main()
{
  char CH[20] ;
  char C ;
  int nb, i ;
  printf("Donnez une chaîne de 19 caractères maximum : ") ;
  gets(CH) ;
  printf("Entrez un caractère : ") ;
  scanf("%c", &C) ;
  nb = 0 ;
  for(i=0; i<=strlen(CH)-1; i++) if (CH[i] == C) nb = nb + 1 ;
  printf("Le caractère %c est apparu %d fois dans la chaîne %s.", C, nb, CH) ;
}
```


Solution 20 :

Programme C :

```
#include <stdio.h>
main () {
  char chaine[50] ;
  int i, j;
  puts("Donnez une chaîne de caractères :") ;
  gets(chaine);
  i = 0 ;
  while (chaine[i] != '\0')
    if (chaine[i] == 'e')
      for(j=i; chaine[j] != '\0'; j++) chaine[j]=chaine[j+1];
    else i = i + 1 ;
  puts("Voici la chaîne après la suppression du caractère 'e' :") ;
  puts(chaine) ;
}
```

Solution 21 :

Programme C :

```
#include <stdio.h>
#include <string.h>
main () {
  char chaine[30] ;
  int i;
  puts("Donnez une chaîne de caractères :") ;
  gets(chaine);
  puts("La voici affichée à l'envers :") ;
```

```
  for(i=strlen(chaine)-1 ; i >=0 ; i--)
     printf("%c", chaine[i]) ;
}
```

Solution 22 :

Programme C :

```
#include <stdio.h>
main () {
  char chaine[20] ;
  int i, s;
  printf("Donnez une chaîne de caractères : ") ;
  gets(chaine);
  s = 0 ;
  for(i=0; i<=strlen(chaine)-1 ;i++) s = s + chaine[i] ;
  printf("La somme des codes ASCII = %d", s) ;
}
```

Solution 23 :

Programme C :

```
#include <stdio.h>
#include <string.h>
main () {
  char verbe[20], ch[20] ;
  strcpy(ch,"");
  printf("Donnez un verbe du premier groupe : ");
  gets(verbe);
  strncat(ch, verbe, strlen(verbe)-2);
  printf("Conjugaison du verbe %s au présent de l'indicatif :\n", verbe) ;
  printf("Je %se\n", ch);
  printf("Tu %ses\n", ch);
  printf("Il/Elle %se\n", ch);
  printf("Nous %sons\n", ch);
  printf("Vous %sez\n", ch);
  printf("Ils/Elles %sent\n", ch);
}
```


Avec le verbe `"sauter"`, le programme affiche :

```
Donnez un verbe du premier groupe : sauter
Conjugaison du verbe sauter au présent de l'indicatif :
Je saute
Tu sautes
Il/Elle saute
Nous sautons
Vous sautez
Ils/Elles sautent
```

Solution 24 :

Programme C :

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
main ()
{
  char mots[5][20];
  int i,j;
  char CH[20];
  // Lire les chaînes de caractères
  for(i=0; i<=4; i++){
    printf("Donnez le mot n° %d : ", i);
    gets(mots[i]);
  }
  // Convertir les mots en majuscules
  for(i=0; i<=4; i++)
    for (j = 0 ; mots[i][j] != '\0' ; j++)
     mots[i][j] = toupper(mots[i][j]);/* Dans la condition de
         la boucle for, on peut mettre j<=strlen(mots[i])-1 */
  // Trier les chaînes par ordre alphabétique
  for(i=0; i<=3; i++)
    for(j=i+1; j<=4; j++)
      if (strcmp(mots[i],mots[j])==1){
          strcpy(CH,mots[i]);
          strcpy(mots[i],mots[j]);
          strcpy(mots[j],CH);
      }
  // Afficher la liste des mots triés
  printf("Voici la liste des mots après le tri :\n");
  for(i=0; i<=4; i++) puts(mots[i]);
}
```


Solution 25 :

```
#include <stdio.h>
#include<string.h>
main()
{
  char T[10][5];
  int i;
  printf("Saisir les chaînes de caractères :\n");
  for(i=0;i<=9;i++) scanf("%s", T[i]);
  printf("Les derniers caractères des chaînes :\n");
  for(i=0;i<=9;i++)
   printf("Le dernier caractère de la chaîne %s est %c\n",T[i], T[i][strlen(T[i])-1]);
}
```

Chapitre 6 : Les sous-programmes : Procédures et Fonctions

1. Introduction

Pour avoir des programmes fiables, lisibles et faciles à maintenir, il faut les réduire et les organiser en utilisant les sous-programmes. Ces derniers permettent donc :

1. *Réduction de la taille des programmes :* Il est possible de déterminer les blocs analogues, les substituer par un sous-programme, ensuite appeler le sous-programme par son nom, dans des points d'appel au niveau du programme appelant (Programme Principal).
   Soit le programme C suivant :

```
#include <stdio.h>
main()
{
  int x, y, s ;
  //*** bloc 1 ***
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
  //*** bloc 2 ***
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
}
```

   Pour réduire le code de ce programme, il est possible d'utiliser un sous-programme (dans cet exemple une procédure) comme suit :

```
#include <stdio.h>
int x, y, s ;
void somme() // Définition de la procédure
{
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
}
main() // Programme Principal
{
  somme(); // Point d'appel
```

```
  somme(); // Point d'appel
}
```

Notons ici que les variables ont été déplacées avant la procédure pour qu'elles soient accessibles aussi bien au niveau de la procédure qu'au niveau du programme principal.

2. *Organisation du code :* Le problème initial sera découpé en sous-problèmes. Chaque sous-problème sera résolu par un sous-programme.
   Soit le programme C suivant :

```
#include <stdio.h>
main()
{
  int x, y, s, p, d ;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
  p = x * y ;
  printf("Le produit = %d\n", p) ;
  d = x - y ;
  printf("La différence = %d\n", d) ;
}
```


Pour plus d'organisation, il est possible de subdiviser le problème en trois sous-problèmes (addition, multiplication et soustraction). Chaque sous-problème sera résolu par un sous-programme (procédure, dans cet exemple). On obtient le programme suivant :

```
#include <stdio.h>
  int x, y, s, p, d ;
  void somme()
  {
  s = x + y ;
  printf("La somme = %d\n", s) ;
  }
  void produit()
  {
  p = x * y ;
  printf("Le produit = %d\n", p) ;
  }
  void difference()
  {
  d = x - y ;
  printf("La différence = %d\n", d) ;
  }
main() // Programme Principal
{
  printf("Donnez la première valeur : ") ;
```

```
    scanf("%d", &x) ;
    printf("Donnez la deuxième valeur : ") ;
    scanf("%d", &y) ;
    somme();
    produit();
    difference();
  }
```

Remarque : Dans ce cas-là, on utilise souvent le terme module au lieu de sous-programme, et on parle de la programmation modulaire.

2. Les sous-programmes

Un sous-programme est un petit programme composé d'un *en-tête*, d'une partie *données* et d'une partie *instructions*. Le programme de calcul de la somme peut être écrit comme suit :

```
#include <stdio.h>
void somme()
{
  int x, y, s ;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
}
main()
{
  somme();
  somme();
}
```


Un sous-programme avec paramètres :

Dans le cas où on a des blocs qui se ressemblent en termes d'instructions, mais ils utilisent des variables différentes, on peut utiliser un sous-programme avec des paramètres (arguments).

Soit le programme C suivant :

```
#include <stdio.h>
main()
{
  int x, y, z, h, s ;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = x + y ;
  printf("La somme = %d\n", s) ;
  printf("Donnez deux valeurs :\n") ;
  scanf("%d%d", &z, &h) ;
  s = z + h ;
```

```
  printf("La somme = %d\n", s) ;
}
```

Pour réduire le code de ce programme, il est possible d'utiliser un sous-programme (procédure, dans cet exemple) avec deux paramètres. On obtient alors le programme suivant :

```
#include <stdio.h>
int x, y, z, h, s ;
void somme(int a, int b)
{
  s = a + b ;
  printf("La somme = %d\n", s) ;
}
main()
{
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  somme(x, y);
  printf("Donnez deux valeurs :\n") ;
  scanf("%d%d", &z, &h) ;
  somme(z, h);
}
```


Les paramètres sont indiqués entre parenthèses et séparés par des virgules. Pour chacun, on doit préciser le type et le nom. Dans le cas d'absence de paramètres, les parenthèses doivent être mises vides.

La déclaration : `void somme(int a, b)` n'est pas acceptée.

Les paramètres `x`, `y` et `z`, `h` sont dits **effectifs**, fournis lors de l'invocation du sous-programme. Les paramètres `a`, `b` sont dits **formels**, apparaissant lors de la déclaration du sous-programme.

Les paramètres formels et effectifs d'un même sous-programme, dans le même programme, doivent être de même nombre, tout en respectant l'ordre des types des paramètres.

Types de sous-programmes :

Il existe deux types de sous-programmes : Procédures et Fonctions.

Les procédures : Une procédure est un ensemble d'instructions regroupées sous un nom, et qui réalise un traitement particulier dans un programme lorsqu'on l'appelle (effet de bord).

En C, une procédure est appelée directement dans un programme par son nom.

En C, l'en-tête de la procédure comporte le nom de la procédure précédé par le mot clé `void`, et suivi éventuellement de paramètres mises entre parenthèses. Ensuite, nous avons le corps de la procédure qui est un bloc d'instructions implémentant le traitement réalisé par cette procédure.

Les fonctions : Une fonction est un sous-programme qui retourne un et un seul résultat au programme appelant.

En C, le point d'appel d'une fonction apparaît souvent dans une expression (partie droite d'une affectation, dans un `printf()` ou dans une condition), mais il est possible d'appeler une fonction directement comme s'il s'agit d'une procédure.

En C, l'en-tête de la fonction comporte le nom de la fonction précédé par son type, et suivi éventuellement de paramètres mises entre parenthèses. Ensuite, nous avons le corps de la fonction. A l'intérieur du corps de la fonction, le mot clé `return` est suivi du résultat retourné par cette fonction. Le résultat peut être une simple valeur ou toute une expression. `return` interrompe la fonction et rend la main au programme appelant.

Remarques :

- Si le type d'un sous-programme n'est pas indiqué, le compilateur suppose qu'il s'agit d'une fonction à résultat entier.
- L'expression après `return` doit être de même type que la fonction. La même chose pour les paramètres effectifs qui doivent être de mêmes types que les paramètres formels. Si ces deux règles ne sont pas respectées, le compilateur C ne déclare pas une erreur, mais ceci peut mener à des résultats inattendus ou erronés.
- Si on ne met pas un `return` dans une fonction, le compilateur C ne déclare pas une erreur, mais ceci peut mener à des résultats inattendus ou erronés.
- Si vous trouvez dans un autre document l'écriture `void main()`, ne vous en faites pas, c'est juste pour dire que le programme principal se comporte comme une procédure.

Le programme C suivant permet de calculer la somme de deux valeurs lues à partir du clavier en utilisant une fonction :

```
#include <stdio.h>
int x, y, s ;
int somme() // Définition de la fonction
{ int sm ;
  sm = x+y ;
  return sm ; /* On peut mettre directement return x+y ; et par conséquent
                 enlever la déclaration int sm ; et l'instruction sm = x+y ; */
}
main() // Programme Principal
{
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  s = somme();
  printf("La somme = %d\n", s) ; /* On peut mettre directement
```

```
                printf("La somme = %d\n", somme()) ; et par conséquent enlever la
                déclaration de la variable s et l'instruction  s = somme() ; */
}
```

Le mot clé `return` à la fin de la fonction permet de retourner le résultat de la fonction.

Voyons un deuxième exemple montrant une fonction avec des paramètres :

```
#include <stdio.h>
int Sup(int a, int b)
{ int x ;
  if (a > b) x = 1 ;
    else x = 0;
  return x;
}
main()
{
  int x, y ;
  printf("Donnez la première valeur : ") ;   scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;   scanf("%d", &y) ;
  if (Sup(x,y)) printf("%d est supérieure à %d", x, y);
    else printf("%d est supérieure à %d", y, x);
}
```

Ce programme permet de comparer deux nombres entiers en indiquant quelle est la valeur supérieure.

La fonction `Sup()` du programme précédent peut être écrite comme suit :

```
int Sup(int a, int b)
{
  if (a > b) return 1;
    else return 0;
}
```


Remarques :

- Si on appelle une fonction directement comme une procédure, sans qu'elle figure dans une expression, le compilateur C ne déclare pas une erreur. Dans ce cas, le traitement lié à la fonction sera effectué, mais le résultat retourné par la fonction ne sera pas utilisé. Si on fait le contraire, i.e. on invoque une procédure dans une expression, cette fois-ci le compilateur déclare une erreur, et n'exécute pas le programme.
- Malgré qu'elle soit souvent appelée directement comme une procédure, `printf` est une fonction entière appartenant à la bibliothèque `stdio.h`, et qui retourne le nombre de caractères affiché en cas de succès, ou bien une valeur négative en cas d'échec. `scanf`, elle aussi, est une fonction qui retourne le nombre de variables affectées par la saisie en cas de succès, ou bien `EOF` (qui a généralement la valeur `-1`) en cas d'echec de lecture.

En C, il existe aussi plusieurs fonctions appartenant à la bibliothèque `math.h` permettant de faire des calculs mathématiques, telles que `pow`, `sqrt`, `cos`, `sin`, `tan`, etc.

La fonction `abs` appartient à la bibliothèque `stdlib.h`.

Le programme C suivant affiche : `La valeur absolue de -7 = 7`

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  printf("La valeur absolue de -7 = %d\n",abs(-7));
}
```

Exercice : Ecrire votre propre fonction `abs`. Nommez la `Absolue`. La fonction `Absolue` doit retourner la valeur absolue d'un paramètre entier.

Solution :

```
#include <stdio.h>
int Absolue(int a) {
    if(a < 0) a=-a;
    return a;
}
main()
{   int x ;
    printf("Donnez une valeur : ");
    scanf("%d", &x);
    printf("La valeur absolue de %d est %d\n", x, Absolue(x));
}
```


Fichier d'en-tête (bibliothèque) :

Nous avons souvent utilisé dans ce cours le terme *bibliothèque* pour référencer les fichiers d'en-tête (header file), tels que `stdio.h`, `math.h`, etc. En fait, une bibliothèque n'est rien d'autre qu'un ensemble de fonctions et/ou procédures stockées dans un fichier (`.h`) qui est invoqué en tant que fichier d'en-tête (header file) dans un programme C.

Ci-dessous, un exemple de création d'une bibliothèque en C :

1. Ecrire le code :

```
int Somme(int x, int y)
{
  return x + y;
}
int Difference(int x, int y)
{
  return x - y;
}
```

2. Sauvegardez le fichier sous le nom *Operations.h*, pour dire que c'est un header file.
3. Dans le même répertoire, écrire le programme C :

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

```
#include "Operations.h" // Appel de la bibliothèque Operations.h
main()
{
  int x, y;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  printf("Somme = %d\n", Somme(x,y)) ;
  printf("Difference = %d", Difference(x,y)) ;
}
```

4. Sauvegarder, compiler et exécuter votre programme.

Remarque : L'appel de la bibliothèque au niveau du programme a été fait par `#include "Operations.h"` pour dire qu'il s'agit d'un fichier d'en-tête du répertoire courant. Pour appeler le fichier par `#include <Operations.h>`, il faut le sauvegarder dans le répertoire où le compilateur C stocke ce genre de fichiers.

Appel d'un sous-programme :

Contrairement à d'autres langages, en C, on ne peut pas définir un sous-programme à l'intérieur d'un autre. Tous les sous-programmes sont au même niveau, c'est-à-dire que chaque sous-programme peut appeler un autre, comme c'est le cas dans l'exemple suivant :


```
#include <stdio.h>
int somme(int a, int b) // Fonction appelée
{
  return a+b;
}
void affichage(int e, int f) // Procédure appelante
{
  printf("%d + %d = %d\n", e, f, somme(e,f));
  printf("%d * %d = %d\n", e, f, produit(e,f));
}
int produit(int c, int d) // Fonction appelée
{
  return c*d;
}
main() // Programme Principal
{
  int x, y ;
  printf("Donnez la première valeur : ");
  scanf("%d", &x);
  printf("Donnez la deuxième valeur : ");
  scanf("%d", &y);
  affichage(x,y);
}
```

En C, un sous-programme peut être appelé alors qu'il est défini ultérieurement, i.e. défini après le sous-programme appelant. Comme c'est le cas dans le programme précédent là où la procédure appelante `affichage` appelle la procédure appelée `produit`. Il ne faut pas oublier les parenthèses quand on appelle un sous-programme, sinon cela provoque une erreur de compilation.

Déclaration et définition d'un sous-programme :

Si maintenant un sous-programme est défini après le programme principal (`main()`), alors il doit être déclaré comme montré dans l'exemple suivant :

```
#include <stdio.h>
void afficher_somme(int e, int f); /* Déclaration de la procédure, i.e.,
                mettre seulement l'en-tête de la procédure suivi d'un point-virgule */
main() // Programme Principal
{
  int x, y ;
  printf("Donnez la première valeur : ");
  scanf("%d", &x);
  printf("Donnez la deuxième valeur : ");
  scanf("%d", &y);
  afficher_somme(x,y); // Appel de la procédure
}
void afficher_somme(int e, int f) /* Définition de la procédure, i.e.,
                                  associer l'en-tête au corps de la procédure */
{
  printf("%d + %d = %d\n", e, f, e+f);
}
```


Remarques :

- Si une fonction ou procédure est mise après `main()` sans déclaration, ceci est sanctionné par un [Warning] signalé par le compilateur.
- La déclaration de la procédure `afficher_somme` peut être mise à l'intérieur du `main()` avant l'appel de la procédure.
- Dans la déclaration d'une procédure, quand l'en-tête est mis complet avec ses types de paramètres, il est dit *prototype*, comme c'est le cas dans l'exemple précédent. La déclaration peut donc ne pas comporter de paramètres.

3. Les variables locales et les variables globales

Soit le programme C suivant :

```
#include <stdio.h>
int s ;
void somme()
{
  int x, y ;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
```

```
  scanf("%d", &y) ;
  s = x + y ;
}
main()
{
  //*** bloc 1 ***
  somme();
  printf("La somme = %d\n", s) ;
  //*** bloc 2 ***"
  somme();
  printf("La somme = %d\n", s) ;
}
```

Les variables `x` et `y` sont dites **variables locales**. Elles ne peuvent être utilisées qu'au niveau du sous-programme. La variable `s` est dite **variable globale**. Les variables globales peuvent être utilisées au niveau des sous-programmes et au niveau du programme principal.

Remarques :

- Les paramètres d'un sous-programme se comportent de la même manière que les variables locales de ce sous-programme.
- Si les variables globales sont déclarées après le programme qu'il les utilise, ceci est sanctionné par une [Error] signalée par le compilateur.

Portée d'une variable :

Dans un même programme ou un même sous-programme, l'endroit où la déclaration d'une variable est faite, influence directement la portée de la variable (le bloc dans lequel il est possible d'accéder à la variable), comme illustré dans l'exemple suivant :

```
main()
{
    int A;
    // Variables accessibles : A
    {
            int B;
            // Variables accessibles : A, B
            {
                    int C;
                    // Variables accessibles : A, B, C
            }
    }
    {
            int E;
            // Variables accessibles : A, E
    }
}
```

Notons qu'une variable peut être déclarée dans n'importe quel bloc dans un programme C.

Deux identificateurs identiques :

Pour enlever l'ambiguïté, il est préférable d'éviter d'utiliser le même identificateur comme variable locale et globale au même temps. Voyons l'exemple suivant :

```
#include <stdio.h>
int  i ;
void S()
{
  int  i ;
  i = 5;
  printf(" i = %d\n", i);
}
main()
{
  i = 10;
  S();
  printf(" i = %d", i);
}
```

Le programme affiche:

```
 i = 5
 i = 10
```

Convertir une fonction en procédure :

Il est possible de remplacer une fonction par une procédure en utilisant une variable globale permettant de récupérer la valeur retournée par la fonction. Soit le programme C suivant :


```
#include <stdio.h>
int x, y ;
int S(int a, int b)
{
  return a + b ;
}
main()
{
  printf("Donnez la première valeur : ");
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ");
  scanf("%d", &y) ;
  printf("Somme = %d", S(x,y) ) ;
}
```

La fonction `S` peut être remplacée par une procédure `S` en ajoutant une variable globale, et cela comme suit :

```
#include <stdio.h>
int x, y, Z ;
void S(int a, int b)
{
  Z = a + b ;
}
```

```
main()
{
  printf("Donnez la première valeur : ");
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ");
  scanf("%d", &y) ;
  S(x,y);
  printf("Somme = %d", Z) ;
}
```

Variable locale statique :

Il est possible d'attribuer un emplacement permanent à une variable locale, en la déclarant par le mot clé `static`. Sa valeur se conserve donc d'un appel au suivant. Soit l'exemple :

```
#include <stdio.h>
void fct()
{
  static int i;
  i++;
  printf("%d\n", i);
}
main() // Programme Principal
{
  int n ;
  for(n=0; n<=1; n++) fct();
}
```

Ce programme affiche :

```
1
2
```


Remarques :

- Une variable locale statique se comporte donc comme une variable globale, sauf que son utilisation se limite au sous-programme là où elle a été déclarée.
- Elle est automatiquement initialisée par 0.
- Le mot `static`, employé sans indication de type, est équivalent à `static int`.
- Une déclaration d'une variable globale précédée par le mot clé `extern` précise que cette variable a été déjà définie dans un autre fichier source. Une telle déclaration n'effectue aucune réservation d'espace mémoire.

4. Le passage des paramètres

Soit le programme C suivant :

```
#include <stdio.h>
void change1(int y) // Procédure
{
  y = 1;
}
```

```
main() // Programme Principal
{
  int I ;
  I = 0;
  change1(I) ;
  printf("%d", I) ;
}
```

Ce programme comporte une procédure `change1` déclarée avec un paramètre formel `y`. Le programme principal `main()` contient une variable locale `I`, passée comme paramètre effectif à la procédure `change1`. La valeur qui sera affichée en sortie est `0`.

Malgré que le paramètre ait changé au niveau de la procédure, ce changement n'a pas été transmis au programme principal. Ce type de passage de paramètres est dit *par valeur*.

Voyons un deuxième exemple :

```
#include <stdio.h>
void change2(int *y) /* y est un pointeur vers un entier, i.e., y contient
                        l'adresse d'une variable de type entier */
{
  *y = 1;  // la variable pointée par y reçoit la valeur 1
}
main()
{
  int I ;
  I = 0;
  change2(&I) ; // On passe comme paramètre l'adresse de la variable I
  printf("%d", I) ;
}
```


`y` est un paramètre formel de la procédure `change2`. Le symbole `*` indique le type pointeur. Ce symbole est utilisé pour transmettre la valeur de la variable pointée par `y` vers le programme principal. Par conséquent, la valeur affichée par le programme sera `1`.

Une variable de type pointeur, au lieu de contenir la valeur d'une variable, elle contient son adresse. Donc lors de l'appel de la procédure, on met comme paramètre effectif une adresse d'une variable indiquée par le symbole `&` précédant le nom de la variable. Le type pointeur sera vu en détail dans un prochain chapitre.

Le passage de paramètres par le symbole `*` est dit *par adresse* (*par variable* ou *par référence*).

Remarques :

- Quand on utilise le passage par valeur, il est possible d'invoquer le sous-programme par des paramètres effectifs exprimés sous forme d'expressions, par exemple, `change1(I*4)`, `change1(4)`…

- Pour la fonction `printf`, les paramètres sont passés par valeur. Pour la fonction `scanf`, les paramètres sont passés par adresse.

Représentation graphique de la pile :

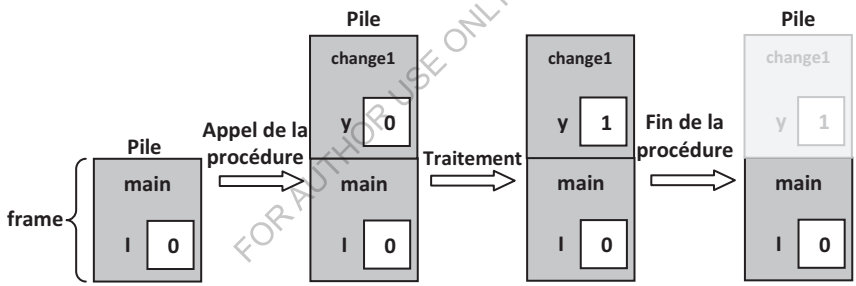
La zone mémoire réservée pour l'exécution d'un programme C dans la RAM est constituée de plusieurs parties, parmi lesquelles on cite :

- Segment de données : variables globales.
- Pile : variables locales et paramètres.
- Tas : allocation dynamique, qui sera étudiée ultérieurement.

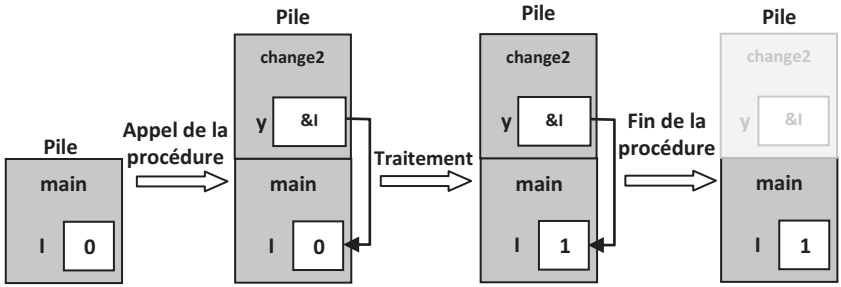
La répartition de l'espace mémoire entre ces différentes parties dépend du modèle mémoire adopté : Small (le modèle par défaut), Tiny, Medium, etc. Le choix du modèle se fait avant la compilation du programme.

En particulier, la pile (stack en anglais) se compose d'une ou plusieurs boites (frames). Dans chacune de ces boites sont stockées, entre autres, les variables locales et les paramètres d'une procédure ou fonction. La partie de la pile allouée à une fonction ou procédure est désallouée quand elle se termine.

La pile avec la procédure `change1` peut être représentée comme suit :



La pile avec la procédure `change2` peut être représentée comme suit :



Remarquez bien qu'en C, le programme principal (`main()`) est traité comme une simple fonction.

Exercice :

1. Qu'affiche le programme C suivant ?

```
#include <stdio.h>
int x, y, Som;
void S(int a, int b)
{  Som = a + b ;
   a = a + b;
   b = 2 ;
}
main()
{  x = 2 ;  y = 9 ;
  S(x,y);
  printf("%d + %d = %d", x, y, Som);
}
```

2. Qu'affiche programme précédent si :
   a) on remplace la procédure par :
   ```
   void S(int a, int *b)
   {  Som = a + *b ;
     a = a + *b;
     *b = 2 ;
   }
   ```
   et on l'appelle par : `S(x,&y);`
   b) on remplace la procédure par :
   ```
   void S(int *a, int b)
   {  Som = *a + b ;
     *a = *a + b;
     b = 2 ;
   }
   ```
   et on l'appelle par : `S(&x,y);`
   c) on remplace la procédure par :
   ```
   void S(int *a, int *b)
   {  Som = *a + *b ;
     *a = *a + *b;
     *b = 2 ;
   }
   ```
   et on l'appelle par : `S(&x,&y);`

Solution :

1. Le programme affiche : `2 + 9 = 11`
2. Le programme affiche :
   a) Pour `S(int a, int *b)` : `2 + 2 = 11`
   b) Pour `S(int *a, int b)` : `11 + 9 = 11`
   c) Pour `S(int *a, int *b)` : `11 + 2 = 11`

Passer un tableau en paramètre :

En langage C, il est possible de définir un sous-programme avec un paramètre de type tableau. Rappelons qu'une variable de type tableau correspond à l'adresse du premier élément (l'élément 0) de ce tableau. Donc, le passage d'un

tableau en paramètre est un passage par adresse, sans le préciser par `*`, ni l'appeler par `&`.

Soit le programme C suivant :

```
#include <stdio.h>
void initialiser(int V[4])
{
  V[0]=0;V[1]=0;V[2]=0;V[3]=0;
}
main()
{
  int T[4]={10, 12, 14, 16} ;
  int i=0;
  initialiser(T) ; /* Malgré que c'est un passage par adresse, on ne met pas
                      initialiser(&T) ; car T lui-même est une adresse */
  for(i=0; i<4; i++) printf("%d\n", T[i]);
}
```

Le programme affiche :

```
0
0
0
0
```


Remarque : Quand on passe un tableau comme paramètre d'un sous-programme, il est possible de ne pas préciser la taille de ce tableau. La déclaration `void initialiser(int V[])` est donc aussi correcte. Il est aussi possible de mettre la déclaration `void initialiser(int *V)`.

Nombre variable de paramètres :

Le langage C permet de définir une fonction ou procédure avec un nombre indéterminé de paramètres. La fonction doit avoir au moins un argument fixe.

La procédure définie dans le programme suivant permet d'afficher ses paramètres jusqu'à atteindre le paramètre ayant comme valeur `-1`.

```
#include<stdio.h>
#include<stdarg.h>
void essai(int par1, char par2, ...)
{
  va_list adpar;
  int parv;
  printf("%d\n", par1);
  printf("%c\n", par2);
  va_start(adpar, par2);
  parv = va_arg(adpar, int);
  while(parv != -1){
    printf("%d\n", parv);
    parv = va_arg(adpar, int);
  }
}
```

```
main()
{
  puts("Premier essai :");
  essai(22, 'c', 19, 14, 17, -1);
  puts("Deuxième essai :");
  essai(1, 'a', 1, -1);
}
```

Le programme affiche :

```
Premier essai :
22
c
19
14
17
Deuxième essai :
1
a
1
```


Le type `va_list` identifie une liste de variables. La fonction `va_start` permet de se pointer juste après un argument fixe. La fonction `va_arg` permet de récupérer l'argument en cours selon un type donné, et passer à l'argument suivant. `va_list`, `va_start` et `va_arg` appartiennent à la bibliothèque `stdarg.h`.

Remarque : On appelle fonction variadique, une fonction qui peut prendre un nombre variable de paramètres. `printf` et `scanf` sont des fonctions variadiques.

5. La récursivité (récursion)

5.1. Définition

Une procédure ou fonction est dite récursive si elle fait référence à elle-même, i.e., elle s'appelle elle-même.

5.2. Exemple (Calcul de la factorielle)

La factorielle d'un nombre *n* se note *n!*, avec *n! = n*(n-1)* ... *2*1*. Ainsi, *5! = 5*4*3*2*1*.

La fonction itérative (en utilisant une boucle) pour calculer la factorielle est donnée ci-dessous :

```
int fact(int n)
{ int i, f ;
  f = 1;
  for (i=1 ; i<=n ; i++) f = f*i ;
  return f ;
}
```

La définition mathématique de la factorielle en tant que formule récurrente est la suivante : pour tout *n* entier, si *(n = 0) fact(0) = 1*, si *(n>0) fact(n) = n * fact(n-1)*.

La fonction récursive pour le calcul de la factorielle est la suivante :

```
int fact(int n)
{ int f ;
  if (n==0) f = 1;
      else f = n*fact(n-1) ;
  return f;
}
```

Ou bien :

```
int fact(int n)
{  if (n==0) return 1;
      else return n*fact(n-1) ;
}
```

Remarques :

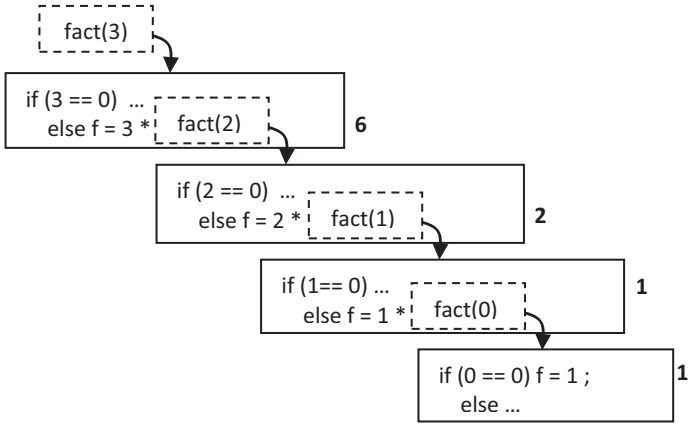
- Toute solution itérative peut être transformée en solution récursive, et vice versa.
- L'appel d'une fonction à l'intérieur d'elle-même est dit appel récursif. L'appel récursif dans la fonction récursive `fact` est `fact(n-1)`.

Une fonction ou une procédure récursive doit vérifier les deux conditions suivantes :

1. L'existence d'un critère d'arrêt. Le critère d'arrêt dans la fonction récursive `fact` est `(n==0)`.
2. Les paramètres de l'appel récursif doivent converger vers le critère d'arrêt.

Arbre des appels :

L'arbre des appels est une représentation graphique des appels récursifs. L'arbre des appels de la fonction récursive `fact` pour *n* = 3 peut être représenté comme suit.



Remarques :

- L'utilisation de la récursivité est coûteuse en termes d'espace mémoire. Comme illustré par l'arbre des appels, le contexte de la fonction `fact` pour n=3 va être stocké 4 fois dans la pile. Ceci peut nous ramener à un problème dit Débordement de pile (Stack Overflow) si le n devient plus grand.
- L'arbre des appels devient plus complexe quand la fonction s'appelle elle-même plus d'une fois.

6. Exercices corrigés

6.1. Exercices

Exercice 1 :

Ecrire un programme C qui calcule la division entière entre deux nombres lus à partir du clavier en utilisant une fonction avec deux paramètres entiers.
Donnez la représentation graphique de la mémoire (pile) qui correspond au programme quand l'utilisateur donne successivement les deux nombres : 9 et 2.

Exercice 2 :

Ecrire un programme C qui affiche le code ASCII d'un caractère lu à partir du clavier en utilisant une fonction.

Exercice 3 :

Ecrire un programme C qui décide si un nombre est premier ou non en utilisant une fonction qui restituera la valeur 0 lorsque le nombre n'est pas premier et la valeur 1 lorsqu'il est premier.
Note : On dit qu'un nombre entier positif est premier s'il possède exactement deux diviseurs positifs : 1 et lui-même. 0 et 1 ne sont pas des nombres premiers. Voici quelques nombres premiers : 2, 3, 5, 7, 11, 13, 17…

Exercice 4 :

Ecrire une fonction qui calcule la moyenne de trois nombres passés en paramètres. Ecrire un programme C qui affiche la moyenne de trois nombres lus à partir du clavier en appelant la fonction déjà définie.

Exercice 5 :

Ecrire un programme C permettant de calculer la surface d'un rectangle en utilisant une fonction ayant comme paramètres la longueur et la largeur de ce rectangle.

Exercice 6 :

Ecrire une procédure qui retourne le quotient et le reste de la division d'un entier p par un entier q. Ecrire le programme C qui lit deux nombres, appelle cette procédure et affiche les résultats.

Exercice 7 :

Ecrire un programme C permettant d'afficher la somme de deux entiers et la concaténation de deux chaînes de caractères en utilisant une procédure.

La procédure aura comme paramètres deux entiers et deux chaînes de caractères.

Exercice 8 :

Ecrire un programme C permettant à partir de la saisie de trois entiers *a*, *b* et *c*, de calculer la factorielle de : soit *a*, soit *b*, soit *c*. L'utilisateur doit faire un choix à partir d'un menu. Le calcul de la factorielle doit être effectué par une fonction. Remplacer ensuite (dans un autre programme) la fonction par une procédure.

Exercice 9 :

Soit un barème de l'impôt défini comme suit : pour un ménage *X* avec un revenu total *R* et un nombre *n* de membres du foyer, l'impôt est donné par :

- 10% de *R* si *R/n < 500* DA
- 20% de *R* si *R/n >= 500* DA

Ecrire un programme C qui lit *R* et *n* au clavier, puis affiche l'impôt et le revenu net en utilisant les deux fonctions suivantes :

1. Une fonction `Impot` qui calcule le montant de l'impôt en fonction de *R* et *n*.
2. Une fonction `RevenuNet` qui calcule le revenu net d'un ménage après paiement de l'impôt en fonction de *R* et *n*.

Exercice 10 :

Qu'affiche le programme C suivant ?

```
#include <stdio.h>
int n =5;
void fct(int p)
{
  printf("%d %d\n", n, p);
}
void main() // Programme Principal
{
  int n = 2;
  fct(n);
}
```

Exercice 11 :

Quel est le résultat d'exécution des deux programmes suivants ?

```
#include <stdio.h>
  int i ;
  int imp(int a)
{
  printf("%d\n", a);
  a = a+2 ;
}
main()
{
  i = 1;
  while (i<10) {
```

```
#include <stdio.h>
  int i ;
  int imp(int *a)
{
  printf("%d\n", *a);
  *a = *a+2 ;
}
main()
{
  i = 1;
  while (i<10) {
```

```
        imp(i);
        i = i+3;
    }
}
```

```
        imp(&i);
        i = i+3;
    }
}
```

Exercice 12 :

Qu'affichent les deux programmes suivants ?

Programme 1

```
#include <stdio.h>
int f(int a, int b)
{
  a=a+a ;
  return a+b ;
}
void main()
{
 int i, j, x ;
 i=10 ;
 j=40 ;
 x=f(i,j) ;
 printf("i = %d, j = %d, x = %d", i, j, x) ;
}
```

Programme 2

```
#include <stdio.h>
int f(int *a, int *b)
{
  *a=*a+*a ;
  return *a+*b ;
}
void main()
{
 int i, j, x ;
 i=10 ;
 j=40 ;
 x=f(&i,&j) ;
 printf("i = %d, j = %d, x = %d", i, j, x) ;
}
```

Exercice 13 :

Ecrire un programme C permettant de lire deux entiers et de permuter leurs contenus en utilisant une procédure qui recevra ces deux entiers en paramètres. Le programme doit enfin afficher les valeurs de ces variables afin de vérifier la permutation. Donnez la représentation graphique de la mémoire (pile) qui correspond au programme précédent quand l'utilisateur donne 9 pour la première variable et 2 pour la deuxième.

Exercice 14 :

Ecrire un programme C qui initialise deux entiers et un réel à 0 en utilisant une procédure qui aura trois variables en paramètres.

Exercice 15 :

Ecrire un programme C qui affiche la somme et le produit des éléments d'un tableau de 5 entiers lus à partir du clavier. Le programme doit utiliser pour les calculs une procédure ayant comme paramètre un tableau.

Exercice 16 :

Ecrire en C des sous-programmes de :

1. calcul de la somme des éléments d'un vecteur ;
2. recherche de la position de la première occurrence de l'élément minimum d'un vecteur ;
3. addition de deux vecteurs et rangement du résultat dans un troisième vecteur.

Ecrire ensuite un programme principal dans lequel on appellera les divers sous-programmes.

Exercice 17 :
Ecrire une fonction qui calcule la somme des éléments d'un tableau d'entiers de taille quelconque.

Exercice 18 :
En utilisant la notion de sous-programme, essayez de réduire la taille du programme C suivant :

```
#include <stdio.h>
main()
{
  int x, y, z, s ;
  puts("Donnez trois nombres :");
  scanf("%d%d%d", &x, &y, &z);
  s = x+y;
  printf("%d + %d = %d\n", x, y, s) ;
  s = y+z;
  printf("%d + %d = %d\n", y, z, s) ;
  s = z+x;
  printf("%d + %d = %d\n", z, x ,s) ;
}
```

Exercice 19 :
Soit le problème : Trouvez le maximum et le minimum dans un tableau de dix entiers, ensuite déterminez la position de la valeur maximale et la valeur minimale.

Subdivisez ce problème en sous-problèmes simples. Ensuite, essayez de les résoudre par la notion de sous-programme en langage C.

Exercice 20 :
Expliquez ce que fait la procédure suivante :

```
void P1 (int n)
{
  if (n>0) {
    printf("%d\n", n);
    P1(n-1) ;
    printf("%d\n", n);
  }
}
```

Exercice 21 :
Ecrire un programme C permettant le calcul de la puissance en utilisant une fonction itérative, ensuite un deuxième programme qui utilise une fonction récursive.

Exercice 22 :
Ecrire un programme contenant une fonction récursive permettant de calculer le $n^{ième}$ terme de la suite de Fibonacci définie comme suit : Si $n = 0$ alors $F_n = 0$; Si $n = 1$ alors $F_n = 1$; Si $n > 1$ alors $F_n = F_{n-1}+F_{n-2}$.
Donnez l'arbre des appels de la fonction pour $n=4$.

Exercice 23 :
Ecrire un programme C qui utilise une fonction récursive pour le calcul de la fonction de Ackermann, notée *A*, de deux paramètres *m* et *n*. La fonction est définie comme suit :

- Si $m<0$ ou $n<0$: $A(m, n) = 0$;
- Si $m=0$: $A(0, n) = n+1$;
- Si $n=0$: $A(m, 0) = A(m-1, 1)$;
- Si $m>0$ et $n > 0$: $A(m, n) = A(m-1, A(m, n-1))$.

Exercice 24 :
Ecrire une fonction récursive pour le calcul de la somme : $U_n = 1 + 2^4 + 3^4 + ... + n^4$, sachant que pour $n<=0$ la fonction retourne 0. Appeler la fonction dans un programme C pour $n=6$.

Réécrire un deuxième programme faisant le même travail avec une fonction itérative.

Exercice 25 :
Ecrire un programme C qui affiche la chaîne miroir d'une chaîne de caractères lue à partir du clavier en utilisant une fonction récursive.
La chaîne miroir de `"bonjour"` est `"ruojnob"`.

Exercice 26 :
Ecrire la fonction itérative, ensuite la fonction récursive permettant de calculer le PGCD (le plus grand diviseur commun) de deux entiers.

Exercice 27 :
Soit la procédure suivante contenant une boucle simple :

```
void compter()
{
  int i ;
  for(i=1; i<=10; i++) printf("%d\n", i) ;
}
```

Exprimez cette procédure par une autre récursive, faisant le même travail.

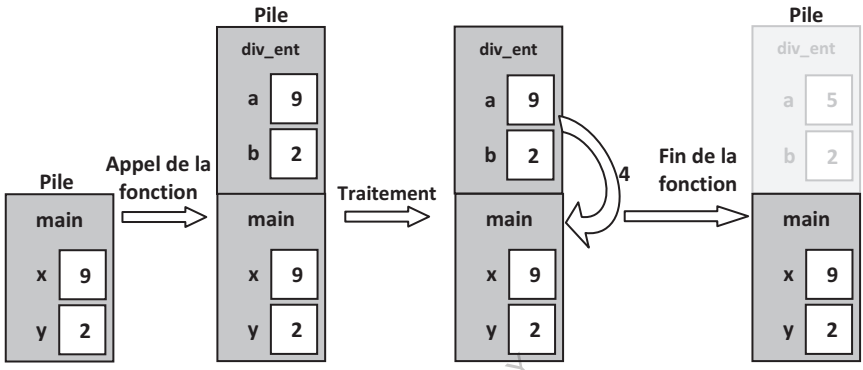
6.2. Corrigés

Solution 1 :
Programme C :

```
#include <stdio.h>
  int div_ent(int a, int b)
  {
    return a/b;
  }
main()
{
  int x, y;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
```

```
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  printf("%d / %d = %d", x, y, div_ent(x,y)) ;
}
```

Représentation de la mémoire :



Solution 2 :

```
#include <stdio.h>
  int ASCII(char a)
  {
    return a;
  }
main()
{
  char c;
  printf("Donnez un caractère : ") ;
  scanf("%c", &c) ;
  printf("Code ASCII du caractère %c est %d", c, ASCII(c)) ;
}
```


Solution 3 :

```
#include <stdio.h>
  int premier(int a)
  {
    int p = 1, i;
    for(i=2; i<= a-1; i++) if (a%i == 0) p = 0;
    if (a<=1) return 0;
    return p;
  }
main()
{
  int x;
  printf("Donnez un nombre entier positif : ") ;
  scanf("%d", &x) ;
  if (premier(x)) printf("%d est un nombre premier.", x) ;
  else printf("%d n'est pas un nombre premier.", x) ;
```

```
}
```

Ou bien :

```
#include <stdio.h>
  int premier(int a)
  {
    int p = 0, i;
    for(i=1; i<= a; i++) if (a%i == 0) p++;
    if (p!=2) return 0;
    else return 1;
  }
main()
{
  int x;
  printf("Donnez un nombre entier positif : ") ;
  scanf("%d", &x) ;
  if (premier(x)) printf("%d est un nombre premier.", x) ;
  else printf("%d n'est pas un nombre premier.", x) ;
}
```

Solution 4 :

```
#include <stdio.h>
  float moyenne(float a, float b, float c)
  {
    return (a+b+c)/3;
  }
main()
{
  float x, y, z;
  printf("Donnez trois nombres :\n") ;
  scanf("%f%f%f", &x, &y, &z) ;
  printf("La moyenne des trois nombres = %f", moyenne(x,y,z));
}
```


Solution 5 :

```
#include <stdio.h>
  float surface(float a, float b)
  {
    return a*b;
  }
main()
{
  float lng, lrg ;
  printf("Donnez la longueur et la largeur :\n");
  scanf("%f%f", &lng, &lrg);
  printf("La surface = %f", surface(lng,lrg)) ;
}
```

Solution 6 :

```
#include <stdio.h>
  int quot, mod;
  void quot_mod(int a, int b)
```

```
  {
    quot = a / b;
    mod = a % b;
  }
main()
{
  int p, q ;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &p) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &q) ;
  quot_mod(p,q);
  printf("%d / %d = %d et %d %% %d = %d", p, q, quot, p, q, mod);
}
```

Solution 7 :

```
#include <stdio.h>
#include <string.h>
  void som_con(int a, int b, char c1[10], char c2[10])
  {
    printf("Le résultat de la somme = %d\n", a+b) ;
    printf("Le résultat de la concaténation = %s", strcat(c1,c2)) ;
  }
main()
{
  int x, y ;
  char ch1[10], ch2[10];
  printf("Donnez deux nombres :\n");
  scanf("%d%d", &x, &y);
  printf("Donnez deux chaînes de caractères :\n");
  scanf("%s%s", ch1, ch2);
  som_con(x,y,ch1,ch2);
}
```


Solution 8 :

Programme C avec une fonction :

```
#include <stdio.h>
  int facto(int x)
  {
    int i, f;
    f = 1 ;
    for (i=2; i<=x; i++) f*=i ;
    return f ;
  }
main()
{
  int a, b, c, choix ;
  printf("Donnez trois entiers :\n");
  scanf("%d%d%d", &a, &b, &c);
  printf("Tapez 1 pour claculer la factorielle de : %d\n", a);
  printf("Tapez 2 pour claculer la factorielle de : %d\n", b);
```

```
  printf("Tapez 3 pour claculer la factorielle de : %d\n", c);
  scanf("%d", &choix) ;
  switch (choix)
  {
      case 1 : if (a >= 0) printf("La factorielle de %d est %d.", a, facto(a));
               else printf("Calcul impossible.");break;
      case 2 : if (b >= 0) printf("La factorielle de %d est %d.", b, facto(b));
               else printf("Calcul impossible.");break;
      case 3 : if (c >= 0) printf("La factorielle de %d est %d.", c, facto(c));
               else printf("Calcul impossible.");break;
      default : printf("Choix invalide !");
  }
}
```

Programme C avec une procédure :

```
#include <stdio.h>
  int Z;
  void facto(int x)
  {
    int i, f;
    f = 1 ;
    for (i=2; i<=x; i++) f*=i ;
    Z = f ;
  }
main()
{
  int a, b, c, choix ;
  printf("Donnez trois entiers :\n");
  scanf("%d%d%d", &a, &b, &c);
  printf("Tapez 1 pour claculer la factorielle de : %d\n", a);
  printf("Tapez 2 pour claculer la factorielle de : %d\n", b);
  printf("Tapez 3 pour claculer la factorielle de : %d\n", c);
  scanf("%d", &choix) ;
  switch (choix)
  {
     case 1 : if (a >= 0) {facto(a); printf("La factorielle de %d est %d.", a, Z) ;}
              else printf("Calcul impossible.");break;
     case 2 : if (b >= 0) {facto(b); printf("La factorielle de %d est %d.", b, Z);}
              else printf("Calcul impossible.");break;
     case 3 : if (c >= 0) {facto(c); printf("La factorielle de %d est %d.", c, Z);}
              else printf("Calcul impossible.");break;
     default : printf("Choix invalide !");
  }
}
```


Solution 9 :

```
#include <stdio.h>
  float Impot(float R, int n)
  {
    if((R/n)<500.0) return 0.1*R;
```

```
    else return 0.2*R;
  }
  float RevenuNet(float R, int n)
  {
    return R-Impot(R, n);
  }
main()
{
  float x;
  int y;
  printf("Donnez le revenu total : ") ;
  scanf("%f", &x) ;
  printf("Donnez le nombre de membres du foyer : ") ;
  scanf("%d", &y) ;
  printf("Impot = %f, RevenuNet = %f", Impot(x,y), RevenuNet(x,y)) ;
}
```

Solution 10 :

Le programme affiche :

```
5 2
```

Solution 11 :

Le premier programme affiche :

```
1
4
7
```

Le deuxième programme affiche :

```
1
6
```

Solution 12 :

Le premier programme affiche :

```
i = 10, j = 40, x = 60
```

Le deuxième programme affiche :

```
i = 20, j = 40, x = 60
```

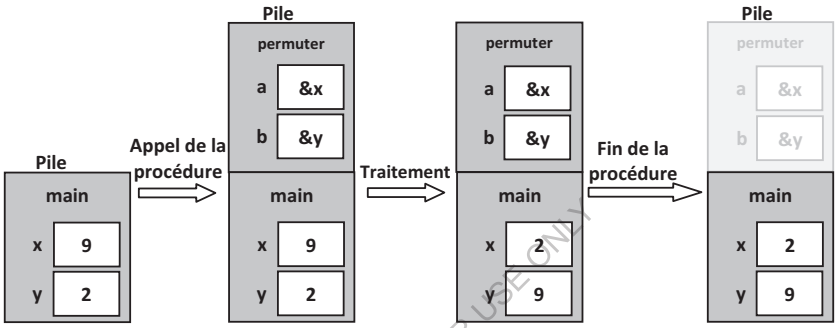
Solution 13 :

Programme C :

```
#include <stdio.h>
  void permuter(int *a, int *b)
  {
  int c;
  c = *a;
  *a = *b;
  *b = c;
  }
```

```
main()
{
  int x, y;
  printf("Donnez la valeur de x : ") ;
  scanf("%d", &x) ;
  printf("Donnez la valeur de y : ") ;
  scanf("%d", &y) ;
  permuter(&x, &y);
  printf("x = %d, y = %d", x, y);
}
```

Représentation de la mémoire :



Solution 14 :

```
#include <stdio.h>
  void init (int *a, int *b, float *c)
  {
    *a=*b=0;
    *c=0;
  }
main()
{
  int x, y;
  float z;
  init(&x,&y,&z);
  printf("x = %d     y = %d\n", x, y);
  printf("z = %f", z) ;
}
```

Solution 15 :

```
#include <stdio.h>
  int som, prod;
  void som_prod_tab(int a[5])
  {
    int k;
    som = 0;
    for(k=0; k<= 4; k++) som+=a[k];
```

```
    prod = 1;
    for(k=0; k<= 4; k++) prod*=a[k];
  }
main()
{
  int T[5];
  int i;
  puts("Donnez les cinq éléments du tableau :") ;
  for(i=0; i<=4; i++) scanf("%d", &T[i]);
  som_prod_tab(T);
  printf("Somme = %d Produit = %d", som, prod);
}
```

Solution 16 :

```
#include <stdio.h>
#define net 4 // Nombre des éléments d'un tableau
  int T1[net], T2[net], T3[net] ;
  int i;
  // Fonction de calcul de la somme d'un tableau
  int somme_tab(int T[net])
    {
      int som=0;
      for(i=0; i<=net-1; i++) som+=T[i];
      return som;
    }
  // Fonction qui détermine la première position d'un élément dans un tableau
  int pos (int x, int T[net])
    {
      int p=-1, t=0;
      i=0;
      while((i<= net-1) && !t)
           if (T[i]==x){p=i; t=1; } else i++;
      return p;
    }
  // Fonction qui détermine la position du minimum d'un tableau
  int pos_min (int T[net])
    {
      int min=T[0];
      for(i=1; i<= net-1; i++) if (min>T[i]) min=T[i];
      return pos(min, T);
    }
  // Procédure d'addition de deux tableaux
  void add_tab (int A[net], int B[net])
    {
       for(i=0; i <= net-1; i++) T3[i]=A[i]+B[i];
    }
main()
{
  printf("Donnez les éléments du premier tableau :\n") ;
  for(i=0; i<= net-1; i++) scanf("%d", &T1[i]) ;
```

```
  printf("Donnez les éléments du deuxième tableau :\n") ;
  for(i=0; i<= net-1; i++) scanf("%d", &T2[i]) ;
  printf("La somme des éléments du premier tableau = %d\n", somme_tab(T1)) ;
  printf("La position du minimum du premier tableau est : %d\n", pos_min(T1)) ;
  printf("L'addition des deux tableaux donne :\n") ;
  add_tab(T1, T2);
  for(i=0; i<= net-1; i++) printf("%d\n", T3[i]) ;
}
```

Solution 17 :

```
int som(int t[], int nb)
{
    int s = 0, i;
    for(i=0; i<nb; i++) s+=t[i];
    return s;
}
```

Solution 18 :

```
#include <stdio.h>
  int s;
  void som(int a, int b)
  {
    s = a + b;
    printf("%d + %d = %d\n", a, b, s) ;
  }
main()
{
  int x, y, z, s ;
  puts("Donnez trois nombres :");
  scanf("%d%d%d", &x, &y, &z);
  som(x,y);
  som(y,z);
  som(z,x);
}
```


Solution 19 :

Ce problème peut être subdivisé en trois sous-problèmes :

1. Déterminer le max d'un tableau.
2. Déterminer le min d'un tableau.
3. Déterminer la position d'une valeur.

Voici le programme C :

```
#include <stdio.h>
  int tab[10];
  int max, min, i;
  void maximum()
  {
    max = tab[0];
    for(i=1; i<=9; i++) if (max<tab[i]) max = tab[i] ;
  }
  void minimum()
```

```
  {
    min = tab[0];
    for(i=1; i<=9; i++) if (min>tab[i]) min = tab[i] ;
  }
  int position(int x)
  {
    int pos = -1;
    for(i=0; i<=9; i++) if (tab[i]==x) pos = i ;
    return pos;
  }
main()
{
  puts("Donnez les éléments du tableau :");
  for(i=0; i<=9; i++) scanf("%d", &tab[i]);
  maximum();
  minimum();
  printf("Le maximum est %d, sa position est %d\n", max, position(max)) ;
  printf("Le minimum est %d, sa position est %d", min, position(min)) ;
}
```

Solution 20 :

Pour un entier positif *n*, la procédure `P1(n)` permet d'afficher les valeurs de *n* jusqu'à 1, ensuite de 1 jusqu'à *n*. Par exemple, pour *n=4*, la procédure affiche :

```
4
3
2
1
1
2
3
4
```


Solution 21 :

Programme C utilisant une fonction itérative :

```
#include <stdio.h>
  int puissance(int x, int y)
  {
    int i, puis = 1;
    for(i=1; i<=y; i++) puis*=x;
    return puis;
  }
main()
{
  int n, p;
  printf("Donnez un nombre : ") ;
  scanf("%d", &n);
  printf("Donnez la puissance : ") ;
```

```
  scanf("%d", &p);
  printf("%d puissance %d = %d", n, p, puissance(n,p)) ;
}
```

Programme C utilisant une fonction récursive :

```
#include <stdio.h>
  int puissance(int x, int y)
  {
    int puis;
    if (y == 0) puis = 1;
      else puis = x * puissance(x,y-1);
    return puis;
  }
main()
{
  int n, p;
  printf("Donnez un nombre : ") ;
  scanf("%d", &n);
  printf("Donnez la puissance : ") ;
  scanf("%d", &p);
  printf("%d puissance %d = %d", n, p, puissance(n,p)) ;
}
```

La fonction récursive `puissance` *peut être aussi écrite comme suit :*

```
  int puissance(int x, int y)
  {
    if (y == 0) return 1;
      else return x * puissance(x,y-1);
  }
```


Solution 22 :

```
#include <stdio.h>
  int fibo(int n)
  {
    int f;
    if ((n==0)||(n==1)) f = n;
     else f = fibo(n-1) + fibo(n-2);
    return f;
  }
main()
{
  int x;
  printf("Entrez une valeur : ") ;
  scanf("%d", &x) ;
  printf("Fibo(%d) = %d", x, fibo(x));
}
```

La fonction récursive `fibo` *peut être aussi écrite comme suit :*

```
  int fibo(int n)
  {  if ((n==0)||(n==1)) return n;
     else return fibo(n-1) + fibo(n-2);
  }
```

Solution 23 :

```
#include <stdio.h>
  int Ackermann(int x, int y)
  {
    if ((x<0) || (y<0)) return 0;
    else if (x==0) return y+1;
    else if (y==0) return Ackermann(x-1, 1);
    else return Ackermann(x-1, Ackermann(x, y-1));
  }
main()
{
  int m, n;
  printf("Donnez m : ") ;
  scanf("%d", &m) ;
  printf("Donnez n : ") ;
  scanf("%d", &n) ;
  printf("Ackermann(%d,%d) = %d", m, n, Ackermann(m,n));
}
```

Solution 24 :

Programme C utilisant une fonction récursive :

```
#include <stdio.h>
#include <math.h>
  int Somme(int n)
  {
    if (n<=0) return 0;
    else return pow(n, 4)+Somme(n-1);
  }
main()
{
  printf("%d", Somme(6));
}
```


Programme C utilisant une fonction itérative :

```
#include <stdio.h>
#include <math.h>
  int Somme(int n)
  {
    int i, F=0;
    for(i=1; i<=n; i++) F+= pow(i, 4);
    return F;
  }
main()
{
  printf("%d", Somme(6));
}
```

Solution 25 :

```
#include <stdio.h>
#include <string.h>
  char chaine[10], miroir[10];
```

```
  char* MIR(char ch[10])
{
  if ((strlen(ch)==0) || (strlen(ch)==1)) return ch;
    else {
        char s[10] = {ch[strlen(ch)-1], '\0'};/* s une chaîne de
                                caractères contenant le dernier caractère de ch */
        char ss[10] = {'\0'};// ss une chaîne de caractères vide
        strcat(s, MIR(strncpy(ss,ch,strlen(ch)-1)));
        strcpy(miroir, s);
        return miroir;
       }
}
main()
{
  printf("Donnez une chaîne de caractères : ") ;
  scanf("%s", chaine) ;
  printf("La chaîne miroir de %s est %s", chaine, MIR(chaine)) ;
}
```

Solution 26 :

```
int PGCD(int m, int n)
{
  int r ;
  while  (m % n != 0) {
      r = m % n ;
      m = n ;      n = r ;
  }
  return n ;
}
```


La version récursive de cette fonction peut être écrite de la manière suivante :

```
int PGCD(int m, int n)
{
  if (m % n == 0) return n ;
      else return PGCD(n,m % n);
}
```

Solution 27 :

```
void compter(int i)
{
  if (i == 10) printf("%d\n", i);
  else {
    printf("%d\n", i) ;
    compter(i + 1) ;
  }
}
```

La procédure récursive `compter` sera invoquée dans le programme principal par : `compter(1);`

Chapitre 7 : Les types personnalisés

1. Introduction

Les types qu'on a vus jusqu'à maintenant (entier, réel, caractère, booléen, tableau et chaîne de caractères) sont prédéfinis, c.-à-d. qu'ils existent dans les langages de programmation. Le langage C offre aux programmeurs la possibilité de définir de nouveaux types de données, dits personnalisés.

2. Définition des types en C

En C, le programmeur peut définir ses propres types, et donner les noms qu'il veut à ces types. La déclaration d'un nouveau type doit précéder son utilisation. Pour donner un nom à un type, on utilise `typedef`.

Dans l'exemple suivant, le programmeur a préféré d'utiliser le mot `Entier` au lieu de `int` pour déclarer des entiers :

```
#include <stdio.h>
typedef int Entier; /* Définition d'un nouveau type Entier */
main()
{
  Entier x, y;
  printf("Donnez la première valeur : ") ;
  scanf("%d", &x) ;
  printf("Donnez la deuxième valeur : ") ;
  scanf("%d", &y) ;
  printf("Somme = %d\n", x+y) ;
}
```


Dans ce cas-là, le type `Entier` est tout simplement synonyme de `int`. Nous verrons dans ce qui suit comment définir des types plus compliqués, notamment le type structure.

Remarque : Un nouveau type défini (type personnalisé) ne peut être utilisé que dans le bloc d'instructions là où il a été défini. Dans l'exemple précédent, le nouveau type `Entier` peut être utilisé par tout dans le programme C. Si nous avions, par exemple `main() { typedef int Entier; …}`, le type `Entier` n'aurait pu être utilisé qu'à l'interieur du programme principal.

3. Le type structure

Une structure est un type qui permet de stocker plusieurs données, de même type ou de types différents, dans une même variable. Une structure est composée de plusieurs champs. Chaque champ correspond à une donnée.

A la différence du tableau qui est un type de données complexe homogène, une structure est un type de données complexe hétérogène : complexe parce qu'il y en a plusieurs données ; hétérogène parce que ces données peuvent être de types différents.

Format général :

```
struct <Nom_structure>
{
   <Type_données1>  <champ1>
   <Type_données2>  <champ2>
   …
   <Type_donnéesn>  <champn>
} ;
```

Remarques :

- La définition d'un type structure se termine par un point-virgule après l'accolade fermante.
- L'espace mémoire réservé à une variable de type structure est égal à la somme des espaces réservés pour ses champs.

Exemple :

Voici la déclaration d'une structure `personne` contenant les champs `nom`, `prenom` et `age` :

```
struct personne
{
  char nom[10];
  char prenom[10];
  short age;
};
```

Pour déclarer une variable de type `personne`, on met par exemple :

```
struct personne pere ;
```

Dans ce cas, la variable `pere` prendra 22 octets en mémoire.

Pour éviter de reprendre `struct personne` chaque fois qu'on déclare une variable de ce type, il est possible de mettre :

```
typedef struct personne
 {
  char nom[10];
  char prenom[10];
  short age;
 } Pers;
Pers pere ;
```

Une fois la variable déclarée, on peut la manipuler champ par champ en concaténant le nom de la variable avec le champ désiré de la manière suivante : `pere.nom`, `pere.prenom` et `pere.age`. La composante obtenue se comporte exactement comme toute variable de même type que le champ considéré.

On peut mettre, par exemple : `pere.age = 61 ;`

Remarques :

- Une structure peut être passée comme paramètre d'une fonction ou procédure.
- La seule opération qu'on peut appliquer sur une structure sans passer par ses champs est l'affectation. Par exemple, si nous avions `A` et `B` deux variables

de type `Pers`, nous aurions pu avoir dans notre programme les affectations `A = B` ou `B = A`.

- L'initialisation d'une structure peut être effectuée à la déclaration, par exemple : `Pers pere = {"Ahmed", "Ali", 27};`
- Il était aussi possible de déclarer des variables lors de la définition de la structure, par exemple :

```
struct personne
{
   char nom[10];
   char prenom[10];
   short age;
} pere;
```

Ou même :

```
struct
{
   char nom[10];
   char prenom[10];
   short age;
} pere;
```

Exemple :

```
#include <stdio.h>
typedef struct personne
 {
  char nom[10];
  char prenom[10];
  short age;
 } Pers;
void Afficher(Pers P){
       puts("Voici les informations du père :");
       printf("%s\n%s\n%d", P.nom, P.prenom, P.age);
}
main()
{
  Pers pere ;
  printf("Donnez le nom du père : ") ;
  scanf("%s", pere.nom) ;
  printf("Donnez le prénom du père : ") ;
  scanf("%s", pere.prenom) ;
  printf("Donnez l'âge du père : ") ;
  scanf("%d", &pere.age) ;
  Afficher(pere) ;
}
```

Le programme C précédent permet de lire les informations d'une variable `pere` de type structure `Pers`. Il permet ensuite d'afficher ces informations en utilisant une procédure permettant l'affichage des champs d'une variable de type `Pers` déjà défini.

Remarques :

- Il est possible de déclarer un tableau dont les éléments sont de type structure, par exemple, `Pers T[10]`. L'accès au premier élément du tableau s'effectuera, par exemple, comme suit : `T[0].age = 29`.
- Au niveau de la procédure `Afficher`, il y avait un passage de paramètre par valeur. Il est aussi possible de passer une structure par adresse à une fonction ou procédure, eg. `void Init(Pers *P)`. L'accès à un champ au niveau de cette procédure se fait par `P->age` ou par `(*P).age`, et l'appel au niveau du programme principal se fait par `Init(&pere)`.
  Soit l'exemple :

```
#include <stdio.h>
typedef struct personne
 {
  char nom[10];
  char prenom[10];
  short age;
 } Pers;
void Init(Pers *P){
    P->age = 19; // ou bien     (*P).age = 19;
}
main()
{
  Pers pere ;
  Init(&pere);
  printf("L'âge du père est : %d", pere.age) ;
}
```


Le programme affiche : `L'âge du père est : 19`

4. Le type union

L'union est un type de données qui permet de stocker des données de types différents à la même adresse mémoire (dans le même emplacement ou la même case mémoire).

Une union peut être définie avec plusieurs champs, mais, à un moment donné, il n'y a qu'un seul champ qui peut prendre une valeur.

A la différence d'une structure qui est un type de données complexe hétérogène, l'union est un type simple hétérogène : simple à cause de l'unicité de la valeur ; l'hétérogénéité vient du fait que cette valeur peut être de différents types.

On ne peut manipuler qu'un seul champ à la fois au détriment des autres champs, i.e., une variable de type union ne peut prendre qu'une seule valeur de ses champs à un moment donné.

Le type union a pour objectif de permettre au programmeur d'utiliser une seule variable de différentes manières sans perdre de l'espace mémoire.

Format général en C :

```
union <Nom_union>
{
  <Type_données1>  <champ1>
  <Type_données2>  <champ2>
  …
  <Type_donnéesn>  <champn>
} ;
```

Remarques :

- La définition d'un type union se termine par un point-virgule après l'accolade fermante.
- L'espace mémoire réservé pour une variable de type union est égal à l'espace mémoire réservé pour le champ le plus grand, i.e, le champ qui prend le plus d'espace par rapport aux autres champs.

Exemple :

On peut définir une union dont les éléments peuvent être soit `int`, soit `float`, comme suit :

```
union data
{
  int i;
  float x;
};
```


Pour déclarer une variable de type `union data`, on met par exemple :

```
union data d;
```

La variable `d` prendra 4 octets en mémoire (la taille du type `float`).

Pour éviter de reprendre `union data` chaque fois qu'on déclare une variable de ce type, il est possible de mettre :

```
typedef union data
 {
  int i;
  float x;
 } Data;
Data d ;
```

Une fois la variable déclarée, on peut la manipuler à travers un seul champ à la fois en concaténant le nom de la variable avec le champ désiré de la manière suivante : `d.i` ou `d.x`.

On peut mettre, par exemple : `d.i = 6`. Si on met par la suite `d.x = 89`, la valeur `6` sera perdue.

Remarques :

- Une union peut être passée comme paramètre d'une fonction ou procédure.
- La seule opération qu'on peut appliquer sur une union sans passer par ses champs est l'affectation. Par exemple, si nous avions `A` et `B` deux variables de type `Data`, on aurait pu trouver dans notre programme les affectations : `A = B` ou `B = A`.

Exemple :

```
#include <stdio.h>
   typedef union data
   {  int i;
      float x;
   } Data;
main()
{
  Data d ;
  int choix ;
  printf("Voulez-vous entrer un entier (1) ou un réel (2) : ") ;
  scanf("%d", &choix) ;
  switch (choix)
    { case 1 : scanf("%d", &d.i) ; printf("%d", d.i) ; break ;
      case 2 : scanf("%f", &d.x) ; printf("%f", d.x) ; break ;
      default : printf("Choix incorrect !") ;
    }
}
```

Le programme C précédent permet, à travers un menu affiché, de lire une variable `d` de type `Data`, soit en tant que variable entière, soit en tant que variable réelle. Il affiche ensuite la valeur saisie selon le choix.

5. Le type énumération

Une énumération est un type de données qui permet à une variable de prendre un nombre fini de valeurs. Les valeurs d'un type énumération sont des identificateurs (noms ou symboles). Chaque identificateur correspond à une constante entière. Le type énumération est un type simple homogène.

Format général en C :

```
enum <Nom_enum> {val0, val1, …, valn} ;
```

Remarques :

- La définition d'un type énumération se termine par un point-virgule après l'accolade fermante.
- L'espace mémoire réservé pour une variable de type énumération est généralement égal à l'espace nécessaire pour le stockage d'un entier.

Exemple :

Voici la déclaration d'un type énumération `Animal` représentant différentes sortes d'animaux :

```
enum Animal {lion, elephant, chat, girafe} ;
```

Pour déclarer une variable de type `Animal`, on met par exemple :

```
enum Animal A ;
```

La variable `A` peut prendre comme valeur soit `lion`, `elephant`, `chat` ou `girafe`.

Pour éviter de reprendre `enum Animal` chaque fois qu'on déclare une variable de ce type, il est possible de mettre :

```
typedef enum Animal {lion, elephant, chat, girafe} Anim ;
Anim A ;
```

Remarques :

- Les valeurs d'un type `enum` correspondent par défaut à des constantes entières `0`, `1`, `2`... Donc, `lion` correspond à `0`, `elephant` à `1`, etc. On peut redéfinir ces valeurs tout en gardant de préférence leur ordre croissant, par exemple : `enum Animal {lion, elephant=5, chat, girafe=20} ;` Comme ça, `lion` correspond à `0` (valeur prise par défaut), `elephant` à `5`, `chat` aura `6` (suivant de `5`, son prédécesseur), et `girafe` aura `20`. Dans ce cas, les deux écritures `if (A==chat) {…}` et `if (A==6) {…}` seront équivalentes.
- Une variable de type énumération se comporte comme celle de type entier. On peut faire une affectation (eg. `A = lion` et `A = 0` sont équivalentes), une boucle (eg. `for(A=lion; A<= girafe; A++) {…}`), choix multiple (eg. `switch (A) { case lion: …; case elephant: …; …}`), une comparaison (eg. `(lion<elephant)` retourne `1` pour dire `VRAI`), un affichage (eg. `printf("Lion : %d", lion)` affiche : `0`), une lecture (eg. `scanf("%d", &A)`), un passage en paramètre d'une fonction ou procédure, etc.

Exemple :

```
#include <stdio.h>
typedef enum Animal {lion, elephant, chat, girafe} Anim;
Anim SaisieChoix()
{
 int choix ;
 puts("Choisir un nombre indiquant un animal : 0 (lion), 1 (éléphant), 2 (chat) ou 3 (girafe) :") ;
 scanf("%d", &choix) ;
 switch (choix)
    {
     case lion : return lion ; break ;
     case elephant : return elephant ; break ;
     case chat : return chat ; break ;
     case girafe : return girafe ; break ;
     default : printf("Choix incorrect ! Vous aurez par défaut le choix lion.\n") ;
               return lion ;
    }
  }
main()
{
  Anim A ;
  A = SaisieChoix() ;
  if (A==chat) printf("Animal domestique.") ;
  else printf("Animal sauvage.") ;
}
```

Dans ce programme, nous avons défini un type énumération `Anim` avec un ensemble d'animaux. Nous avons défini une fonction de type `Anim`. La fonction retourne une valeur de type `Anim` selon un nombre saisi au choix. Au niveau du programme principal, nous avons déclaré une variable `A` de type `Anim` qui aura sa valeur selon le nombre saisi au clavier, et selon cette valeur, le programme va nous indiquer s'il s'agit d'un animal sauvage ou domestique.

6. Exercices corrigés

6.1. Exercices

Exercice 1 :

Présentez en C les types structures suivants :

- Un nombre complexe est défini par une partie réelle et une partie imaginaire.
- Une date est composée d'un numéro de jour (1..31), d'un numéro de mois (1..12) et d'une année.
- Un stage peut être défini par un intitulé (une chaîne de caractères), une date de début et une date de fin (deux dates), un nombre de places (entier).
- Une identité décrivant le nom, le prénom et la date de naissance.
- Une fiche bibliographique est définie par le titre du livre (chaîne), les auteurs (10 auteurs maximum, chacun est défini par nom, prénom et date de naissance), la date de parution, l'éditeur (nom, prénom et date de naissance), le numéro ISBN (chaîne).

Comment peut-on affecter une valeur à :

- Une partie réelle d'une variable *X* de type complexe.
- Jour d'une variable *Y* de type date.
- Mois de la date de début d'une variable *Z* de type stage.
- Année de la date de naissance d'une variable *G* de type identité.
- Jour de la date naissance du premier auteur d'une variable *H* de type fiche bibliographique.

Exercice 2 :

Ecrire un programme C permettant de définir un type structure dit `pere` possédant les champs `nom` de type chaîne de caractères, `date_nais` de type chaîne de caractères, `nbr_enf` de type entier, et `liste_enf` de type chaîne de caractères. Le programme doit permettre de lire et d'afficher les informations d'une variable `pere1` de type structure déjà défini.

Exercice 3 :

Reprendre l'exercice 2, mais cette fois-ci en modifiant la structure `pere` comme suit : le champ `date_nais` doit être à son tour défini en tant que type structure possédant les champs : `jour`, `mois`, et `annee`. Le champ `liste_enf` doit être déclaré en tant que tableau de chaînes de caractères.

Exercice 4 :
Reprendre l'exercice 2, mais cette fois-ci en utilisant une procédure qui reçoit en paramètre une variable de type enregistrement (`pere`), puis, elle l'affiche.

Exercice 5 :
En utilisant le type structure `pere` défini dans l'exercice 2, essayez cette fois de déclarer dans un programme C deux variables `pere1` et `pere2` de type `pere`, de lire uniquement le nombre d'enfants des deux pères, et de calculer et d'afficher le nombre total d'enfants.

Exercice 6 :
Ecrire un programme C permettant de déclarer un tableau de cinq éléments de type structure `pere` défini dans l'exercice 2, de lire ses éléments, de les afficher, et enfin d'afficher le nombre total d'enfants.

Exercice 7 :
On considère dix candidats inscrits à une formation diplômante. Chaque candidat va obtenir une note pour cette formation. Ecrire le programme C permettant d'afficher la liste des candidats dont la note est supérieure ou égale à la moyenne des notes de tous les candidats (les noms des candidats et les notes étant lus à partir du clavier). Utilisez un tableau dont les éléments sont de type structure. Le type structure doit contenir un champ indiquant le nom du candidat et un autre indiquant la note du candidat.

Exercice 8 :
Ecrire un programme C qui lit un tableau dont les éléments sont de type structure `point` défini comme suit :

```
struct point {
  int num ;
  float x ;
  float y ;
} ;
```

Le programme doit afficher l'ensemble des informations précédentes. Notons que le programme doit utiliser une procédure pour la lecture et une autre pour l'affichage.

Exercice 9 :
Définir une structure `NombreRationnel` permettant de coder un nombre rationnel, avec numérateur et dénominateur. On écrira une procédure de saisie sans paramètres. Une procédure d'affichage avec un paramètre passé par valeur. On écrira aussi des fonctions de multiplication et d'addition de deux rationnels. Pour l'addition, pour simplifier, on ne cherchera pas nécessairement le plus petit dénominateur commun. Le programme principal doit être capable de lire deux nombres rationnels, de les afficher, et d'afficher leur multiplication et addition.

Exercice 10 :
Une menuiserie industrielle gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur et une épaisseur en millimètres, ainsi que le type de bois qui peut être *pin* (code 0), *chêne* (1) ou *hêtre* (code 2).

1. Définir une structure `Panneau` contenant toutes les informations relatives à un panneau de bois.
2. Ecrire des procédures de saisie et d'affichage d'un panneau de bois.
3. Ecrire une fonction qui calcul le volume en mètres cubes d'un panneau.

Exercice 11 :
Un grossiste en composants électroniques vend quatre types de produits :

- Des cartes mères (code 1) ;
- Des processeurs (code 2) ;
- Des barrettes de mémoire (code 3) ;
- Des cartes graphiques (code 4).

Chaque produit possède une référence (qui est un nombre entier), un prix en dinars et une quantité disponible.

1. Définir une structure `Produit` qui code un produit.
2. Ecrire des procédures de saisie et d'affichage des données d'un produit.
3. Ecrire une procédure `Commande` qui permet à un utilisateur de saisir les données d'un produit (appel de la procédure de saisie). L'utilisateur saisit aussi la quantité commandée du produit. La procédure affiche toutes les données du produit (procédure d'affichage), ainsi que le prix total à payer pour effectuer la commande.
4. Le programme principal fait appel à seulement la procédure `Commande`.

Exercice 12 :
L'université organise un tournoi de tennis de table en double. Chaque équipe engagée se compose de deux joueurs et possède un nom. Chaque joueur est caractérisé par un *nom*, un *prénom*, un *âge*, et un *score* allant de 0 à 100 indiquant son niveau.

1) Définissez les types structures `joueur` et `equipe` correspondants.
2) Ecrivez une procédure `saisir_joueur` permettant de saisir les caractéristiques d'un joueur. Le paramètre doit être passé par adresse.
3) En utilisant la procédure précédente, écrivez une procédure `saisir_equipe` permettant de saisir les caractéristiques d'une équipe. Le paramètre doit être passé par adresse.
4) Ecrivez le programme `main()` qui crée et remplit un tableau de quatre équipes en utilisant la procédure `saisir_equipe`.
5) Ecrivez les procédures `afficher_joueur` et `afficher_equipe` pour afficher à l'écran les équipes et leurs joueurs.

6) Complétez le programme `main()` de manière à afficher les équipes après la saisie.

Exercice 13 :

Une taille peut être exprimée soit en centimètres, soit en mètres. Quel est le type adéquat pour représenter une telle information ? Donnez sa présentation en C.

Exercice 14 :

Comme le type booléen n'existe pas en C, essayez de le définir par un type énumération. Résoudre le problème de la recherche dans un tableau (technique de Flag) en utilisant le type que vous avez défini.

Exercice 15 :

Une entreprise veut stocker dans un tableau ses recettes mensuelles pour une année.

1) Définissez un type énumération `mois` permettant de représenter les 12 mois de l'année.
2) Ecrivez une procédure `Affiche_mois` permettant d'afficher le mot correspondant au numéro du mois passé en paramètre.
3) Ecrivez une procédure `Saisir_recettes` demandant à un utilisateur de remplir le tableau des recettes mensuelles.
4) Ecrivez le programme principal `main()` qui appelle `Saisir_recettes` et affiche la somme des recettes de l'année entière.

Exercice 16 :

Ecrire un programme C permettant de définir un type énumération ayant comme valeurs les jours de la semaine, ensuite de lire une variable entière et de dire si ça correspond à un jour férié dans la liste des valeurs énumérées ou non.

Exercice 17 :

Ecrire un programme C permettant de définir un type énumération ayant comme valeurs les couleurs d'un feu permettant la gestion de la circulation dans un carrefour, ensuite de lire une variable entière et d'afficher à quelle couleur cette valeur correspond.

6.2. Corrigés

Solution 1 :

```
#include <stdio.h>
// Déclaration des types
  typedef struct nombre_complexe{
      float P_reelle ;
      float P_imaginaire ;
  } Nombre_complexe;
  typedef struct date {
      short jour;
      short mois;
      int annee;
```

```
  } Date;
  typedef struct stage {
      char intitule[50];
      Date date_debut, date_fin;
      int nbr_place;
  } Stage;
  typedef struct identite {
      char nom[15];
      char prenom[15];
      Date date_nais;
  } Identite;
  typedef struct fiche_biblio {
      char titre[50] ;
      Identite auteurs[10] ;
      Date date_parution ;
      Identite editeur;
      char num_ISBN[10];
  } Fiche_biblio;
// Déclaration des variables
  Nombre_complexe X ;
  Date Y;
  Stage Z;
  Identite G;
  Fiche_biblio H;
main(){
//Accès aux champs des variables. Les valeurs affectées sont seulement des exemples.
  X.P_reelle = 5 ;
  Y.jour = 22 ;
  Z.date_debut.mois = 1;
  G.date_nais.annee = 1999 ;
  H.auteurs[1].date_nais.jour = 7 ;
}
```


Solution 2 :

```
#include <stdio.h>
  typedef struct pere {
      char nom[20];
      char date_nais[10];
      int nbr_enf;
      char liste_enf[100];
  } Pere;
  Pere pere1 ;
main(){
  printf("Donnez le nom du père : ") ;
  scanf("%s", pere1.nom) ;
  printf("Donnez la date de naissance du père : ") ;
  scanf("%s", pere1.date_nais) ;
  printf("Donnez le nombre d'enfants du père : ") ;
  scanf("%d", &pere1.nbr_enf) ;
  printf("Donnez la liste des enfants du père : ") ;
```

```
  scanf("%s", pere1.liste_enf) ;
  puts("Voici les informations concernant ce père :") ;
  printf("Nom : %s, Date naissance : %s, Nombre d'enfants : %d\n", pere1.nom,
          pere1.date_nais, pere1.nbr_enf);
  printf("Liste des enfants : %s", pere1.liste_enf) ;
}
```

Solution 3 :

```
#include <stdio.h>
  typedef struct date {
      short jour;
      short mois;
      int annee;
  } Date;
  typedef struct pere {
      char nom[20];
      Date date_nais;
      int nbr_enf;
      char liste_enf[20][20];
  } Pere;
    Pere pere1;
  int i;
main(){
  printf("Donnez le nom du père : ") ;
  scanf("%s", pere1.nom) ;
  puts("Donnez la date de naissance du père :") ;
  printf("Jour : ") ;  scanf("%d", &pere1.date_nais.jour) ;
  printf("Mois : ") ;  scanf("%d", &pere1.date_nais.mois) ;
  printf("Année : ") ;  scanf("%d", &pere1.date_nais.annee) ;
  printf("Donnez le nombre d'enfants du père : ") ;
  scanf("%d", &pere1.nbr_enf) ;
  puts("Donnez la liste des enfants :") ;
  for (i=1; i<=pere1.nbr_enf; i++) {
    printf("Donnez le nom de l'enfant num %d : ", i) ;
    scanf("%s", pere1.liste_enf[i]) ;
  }
  puts("Voici les informations concernant ce père :") ;
  printf("Nom : %s, Date naissance : %d/%d/%d, Nombre d'enfants : %d\n",
      pere1.nom, pere1.date_nais.jour, pere1.date_nais.mois,
      pere1.date_nais.annee, pere1.nbr_enf);
  puts("Voici la liste de ses enfants :") ;
  for (i=1; i<=pere1.nbr_enf; i++) printf("%s\n",
pere1.liste_enf[i]) ;
}
```

Solution 4 :

```
#include <stdio.h>
  typedef struct pere {
      char nom[20];
      char date_nais[10];
```

```
      int nbr_enf;
      char liste_enf[100];
  } Pere;
  Pere pere1 ;
  void Afficher(Pere X){
  puts("Voici les informations concernant ce père :") ;
  printf("Nom : %s, Date naissance : %s, Nombre d'enfants : %d\n", X.nom,
          X.date_nais, X.nbr_enf) ;
  printf("Liste des enfants : %s", X.liste_enf) ;
  }
main(){
  printf("Donnez le nom du père : ") ;
  scanf("%s", pere1.nom) ;
  printf("Donnez la date de naissance du père : ") ;
  scanf("%s", pere1.date_nais) ;
  printf("Donnez le nombre d'enfants du père : ") ;
  scanf("%d", &pere1.nbr_enf) ;
  printf("Donnez la liste des enfants du père : ") ;
  scanf("%s", pere1.liste_enf) ;
  Afficher(pere1);
}
```

Solution 5 :

```
#include <stdio.h>
  typedef struct pere {
      char nom[20];
      char date_nais[10];
      int nbr_enf;
      char liste_enf[100];
  } Pere;
  Pere pere1, pere2 ;
  int somme;
main(){
  printf("Donnez le nombre d'enfants du premier père : ") ;
  scanf("%d", &pere1.nbr_enf) ;
  printf("Donnez le nombre d'enfants du deuxième père : ") ;
  scanf("%d", &pere2.nbr_enf) ;
  somme = pere1.nbr_enf + pere2.nbr_enf;
  printf("Nombre total d'enfants est :  %d", somme) ;
}
```

Solution 6 :

```
#include <stdio.h>
  typedef struct pere {
      char nom[20];
      char date_nais[10];
      int nbr_enf;
      char liste_enf[100];
  } Pere;
  Pere peres[5];
  int j, nbr ;
```

```
main(){
  nbr = 0;
  for (j=0; j<=4; j++) {
   printf("Donnez le nom du père num %d : ", j) ;
   scanf("%s", peres[j].nom) ;
   printf("Donnez la date de naissance du père num %d : ", j);
   scanf("%s", peres[j].date_nais) ;
   printf("Donnez le nombre d'enfants du père num %d : ", j) ;
   scanf("%d", &peres[j].nbr_enf) ;
   nbr += peres[j].nbr_enf;
   printf("Donnez la liste des enfants : du père num %d : ", j) ;
   scanf("%s", peres[j].liste_enf) ;
  }
  printf("\nVoici la liste des pères :\n", j) ;
  for (j=0; j<=4; j++) {
   printf("Voici les informations concernant le père num %d\n", j) ;
   printf("Nom : %s\n", peres[j].nom) ;
   printf("La date de naissance : %s\n", peres[j].date_nais) ;
   printf("Le nombre d'enfants : %d\n", peres[j].nbr_enf) ;
   printf("La liste des enfants : %s\n", peres[j].liste_enf) ;
  }
  printf("Le nombre total d'enfants = %d", nbr);
}
```

Solution 7 :

```
#include <stdio.h>
#define max 10
  typedef struct candidat{
      char nom[20];
      float note;
  } Candidat;
  Candidat liste[max];
  float somme, moy;
  int i;
main(){
  somme = 0;
  /* Saisir les noms et les notes des candidats */
  for (i=0; i<=max-1; i++){
    printf("Saisir le nom du candidat n ° %d : ", i) ;
    scanf("%s",liste[i].nom) ;
    printf("Saisir la note du candidat %s : ", liste[i].nom) ;
    scanf("%f", &liste[i].note) ;
    somme += liste[i].note ;
  }
  /* Afficher la moyenne des notes des candidats */
  moy = somme / max ;
  printf("La moyenne des notes des candidats est égale à %f\n", moy) ;
  /* Afficher les candidats dont la note est >= à la moyenne des notes de tous les candidats */
  puts("Voici la liste des candidats dont la note est >= à la moyenne");
  for (i=0; i<=max-1; i++)
```

```
      if (liste[i].note >= moy)
          printf("%s avec une note de %f\n", liste[i].nom, liste[i].note) ;
}
```

Solution 8 :

```
#include <stdio.h>
#define Nbr_P 2
  typedef struct point {
    int num ;
    float x ;
    float y ;
  } Point;
  Point T[Nbr_P];
  int i ;
  void Lire(){
       for(i=0; i<Nbr_P; i++){
              printf("Donnez les informations de l'élément %d :\n", i) ;
              printf("Numéro : "); scanf("%d", &T[i].num);
              printf("x : "); scanf("%f", &T[i].x);
              printf("y : "); scanf("%f", &T[i].y);
         }
  }
  void Afficher(){
       for(i=0; i<Nbr_P; i++){
           printf("Voici les informations de l'élément %d :\n", i) ;
           printf("Numéro : %d,\t x : %f,\t y : %f\n", T[i].num, T[i].x, T[i].y) ;
         }
  }
main(){
  puts("*** Introduire les informations des éléments ***") ;
  Lire();
  puts("*** Voici les informations des éléments ***") ;
  Afficher();
}
```


Solution 9 :

```
#include <stdio.h>
  typedef struct rationnel {
    int numerateur, denominateur ;
  } NombreRationnel;
  NombreRationnel X, Y, M, A;
  void Saisir(){
    puts("Entrez le numérateur et le dénominateur de X :");
    scanf("%d%d", &X.numerateur, &X.denominateur);
    puts("Entrez le numérateur et le dénominateur de Y :");
    scanf("%d%d", &Y.numerateur, &Y.denominateur);
  }
  void Afficher(NombreRationnel n){
    printf("Numérateur :%d\tDénominateur : %d\n",
                  n.numerateur, n.denominateur);
```

```
  }
  NombreRationnel Multiplier(NombreRationnel m, NombreRationnel n){
       NombreRationnel r;
       r.numerateur = m.numerateur * n.numerateur;
       r.denominateur = m.denominateur * n.denominateur;
       return r;
  }
    NombreRationnel Additionner(NombreRationnel m, NombreRationnel n){
       NombreRationnel r;
       r.numerateur = m.numerateur * n.denominateur +
                      n.numerateur * m.denominateur;
       r.denominateur = m.denominateur * n.denominateur;
       return r;
  }
main(){
  puts("********* Introduire X et Y **************") ;
  Saisir();
  puts("********* Afficher X et Y ****************") ;
  Afficher(X);
  Afficher(Y);
  puts("\n**** La multiplication de X par Y donne :");
  M = Multiplier(X, Y);
  printf("\tNumérateur = %d\tDénominateur = %d\n",
            M.numerateur, M.denominateur);
  puts("\n**** L'addition de X et Y donne :");
  A = Additionner(X, Y);
  printf("\tNumérateur = %d\tDénominateur = %d\n",
           A.numerateur, A.denominateur);
}
```


Solution 10 :

```
#include <stdio.h>
  typedef struct panneau {
    float longueur, largeur, epaisseur ;
    short type;
  } Panneau;
  Panneau X;
  void Saisir(){
    printf("Entrez la longueur en millimètres du panneau : ");
    scanf("%f", &X.longueur);
    printf("Entrez la largeur en millimètres du panneau : ");
    scanf("%f", &X.largeur);
    printf("Entrez l'épaisseur en millimètres du panneau : ");
    scanf("%f", &X.epaisseur);
    printf("Entrez le type du panneau 0-pin, 1-chêne ou 2-hêtre : ");
    scanf("%d", &X.type);
  }
  void Afficher(Panneau P){
    printf("La longueur en millimètres du panneau : %f\n", P.longueur);
    printf("La largeur en millimètres du panneau : %f\n", P.largeur);
```

```
    printf("L'épaisseur en millimètres du panneau : %f\n", P.epaisseur);
    printf("Le type du panneau : ");
      switch(P.type){
        case 0: printf("Pin\n"); break;
        case 1: printf("Chêne\n"); break;
        case 2: printf("Hêtre\n"); break;
        default : printf("Type inconnu !\n");
                }
  }
  float Volume (Panneau p){
       return (p.largeur*p.longueur*p.epaisseur)/1e9;
  }
main(){
  puts("********* Introduire un Panneau **************") ;
  Saisir();
  puts("********* Afficher le Panneau ****************") ;
  Afficher(X);
  printf("\n**** Volume en mètres cubes du panneau = %f", Volume(X));
}
```

Solution 11 :

```
#include <stdio.h>
  typedef struct produit {
       short type;
       unsigned int reference;
       float prix;
       unsigned int quantite_dispo;
  } Produit;
  Produit X;
  void Saisir(){
    printf("Entrez le type du produit 1-carte mère, 2-processeur, ");
    printf("3-barrette mémoire ou 4-carte graphique : ");
    scanf("%d", &X.type);
    printf("Entrez la référénce du produit : ");
    scanf("%d", &X.reference);
    printf("Entrez le prix du produit : ");
    scanf("%f", &X.prix);
    printf("Entrez la quantité disponible du produit : ");
    scanf("%d", &X.quantite_dispo);
  }
  void Afficher(Produit P){
       printf("Le type du produit : ");
       switch(P.type){
          case 1: printf("Carte mère\n"); break;
          case 2: printf("Processeur\n"); break;
          case 3: printf("Barrette mémoire\n"); break;
          case 4: printf("Carte graphique\n"); break;
          default : printf("Type inconnu !\n");
       }
       printf("La référence du produit : %d\n", P.reference);
```

```
        printf("Le prix du produit : %f\n", P.prix);
        printf("La quantité du produit : %d\n",P.quantite_dispo);
  }
  void Commande (){
    puts("********* Introduire un produit ****************") ;
    Saisir();
    unsigned int qt;
    printf("Entrez la quantité commandée : ");
    scanf("%d", &qt);
    puts("********* Afficher le produit ****************") ;
    Afficher(X);
    if (qt <= X.quantite_dispo)
        printf("Prix total à payer pour satisfaire la commande = %f", X.prix*qt);
    else printf("Qantité disponible insuffisante pour satisfaire la commande !!");
  }
main(){
  Commande();
}
```

Solution 12 :

```
#include <stdio.h>
#define N 4
typedef struct joueur
  {
    char nom[10];
    char prenom[10];
    short age;
    short niveau;
  } Joueur;
typedef struct equipe
  { char nom[10];
    Joueur joueur1;
    Joueur joueur2;
  } Equipe;
void Saisir_joueur(Joueur *j)
  { printf("Entrez le prénom : "); scanf("%s",j->prenom);
    printf("Entrez le nom : "); scanf("%s",j->nom);
    printf("Entrez l'âge : "); scanf("%d",&j->age);
    printf("Entrez le niveau : "); scanf("%d",&j->niveau);
  }
void Saisir_equipe(Equipe *e)
  { printf("Entrez le nom : ");  scanf("%s",e->nom);
    puts("Joueur 1 :");
    Saisir_joueur(&e->joueur1);
    puts("joueur 2 :");
    Saisir_joueur(&e->joueur2);
  }
void Afficher_joueur(Joueur j)
  { printf("\tPrénom : %s\n",j.prenom);
    printf("\tNom : %s\n",j.nom);
```

```
    printf("\tAge : %d\n",j.age);
    printf("\tNiveau : %d\n",j.niveau);
  }
void Afficher_equipe(Equipe e)
  { printf("Nom : %s\n",e.nom);
    puts("Joueur 1 :");
    Afficher_joueur(e.joueur1);
    puts("Joueur 2 :");
    Afficher_joueur(e.joueur2);
  }
main()
{ Equipe tab[N];
  int i, j;
  for(i=0;i<N;i+=1){
    printf("Equipe %d :\n",i); j=i;
    Saisir_equipe(&tab[j]);
  }
  puts("**** Voici la liste des équipes ****");
  for(i=0;i<N;i++){
    printf("Equipe %d :\n",i);
    Afficher_equipe(tab[i]);
  }
}
```


Solution 13 :

Le type adéquat pour représenter cette information est le type : `union`.

```
union taille
{ short centimetres;
  float metres;
};
```

Solution 14 :

```
#include <stdio.h>
typedef enum BL {FALSE, TRUE}  Boolean;
main()
{
  int T[5] ;
  int NBR ;
  Boolean Flag ;
  int i ;
  printf("Entrez les éléments du tableau :\n");
  for(i=0; i<=4; i++) scanf("%d", &T[i]) ;
  printf("Entrez le nombre à rechercher : ") ;
  scanf("%d", &NBR) ;
  Flag = FALSE ; i = 0;
  while ((i<=9)&& ! Flag) if (T[i]==NBR) Flag = TRUE ;
                          else i++;
  if (Flag) printf("%d fait partie du tableau.", NBR) ;
  else printf("%d ne fait pas partie du tableau.", NBR) ;
}
```

Solution 15 :

```
#include <stdio.h>
typedef enum {jan=1,
        fev,mars,avr,mai,juin,juil,aout,sept,oct,nov,dec } mois;
  void Affiche_mois(mois m)
  { switch(m)
    { case jan: printf("Janvier"); break;
      case fev: printf("Février"); break;
      case mars: printf("Mars"); break;
      case avr: printf("Avril"); break;
      case mai: printf("Mai"); break;
      case juin: printf("Juin"); break;
      case juil: printf("Juillet"); break;
      case aout: printf("Aout"); break;
      case sept: printf("Septembre"); break;
      case oct: printf("Octobre"); break;
      case nov: printf("Novembre"); break;
      case dec: printf("Décembre"); break;
      }
  }
  void Saisir_recettes(int recettes[12])
  {   mois m;
      for(m=jan;m<=dec;m++)
      { printf("Entrez la recette pour ");
        Affiche_mois(m);
        printf(" : ");
        scanf("%d",&recettes[m-1]);
      }
  }
main()
{
  int R[12] , i, somme = 0;
  Saisir_recettes(R);
  for(i=0; i<=11; i++) somme+=R[i];
  printf("La somme des recettes de l'année entière = %d", somme);
}
```


Solution 16 :

```
#include <stdio.h>
  typedef enum jour { Samedi, Dimanche, Lundi, Mardi,
                      Mercredi, Jeudi, Vendredi } Jour;
  int num ;
main(){
  puts("Entrez un numéro de jour entre 0 et 6 :") ;
  scanf("%d", &num) ;
  if ((num == Samedi) || (num == Vendredi))
      printf("C'est un jour férié.");
  else printf("Ce n'est pas un jour férié.") ;
}
```

Solution 17 :

```
#include <stdio.h>
  typedef enum feu { vert, orange, rouge } Feu;
  int num ;
main(){
  puts("Entrez un numéro de jour entre 0 et 2 :") ;
  scanf("%d", &num) ;
  switch (num)
  {
      case vert : puts("Feu vert.");break;
      case orange : puts("Feu orange.");break;
      case rouge : puts("Feu rouge.");break;
      default : puts("Choix incorrect !");
  }
}
```

Chapitre 8 : Les fichiers

1. Introduction

Toutes les structures de données que nous avons utilisées jusqu'à maintenant permettent le stockage temporaire (pendant l'exécution du programme) des variables dans la RAM. Les fichiers permettent le stockage permanent des données.

2. Les fichiers

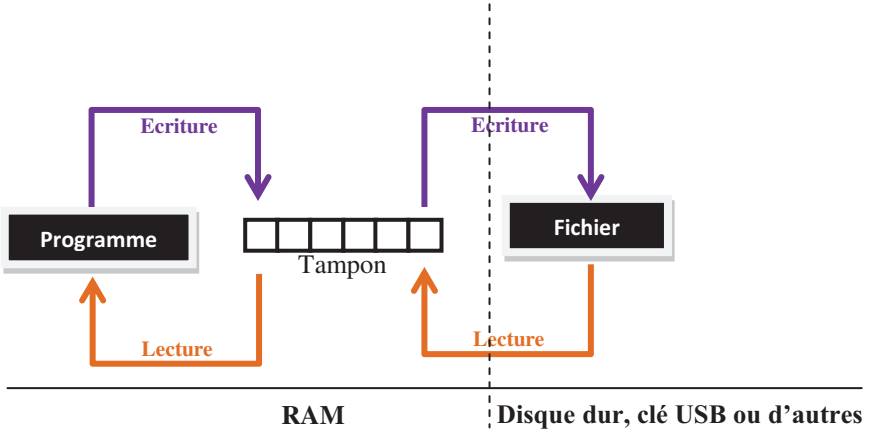
Un fichier est un ensemble d'informations stockées sur un support non volatil (disque dur, clé USB, etc.) pour une durée indéterminée.

Le langage C offre deux façons pour accéder aux fichiers : des fonctions de bas niveau (routines du système d'exploitation) et des fonctions de haut niveau (indépendantes du système). Dans ce qui suit, nous utiliserons les fonctions de haut niveau pour la manipulation des fichiers.

2.1. La structure FILE

L'accès à un fichier en écriture ou en lecture passe par une zone mémoire intermédiaire dite tampon (buffer).

Lors de l'écriture dans un fichier, on écrit d'abord dans le tampon. Quand cette zone est pleine, le système écrit le contenu du tampon dans le fichier, puis on recommence à remplir ce tampon. De même, lors de la lecture, les données lues à partir du fichier sont copiées dans le tampon. On ne lit donc pas directement à partir du fichier, mais plutôt à partir de la mémoire tampon. Cette méthode permet de minimiser le nombre d'accès au fichier, car ces accès sont relativement coûteux, surtout en termes de temps.



Le tampon se trouve au niveau de la mémoire centrale, plus exactement la RAM. Le fichier se trouve bien évidemment au niveau du disque dur ou de la clé USB, ou d'autres supports non volatils.

Pour la manipulation des fichiers en C, on utilise une variable de type structure `FILE`, qui est définie dans `stdio.h`. Cette bibliothèque contient également les fonctions utilisées dans ce chapitre.

Exemple : La déclaration d'un pointeur `fp` vers un fichier se fait comme suit :

```
FILE *fp ;
```

La structure `FILE` permet de désigner un fichier. Elle contient plusieurs champs. On cite en particulier :

- Un pointeur vers l'adresse de début du tampon ;
- Un pointeur vers la position courante dans le tampon ;
- La taille du tampon.

2.2. Ouverture d'un fichier

Pour lire ou écrire dans un fichier, il faut tout d'abord l'ouvrir en utilisant la fonction `fopen` comme suit :

```
FILE *fichier = fopen(nom_fichier, mode);
```

La fonction `fopen` renvoie un pointeur (`fichier`) vers une structure `FILE`, ou bien `NULL` en cas d'erreur. Le paramètre `nom_fichier` est une chaîne de caractères contenant le chemin du fichier à ouvrir, i.e. son emplacement et son nom. Le paramètre `mode` est également une chaîne de caractères qui indique comment le fichier doit être ouvert. Il existe de nombreux modes, mais dans ce cours, nous utiliserons les plus simples :

- `"r"` (read) : lecture seule.
  - ☞ Renvoie `NULL` si le fichier n'existe pas.
- `"w"` (write) : écriture seule.
  - ☞ Si le fichier existe, son contenu sera réinitialisé.
  - ☞ Sinon, il est créé.
- `"a"` (append) : ajout de données à la fin du fichier.
  - ☞ Fichier créé s'il n'existe pas déjà.
  - ☞ Sinon, ça permet d'écrire de nouvelles données après les données existantes.

Exemple : L'exemple suivant permet l'ouverture en lecture d'un fichier appelé `mon_fichier.txt` se trouvant dans le répertoire courant du programme :

```
FILE *fp = fopen("mon_fichier.txt", "r");
if (fp == NULL)
  { // traitement en cas d'erreur
  ...
  }
else
  { // traitement normal
```

```
...
}
```

Voici comment ouvrir un fichier `C:\Mydir\myfile.txt` sous Windows :

```
FILE *fp = fopen("C:\\Mydir\\myfile.txt", "r");
```

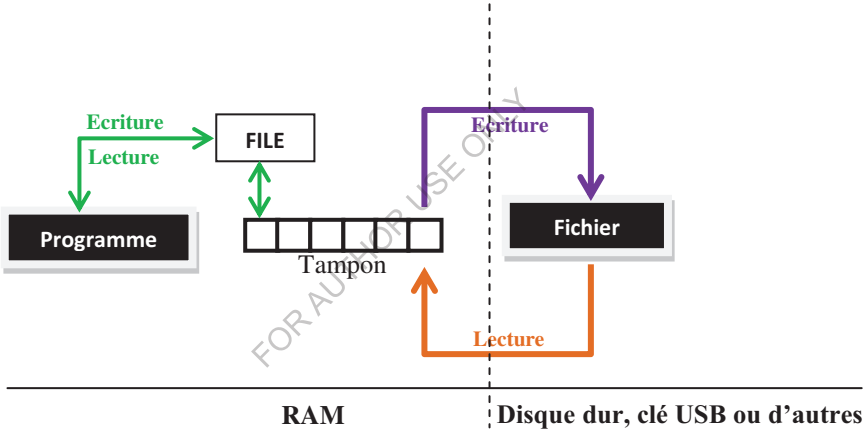
Notons ici que nous avons utilisé le caractère spécial `'\\'` pour représenter le caractère antislash (\), car ce dernier est utilisé en C pour introduire des caractères spéciaux, tels que `'\n'`, `'\p'`, etc.

Sous Unix, on peut trouver, par exemple :

```
FILE *fp = fopen("/home/Mydir/myfile.txt", "r");
```

Une erreur d'ouverture peut se produire, par exemple, en cas de fichier inexistant, ou en cas de droits d'accès limités.

Lorsqu'on ouvre un fichier, on récupère un pointeur vers une variable de type `FILE`, qui est utilisée pour lire ou écrire le fichier.



2.3. Fermeture d'un fichier

A la fin de la manipulation d'un fichier, on le ferme par la fonction `fclose` comme suit :

```
int nom_var = fclose(fichier);
```

`nom_var` est une variable entière. Le paramètre `fichier` est un pointeur vers une structure `FILE` représentant le fichier à fermer. Si la fonction réussit, elle renvoie la valeur `0`. Si elle échoue, elle renvoie la constante `EOF` (End of File, qui a généralement la valeur `-1`). Cette constante est fréquemment utilisée dans les fonctions manipulant des fichiers, pour indiquer qu'un comportement anormal s'est produit.

La fermeture d'un fichier permet de libérer l'espace réservé (tampon) pour manipuler le fichier lors de son ouverture. En cas d'écriture, la fermeture permet aussi d'écrire dans le fichier les données encore présentes dans le

buffer. Donc, en cas d'oubli de `fclose`, certaines données risquent de ne pas être transférées entre la mémoire centrale et le fichier.

Exemple : La fermeture du fichier précédemment ouvert `fp` se fait comme suit:

```
int x = fclose(fp);
if (x == EOF)  { //traitement en cas d'erreur
  …
  }
else  { // traitement normal
  …
  }
```

Remarque : La fonction `fclose` est souvent appelée directement par son nom dans un programme C comme une procédure, sans qu'elle figure dans une expression.

2.4. Accès au fichier

On distingue deux méthodes d'accès au fichier :

1. Accès séquentiel qui consiste à traiter les informations dans l'ordre où elles apparaissent, à partir du premier jusqu'à l'information désirée.
2. Accès direct qui consiste à se placer immédiatement sur l'information désirée, sans avoir à parcourir celles qui la précèdent.

Un fichier peut être manipulé avec un accès séquentiel ou direct, comme on peut combiner les deux méthodes pour un même fichier.

Sur le plan implémentation :

Il existe différentes techniques pour accéder séquentiellement au contenu d'un fichier :

- Accès non-formaté : on lit/écrit des caractères dans des fichiers textes. On distingue deux sortes d'accès non-formaté :
  - ☞ Mode caractère : on lit/écrit un seul caractère à la fois ;
  - ☞ Mode chaîne : on lit/écrit plusieurs caractères (une chaîne de caractères) simultanément.
- Accès formaté : on lit/écrit des données typées dans des fichiers textes.
- Accès par bloc : on manipule des séquences d'octets, indépendamment de leur type. Il s'agit de travailler avec des fichiers binaires.

Dans ce qui suit, on décrit toutes ces manières d'accès aux fichiers, sans oublier l'accès direct.

2.4.1. Lecture/écriture non-formatées en mode caractère

Pour la lecture, on utilise la fonction `fgetc` comme suit :

```
int code_car = fgetc(fichier);
```

Le paramètre `fichier` est un pointeur vers une structure `FILE` représentant un fichier déjà ouvert en lecture. La fonction renvoie le code ASCII du caractère lu, ou bien la constante `EOF` en cas d'erreur.

Exemple : On affiche le contenu d'un fichier déjà ouvert `fp` caractère par caractère comme suit :

```
int c=fgetc(fp);
while(c != EOF){
  putchar(c);
  c=fgetc(fp);
}
```

Fin du fichier

`fgetc` renvoie `EOF` quand on arrive à la fin du fichier. Le problème est que la valeur `EOF` peut aussi être renvoyée à cause d'une autre erreur, par exemple, si l'accès au fichier n'est plus possible.

Pour être sûr qu'on a atteint la fin du fichier, on utilisera la fonction `feof` comme suit :

```
int nom_var = feof(fichier);
```

Cette fonction renvoie une valeur non-nulle si la position courante correspond à la fin du fichier représenté par le pointeur. Sinon, elle renvoie la valeur `0`.

Exemple : Même exemple précédent.

```
int c = fgetc(fp);
while(!feof(fp)){
  putchar(c);
  c = fgetc(fp);
}
```


Pour l'écriture, on utilise la fonction `fputc` comme suit :

```
int nom_var = fputc(code_car, fichier);
```

La fonction `fputc` prend `code_car` de type `int` en paramètre (code ASCII du caractère à écrire, comme on peut mettre directement un caractère). La fonction prend aussi comme paramètre un pointeur vers un fichier déjà ouvert pour écriture. En cas de succès, la fonction renvoie la valeur qui a été écrite. Sinon, elle renvoie `EOF`, par exemple, dans le cas d'un disque plein (saturé).

Exemple : On écrit le caractère `'A'` dans un fichier déjà ouvert en écriture `fp` comme suit :

```
char c = 'A' ;
int x = fputc(c,fp);
if (x == EOF)
  { // traitement de l'erreur
  …
  }
```

2.4.2. Lecture/écriture non-formatées en mode chaîne

Pour la lecture d'une une chaîne de caractères, au lieu de lire un seul caractère à la fois, on utilise la fonction `fgets` comme suit :

```
char *nom_ch = fgets(chaine, nombre, fichier);
```

`chaine` va contenir la chaîne de caractères lue à partir du fichier (ou plus exactement à partir du tampon). `nombre` de type `int` indique le nombre

maximal de caractères à lire. Le pointeur `fichier` désigne le fichier à lire (qui a été préalablement ouvert en lecture, bien entendu).

La fonction renvoie un pointeur vers la chaîne lue (i.e. `&chaine[0]`), ou bien `NULL` en cas de fin de fichier ou en cas d'erreur. La fonction `feof` nous permet d'être sûrs qu'on a atteint la fin du fichier.

La fonction `fgets` tente de lire `nombre-1` caractères du fichier. Elle rajoute automatiquement le caractère `'\0'` à la fin de la chaîne. La lecture est interrompue si la fonction rencontre le caractère `'\n'` (le `'\n'` est quand même recopié dans le tampon).

Exemple : On affiche le contenu d'un fichier déjà ouvert `fp` chaîne par chaîne comme suit :

```
char tmp[12] ;
char *ch =fgets(tmp, 12, fp);
  while(ch != NULL){
     printf("%s",tmp);
     ch =fgets(tmp, 12, fp);
  }
```

Si, par exemple, le fichier contient les deux lignes :

```
bonjour
ahmed
```

Alors, `tmp` va prendre en premier lieu {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\n', '\0'}. Ensuite, elle prendra {'a', 'h', 'm', 'e', 'd', '\0'}.

L'écriture d'une chaîne est effectuée grâce à la fonction `fputs` utilisée comme suit :

```
int nom_var = fputs(chaine, fichier);
```

`chaine` étant la chaîne de caractères à écrire dans le fichier. Le paramètre `fichier` est un pointeur vers un fichier déjà ouvert en écriture. La fonction ne recopie pas le caractère `'\0'` terminal. La fonction renvoie une valeur non-négative en cas de succès, ou bien `EOF` en cas d'erreur.

Exemple : Stocker la chaîne `"bonjour"` dans un fichier déjà ouvert en écriture `fp` comme suit :

```
int x = fputs("bonjour", fp) ;
if (x == EOF) // traiter l'erreur
```

2.4.3. Lecture/écriture formatées

Si on veut formater les données lues ou écrites dans les fichiers textes, i.e. les lire ou les écrire selon leurs types et selon la manière d'organisation, on utilise les fonctions `fscanf` et `fprintf`, dédiées respectivement à la lecture et à l'écriture dans un fichier.

Elles se comportent comme les fonctions `scanf` et `printf`, à la différence du fait qu'un paramètre supplémentaire, placé en première position, désigne le fichier dans lequel on veut lire ou écrire :

```
int nbr_var = fscanf(fichier, format, liste_adresses);
int nbr_var = fprintf(fichier, format, liste_expressions);
```

Ces fonctions renvoient le nombre de variables lues ou écrites, ou bien `EOF` en cas d'erreur (ou de fin de fichier).

Exemple : Lire un `float` à partir d'un fichier ouvert en lecture comme suit :

```
float f ;
int x = fscanf(fp, "%f", &f) ;
if (x == EOF) // traiter l'erreur
```

Notons qu'à chaque lecture par `fscanf`, le pointeur passe automatiquement à la suite dans le fichier.

Exemple : Ecrire un `float` dans un fichier ouvert en écriture comme suit :

```
float f = 12.34 ;
int x = fprintf(fp, "%f", f) ;
if (x == EOF) // traiter l'erreur
```

Remarque : Les deux types d'accès formaté et non-formaté permettent la manipulation des fichiers textes contenant du texte ASCII. Si nous avions, par exemple, un fichier texte contenant `12.340000`, cette information aurait pu être lue comme :

1) la chaîne de caractères `"12.340000"` en accès non-formaté, ou
2) comme le nombre réel `12.34` en accès formaté, comme si nous avions une lecture de chaîne de caractères suivie d'une conversion en réel dans ce deuxième cas.

2.4.4. Lecture/écriture par bloc

Lorsqu'on veut manipuler de grandes quantités de données dans des fichiers binaires, sans s'occuper de leur type ou de leur organisation, on utilise les accès par bloc. Ceci est réalisé grâce aux fonctions `fread` et `fwrite` :

```
size_t  nbr_blocs_lus = fread(&tmp, taille, nombre, fichier);
size_t  nbr_blocs_ecrits = fwrite(&tmp, taille, nombre, fichier);
```

`&tmp` correspond à l'adresse d'un ou plusieurs blocs d'information. `taille` de type `size_t` est la taille d'un seul bloc en octets. `nombre` de type `size_t` est le nombre de blocs à lire ou à écrire. `fichier` est un pointeur vers un fichier. Le type `size_t` est défini dans `stdio.h` ; il s'agit d'un entier non-signé.

La fonction `fread` lit `taille*nombre` octets à partir du fichier, et les recopie dans `tmp`. La fonction `fwrite` lit `taille*nombre` octets à partir de `tmp`, et les recopie dans le fichier. Les deux fonctions renvoient le nombre de blocs lus/écrits : une valeur inférieure à `nombre` indique donc qu'une erreur est survenue au cours du traitement.

Remarques :

- Lorsqu'une variable est écrite dans un fichier binaire, on écrit directement la valeur exacte de la variable, telle qu'elle est codée en binaire en mémoire. Cette manière est plus précise et plus compacte pour stocker des nombres.

Par exemple, l'information 1111.51 a besoin de 7 octets (nombre de caractères) pour la stocker comme chaîne de caractères dans un fichier texte, mais elle n'a besoin que de 4 octets (l'espace nécessaire pour le stockage d'un réel dans la mémoire) pour la stocker dans un fichier binaire.

- Lorsqu'on veut lire ou écrire un tableau dans un fichier en mode par bloc, alors le premier paramètre doit être mis sans `&`, tandis qu'avec une structure, il faudra utiliser `&` pour en obtenir l'adresse. Dans ce dernier cas, il est préférable d'utiliser la fonction `sizeof` pour déterminer avec certitude la taille d'un bloc.

Exemple : Lire par bloc les cinq premiers caractères d'un fichier texte ouvert en lecture, ensuite les afficher :

```
char ch[10];
size_t nbr = fread(ch, 1, 5, fp);
if (nbr != 5) // traiter l'erreur
else printf("%s", ch);
```

2.4.5. Accès direct

Toutes les fonctions de manipulation des fichiers qu'on a vues précédemment permettent un accès séquentiel au fichier. Pour accéder directement à n'importe quel point dans un fichier, il est possible d'utiliser la fonction `fseek` comme suit :

```
int nom_var = fseek(fichier, nbr_octets, mode_action) ;
```


La fonction `fseek` possède comme paramètres :

- Un pointeur vers un fichier (`fichier`).
- Le nombre d'octets après lesquels on désire se positionner (`nbr_octets`) de type `long`. Pour se mettre au début du fichier, cette valeur doit être `0`.
- Le paramètre `mode_action` de type `int` indique la manière d'agir sur le pointeur :
  - ☞ `0` : indique un déplacement à partir du début de fichier.
  - ☞ `1` : indique un déplacement à partir de la position courante du fichier.
  - ☞ `2` : indique un déplacement à partir de la fin de fichier.

`fseek` retourne `0` dans le cas de la réussite du positionnement, et une valeur quelconque dans le cas contraire.

Exemple 1 : Se positionner pour lire et afficher le sixième caractère d'un fichier texte ouvert en lecture :

```
int x = fseek(fp, 5*sizeof(char), 0) ;   int c ;
if (x != 0) // traiter l'erreur
else { c=fgetc(fp);
       putchar(c); }
```

Exemple 2 : Se positionner pour lire le sixième réel à partir d'un fichier binaire là où on a stocké des réels :

```
int x = fseek(fp, 5*sizeof(float), 0) ;
if (x != 0) // traiter l'erreur
```

Remarques :

- Lorsqu'on écrit sur un emplacement qu'on a atteint après `fseek`, la donnée qui existait éventuellement à cet emplacement est effacée et remplacée par la donnée écrite.
- Dans le cas de la création d'un fichier en mode direct, dès qu'on écrit le $n^{ième}$ octet du fichier, il y aura automatiquement la réservation de la place de tous les octets précédents ; leur contenu va être aléatoire.
- Pour avoir la position courante dans le fichier, i.e., après combien d'octets on en est dans le fichier, il est possible d'utiliser la fonction `ftell` de la forme : `long nbr_octets = ftell(fichier) ;`

Remarque importante : Notons que dans ce cours, nous avons essayé de simplifier l'utilisation des fonctions de manipulation des fichiers. C'est pourquoi, nous avons évité de donner la forme originale (en-tête ou prototype) des fonctions dans la bibliothèque `stdio.h`. Par exemple, l'en-tête (prototype) de la fonction `fopen` dans la bibliothèque est le suivant :

```
FILE *fopen(const char *nom_fichier, const char *mode);
```

Alors que son utilisation a été simplifiée dans ce cours comme suit :

```
FILE *fichier = fopen(nom_fichier, mode);
```

Dans cette forme simplifiée, nous avons montré la variable qui va prendre la valeur de retour de `fopen`. Nous avons aussi remplacé les pointeurs par des chaînes de caractères, car à ce niveau, le lecteur n'a pas encore maitrisé la notion de pointeur qui va être étudiée, avec plus de détail, dans le chapitre suivant.

3. Exercices corrigés

3.1. Exercices

Exercice 1 :

Ecrire un programme C permettant de stocker le caractère `'A'` dans un fichier texte. Ecrire un deuxième programme C permettant de lire un caractère à partir du fichier texte précédent.

Exercice 2 :

Ecrire un programme C permettant de lire le contenu d'un fichier texte nommé `fichier.txt`, se trouvant dans le même répertoire que le programme. Pour ce faire, utiliser la lecture non-formatée en mode caractère.

Exercice 3 :

Reprendre le même exercice précédent, mais cette fois-ci en utilisant la lecture non-formatée en mode chaîne.

Exercice 4 :

Ecrire un programme C permettant de créer un fichier texte nommé `fichier.txt`, stockant une suite de cinq caractères lus à partir du clavier. Utiliser l'écriture non-formatée en mode caractère.

Exercice 5 :
Ecrire un programme C permettant de créer un fichier texte nommé `fichier.txt`, stockant une suite de cinq chaînes de caractères lues à partir du clavier. Utiliser l'écriture non-formatée en mode chaîne de caractères.

Exercice 6 :
Ecrire un programme C permettant de créer un fichier texte nommé `fichier.txt`, stockant une chaîne de caractères lue à partir du clavier. Utiliser l'écriture non-formatée en mode caractère.

Exercice 7 :
Ecrivez un programme C qui réalise une copie d'un fichier existant `fichier.txt` par lecture/écriture en mode caractère.

Exercice 8 :
Par un menu de choix multiple, écrire un programme C permettant de demander à l'utilisateur de lire un fichier texte (`fichier.txt`), de créer un nouveau fichier texte (`fichier.txt`) ou d'ajouter des données à un fichier existant (`fichier.txt`). Chaque choix doit être assuré par une procédure. Pour ce faire, utiliser lecture/écriture non-formatées en mode chaîne de caractères.

Exercice 9 :
Ecrire un programme C permettant de lire un réel à partir du clavier, et le stocker ensuite dans un fichier `reels.txt`. Utiliser le mode écriture formatée.

Exercice 10 :
Ecrire un programme C permettant d'afficher un nombre réel à partir du fichier `reels.txt`. Utiliser le mode lecture formatée.

Exercice 11 :
En utilisant le mode lecture/écriture formatées, on veut stocker les informations des employés d'une entreprise dans un fichier `employers.txt`. Chaque employé est identifié par un *numéro* de type entier, un *nom* et *prénom* de type chaîne de caractères. Les informations étant lues à partir du clavier. Après le stockage de ces informations, essayez de les afficher à nouveau à partir du fichier pour confirmer le stockage.

Exercice 12 :
Utilisez l'écriture par bloc pour stocker la chaîne de caractères `"bonjour"` dans un fichier `bloc.txt`.

Exercice 13 :
Utilisez la lecture par bloc pour lire un bloc de sept caractères à partir du fichier `bloc.txt` créé dans l'exercice précédent.

Exercice 14 :
Ecrire un programme C permettant d'enregistrer séquentiellement dans un fichier, dont le nom est lu à partir du clavier, une suite de nombres entiers qu'on lui fournit au clavier. Utiliser l'écriture par bloc.

Exercice 15 :
Ecrire un programme C permettant de lire un fichier créé par le programme de l'exercice précédent. Le nom du fichier étant lu à partir du clavier. Utiliser la lecture par bloc.

Exercice 16 :
Reprendre l'exercice 11, mais cette fois-ci en utilisant lecture/écriture par bloc.

Exercice 17 :
Avec la même structure d'un employé définie dans l'exercice 11, et en utilisant lecture/écriture par bloc, on veut stocker les informations des employés d'une entreprise dans un fichier binaire. Les informations étant lues à partir du clavier. A partir de ce fichier, essayez de stocker ces informations dans un tableau de structure, ensuite les afficher à nouveau à partir de ce tableau pour confirmer le stockage.

Exercice 18 :
Avec la même structure d'un employé définie dans l'exercice 11, et en utilisant lecture/écriture par bloc, on veut stocker les informations de cinq employés d'une entreprise dans un tableau lu à partir du clavier. Le tableau doit être ensuite stocké dans un fichier binaire. Le contenu du fichier doit être enfin affiché.

Exercice 19 :
Ecrire un programme C permettant, à partir du fichier créé par l'exercice 16, de retrouver les informations correspondant à un employé dont le nom est lu au clavier. Il vous faudra une recherche séquentielle.

Exercice 20 :
Ecrire un programme C permettant de lire n'importe quel entier du fichier créé par le programme de l'exercice 14. La position du nombre à lire étant lue à partir du clavier. Utiliser l'accès direct.

3.2. Corrigés

Solution 1 :

Programme 1 :

```
/* Programme C permettant de stocker le caractère 'A' dans un
fichier texte */
#include <stdio.h>
main()
{ char c; int x;
  FILE *fp = fopen("d:\\fichier.txt", "w");
  if (fp != NULL){
    c= 'A';
    x = fputc(c,fp);
    if(x == EOF) printf("Echec de stockage du caractère %c dans fichier.txt !\n", c);
    else printf("Stockage réussi du caractère %c dans fichier.txt.\n", c);
    fclose(fp);
```

```
  } else printf("Echec de création !");
}
```

Programme 2:

```
/* Programme C permettant de lire un caractère à partir du
fichier texte précédent */
#include <stdio.h>
main()
{
  int x;
  FILE *fp = fopen("d:\\fichier.txt", "r");
  if (fp != NULL){
    x = fgetc(fp);
    if(x == EOF) printf("Echec de lecture du caractère à partir du fichier.txt !\n");
      else printf("Lecture du caractère %c avec succès à partir du fichier.txt\n", x);
    fclose(fp);
  } else printf("Echec d'ouverture !");
}
```

Solution 2 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("fichier.txt", "r");
  if (fp != NULL){
      int c=fgetc(fp);
      while(c != EOF){       // On peut mettre aussi  while(!feof(fp))
        putchar(c);
        c=fgetc(fp);
      }
      fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```


Solution 3 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("fichier.txt", "r");
  if (fp != NULL){
    char tmp [12];
    char *ch =fgets(tmp, 12, fp);
    while(ch != NULL){
       printf("%s",tmp);
       ch =fgets(tmp, 12, fp);
    }
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```

Ou bien :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("fichier.txt", "r");
  if (fp != NULL){
    char tmp [12];
    char *ch =fgets(tmp, 12, fp);
    while(! feof(fp)){
       printf("%s",tmp);
       ch =fgets(tmp, 12, fp);
    }
    printf("%s",tmp); /* Ajoute cette instruction pour que tu puisses
                         afficher la dernière ligne du fichier */
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```

Solution 4 :

```
#include <stdio.h>
main()
{
  FILE * fp = fopen("fichier.txt", "w");
  int i; char c; int x;
  for (i=1; i<=5; i++) {
    printf("Donnez un caractère : ");
    c = getchar(); // ou bien      scanf("%c", &c);
    getchar();
    x = fputc(c,fp);
    if(x == EOF) printf("Echec de stockage du caractère %c dans fichier.txt !\n", c);
    else printf("Stockage réussi du caractère %c dans fichier.txt.\n", c);
  }
  fclose(fp);
}
```


Solution 5 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("fichier.txt", "w");
  int i; char ch[10]; int x;
  for (i=1; i<=5; i++) {
    printf("Donnez une chaîne de caractères : ");
    gets(ch);        // ou bien    scanf("%s", ch);
    x = fputs(ch,fp);
    if(x == EOF)
    printf("Echec de stockage de la chaîne de caractères %s dans fichier.txt !\n", ch);
    else printf("Stockage réussi de la chaîne de caractères %s dans fichier.txt.\n", ch);
  }
```

```
  fclose(fp);
}
```

Solution 6 :

```
#include <stdio.h>
#include <string.h>
main()
{
  FILE *fp = fopen("fichier.txt", "w");
  int i; char ch[10]; int x;
  printf("Donnez une chaîne de caractères à stocker dans fichier.txt : ");
  gets(ch);
  for (i=0; i<strlen(ch); i++) {
    x = fputc(ch[i],fp);
    if(x == EOF)
     printf("Echec de stockage du caractère %c dans fichier.txt !\n", ch[i]);
  }
  fclose(fp);
}
```

Solution 7 :

```
#include <stdio.h>
main()
{
  // On ouvre le fichier source en lecture
  FILE *source = fopen("fichier.txt", "r");
  if(source != NULL){
    // On ouvre le fichier cible en écriture
    FILE *cible = fopen("copie.txt", "w");
    /* On lit la source caractère par caractère
       en écrivant chaque caractère dans la cible */
    int c=fgetc(source); int x; int flag = 1;
    while(c != EOF){
      x = fputc(c,cible);
      if(x == EOF) {
       printf("Echec de stockage du caractère %c dans copie.txt !\n", c);
       flag = 0;
      }
      c=fgetc(source);
    }
  if (flag) printf("Copiage réussi de fichier.txt en copie.txt.\n");
  else printf("Echec de Copiage !\n");
  fclose(cible);
  fclose(source);
  }
  else printf("Impossible d'ouvrir fichier.txt !\n");
}
```


Solution 8 :

```
#include <stdio.h>
void Lire_fichier()
```

```
{
  FILE *fp = fopen("fichier.txt", "r");
  if (fp != NULL){
    char tmp [12];
    char *ch =fgets(tmp, 12, fp);
    while(! feof(fp)){
       printf("%s",tmp);
       ch =fgets(tmp, 12, fp);
    }
    printf("%s",tmp); /* Ajoute cette instruction pour que tu puisses afficher
                          la dernière ligne du fichier */
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
void Creer_fichier()
{
  FILE *fp = fopen("fichier.txt", "w");
  int i; char ch[10]; int x; char c;
  do {
    printf("Donnez une chaîne de caractères : ");
    scanf("%s", ch);
    x = fputs(ch,fp);
    if(x == EOF)
      printf("Echec de stockage de la chaîne de caractères %s dans fichier.txt !\n", ch);
    else printf("Stockage réussi de la chaîne de caractères %s dans fichier.txt.\n", ch);
    printf("Voulez-vous ajouter une autre chaîne o/n ? ");
    getchar();
    c=getchar(); getchar();
  } while(c != 'n');
  fclose(fp);
}
void Ajouter_fichier()
{
  FILE *fp = fopen("fichier.txt", "a");
  int i; char ch[10]; int x; char c;
  do {
    printf("Donnez une chaîne de caractères : ");
    scanf("%s", ch);
    x = fputs(ch,fp);
    if(x == EOF)
      printf("Echec de stockage de la chaîne de caractères %s dans fichier.txt !\n", ch);
    else printf("Stockage réussi de la chaîne de caractères %s dans fichier.txt.\n", ch);
    printf("Voulez-vous ajouter une autre chaîne o/n ? ");
    getchar();
    c=getchar(); getchar();
  } while(c != 'n');
  fclose(fp);
}
```

```
main()
{
       int choix;
       do{
          printf("\nDonnez votre choix : \n\t1-Lire fichier, \n\t2-Créer fichier,");
          printf("\n\t3-Ajouter à un fichier et \n\t4-Quitter le programme\n");
          scanf("%d", &choix);
          switch(choix){
              case 1: Lire_fichier(); break;
              case 2: Creer_fichier(); break;
              case 3: Ajouter_fichier(); break;
              case 4: break;
              default : printf("Choix incorrect !!");
            }
       } while(choix != 4);
}
```

Solution 9 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("reel.txt", "w");
  float f ;
  printf("Donnez un nombre réel : ");
  scanf("%f", &f);
  int x = fprintf(fp, "%f", f);
  if(x != EOF) printf("Le stockage a été fait avec succès.\n");
  else printf("Echec de stockage dans le fichier !\n");
  fclose(fp);
}
```


Solution 10 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("reel.txt", "r");
  float f;
  int x = fscanf(fp, "%f", &f) ;
  if(x != EOF) printf("Le nombre lu à partir du fichier est : %f\n", f) ;
  else printf("Echec de lecture à partir du fichier !\n");
  fclose(fp);
}
```

Solution 11 :

```
#include <stdio.h>
typedef struct employe{
       int num;
       char nom[10], prenom[10];
} Employe;
main()
{
```

```
  Employe e ; char c; int x;
  // Stockage (écriture) des employés
  FILE *fp = fopen("employes.txt", "w");
  do{
   printf("Donnez le numéro de l'employé : ");
   scanf("%d", &e.num);
   printf("Donnez le nom de l'employé : ");
   scanf("%s", e.nom);
   printf("Donnez le prénom de l'employé : ");
   scanf("%s", e.prenom);
   x = fprintf(fp, "%d\n%s\n%s\n", e.num, e.nom, e.prenom);
   if(x != EOF) printf("Le stockage a été fait avec succès.\n");
   else printf("Echec de stockage dans le fichier !\n");
   printf("Voulez-vous ajouter un autre employé o/n ? ");
   getchar();
   c=getchar(); getchar();
  } while(c != 'n');
  fclose(fp);
  // Affichage (lecture) des employés
  fp = fopen("employes.txt", "r");
  x = fscanf(fp, "%d\n%s\n%s\n", &e.num, e.nom, e.prenom) ;
  if (x != EOF) printf("Voici la liste des employés stockés dans le fichier :\n");
  while(x != EOF){
    printf("\t%d %s %s\n", e.num, e.nom, e.prenom) ;
    x = fscanf(fp, "%d\n%s\n%s\n", &e.num, e.nom, e.prenom) ;
  }
  fclose(fp);
}
```


Solution 12 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("bloc.txt", "w");
  size_t nbe = fwrite("bonjour", 7, 1, fp);
  if(nbe == 1) printf("Stockage avec succès.\n") ;
  else printf("Echec de stockage dans le fichier bloc.txt !\n");
  fclose(fp);
}
```

Solution 13 :

```
#include <stdio.h>
main()
{
  FILE *fp = fopen("bloc.txt", "r");
  char ch[8];
  size_t nbl = fread(ch, 7, 1, fp);
  if(nbl == 1) printf("La chaîne lue est : %s.\n", ch) ;
  else printf("Echec de lecture à partir du fichier bloc.txt !\n");
  fclose(fp);
}
```

Solution 14 :

```
#include <stdio.h>
main()
{
  int x ; char nomfichier[10]; char c; size_t nbe, nbl;
  FILE * fp;
  // Stockage (écriture) des entiers
  printf("Donnez le nom du fichier à créer : ");
  scanf("%s", nomfichier);
  fp = fopen(nomfichier, "w");
  if (fp != NULL){
    do{
     printf("Donnez un entier : ");
     scanf("%d", &x);
     nbe = fwrite(&x, sizeof(x), 1, fp);
     if(nbe == 1) printf("Stockage avec succès.\n") ;
     else printf("Echec de stockage dans le fichier !\n");
     printf("Voulez-vous ajouter un autre nombre o/n ? ");
     getchar();
     c=getchar(); getchar();
    } while(c != 'n');
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```

Solution 15 :

```
#include <stdio.h>
main()
{
  int x ; char nomfichier[10]; size_t nbl; FILE * fp;
  // Affichage (lecture) des entiers
  printf("Donnez le nom du fichier à lire : ");
  scanf("%s", nomfichier);
  fp = fopen(nomfichier, "r");
  if (fp != NULL){
    puts("Voici la liste des entiers stockés dans le fichier :");
    nbl = fread(&x, sizeof(x), 1, fp);
    if(nbl == 1) printf("\t%d\n", x) ;
    while(!feof(fp) && nbl){
      nbl = fread(&x, sizeof(x), 1, fp);
      if(nbl == 1) printf("\t%d\n", x) ;
    }
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```

Solution 16 :

```
#include <stdio.h>
typedef struct employe{
```

```
       int num;
       char nom[10], prenom[10];
} Employe;
main()
{
  Employe e ; char c; size_t nbe, nbl; FILE * fp;
  // Stockage (écriture) des employés
  fp = fopen("employes.blc", "w");
  do{
   printf("Donnez le numéro de l'employé : ");
   scanf("%d", &e.num);
   printf("Donnez le nom de l'employé : ");
   scanf("%s", e.nom);
   printf("Donnez le prénom de l'employé : ");
   scanf("%s", e.prenom);
   nbe = fwrite(&e, sizeof(e), 1, fp);
   if(nbe == 1) printf("Stockage avec succès.\n") ;
   else printf("Echec de stockage dans employes.blc !\n");
   printf("Voulez-vous ajouter un autre employé o/n ? ");
   getchar();
   c=getchar(); getchar();
  } while(c != 'n');
  fclose(fp);
  // Affichage (lecture) des employés
  fp = fopen("employes.blc", "r");
  puts("Voici la liste des employés stockés dans le fichier :");
  nbl = fread(&e, sizeof(e), 1, fp);
  if(nbl == 1) printf("\t%d %s %s\n", e.num, e.nom, e.prenom) ;
  while(!feof(fp) && nbl){
    nbl = fread(&e, sizeof(e), 1, fp);
    if(nbl == 1) printf("\t%d %s %s\n", e.num, e.nom, e.prenom) ;
  }
  fclose(fp);
}
```


Solution 17 :

```
#include <stdio.h>
#include <string.h>
  typedef struct employe{
       int num;
       char nom[10], prenom[10];
  } Employe;
main()
{
  Employe e ; char c; size_t nbe, nbl; FILE *fp;
  Employe Employes[100]; int i, j;
  // Stockage (écriture) des employés
  fp = fopen("employes.blc", "w");
  do{
   printf("Donnez le numéro de l'employé : ");
```

```
   scanf("%d", &e.num);
   printf("Donnez le nom de l'employé : ");
   scanf("%s", e.nom);
   printf("Donnez le prénom de l'employé : ");
   scanf("%s", e.prenom);
   nbe = fwrite(&e, sizeof(e), 1, fp);
   if(nbe == 1) printf("Stockage avec succès.\n") ;
   else printf("Echec de stockage dans employes.blc !\n");
   printf("Voulez-vous ajouter un autre employé o/n ? ");
   getchar();
   c=getchar(); getchar();
  } while(c != 'n');
  fclose(fp);
  // Stockage des employés dans un tableau de structure
  fp = fopen("employes.blc", "r");
  i = 0;
  nbl = fread(&e, sizeof(e), 1, fp);
  if(nbl == 1) {
       Employes[i].num= e.num;
       strcpy(Employes[i].nom, e.nom);
       strcpy(Employes[i].prenom, e.prenom) ;
       i++;
  }
  while(!feof(fp) && nbl){
    nbl = fread(&e, sizeof(e), 1, fp);
    if(nbl == 1) {
       Employes[i].num= e.num;
       strcpy(Employes[i].nom, e.nom);
       strcpy(Employes[i].prenom, e.prenom) ;
       i++;
    }
  }
  fclose(fp);
  // Affichage des employés à partir de ce tableau
  if (i != 0) puts("Voici la liste des employés stockés dans le fichier :");
  for(j=0; j<i; j++)
   printf("\t%d %s %s\n", Employes[j].num, Employes[j].nom, Employes[j].prenom) ;
}
```


Solution 18 :

```
#include <stdio.h>
typedef struct employe{
       int num;
       char nom[10], prenom[10];
} Employe;
Employe T[5];
main()
{
  FILE *fp; int i;
  fp = fopen("employes.blc", "w");
```

```
  for (i=0; i<5; i++){
    printf("Donnez le numéro de l'employé %d : ", i);
    scanf("%d", &T[i].num);
    printf("Donnez le nom de l'employé %d : ", i);
    scanf("%s", T[i].nom);
    printf("Donnez le prénom de l'employé %d : ", i);
    scanf("%s", T[i].prenom);
  }
  // Stockage (écriture) du tableau des employés dans un fichier binaire
  fwrite(T, sizeof(Employe), 5, fp);
  fclose(fp);
  // Affichage (lecture) du fichier des employés
  fp = fopen("employes.blc", "r");
  fread(T, sizeof(Employe), 5, fp);
  for(i=0; i<5; i++){
    printf("Voici les informations de l'employé %d :\n", i);
    printf("\tNum : %d, Nom : %s, Prenom : %s\n", T[i].num, T[i].nom, T[i].prenom) ;
  }
  fclose(fp);
}
```

Solution 19 :

```
#include <stdio.h>
typedef struct employe{
       int num;
       char nom[10], prenom[10];
} Employe;
main()
{
  Employe e ; char nom[10]; size_t nbl; FILE *fp; int existe;
  // Affichage (lecture) des employés
  fp = fopen("employes.blc", "r");
  if(fp != NULL){
  printf("Donnez le nom de l'employé que vous cherchez : ");
  gets(nom);
  existe = 0;
  nbl = fread(&e, sizeof(e), 1, fp);
  if(nbl == 1) if (strcmp(e.nom, nom) == 0) {
     puts("Voici les informations de l'élément recherché :");
     printf("\tNum : %d, Nom : %s, Prenom : %s\n", e.num, e.nom, e.prenom) ;
     existe = 1;
  }
  while(!feof(fp) && nbl && !existe){
    nbl = fread(&e, sizeof(e), 1, fp);
    if(nbl == 1) if (strcmp(e.nom, nom) == 0) {
      puts("Voici les informations de l'élément recherché :");
      printf("\tNum : %d, Nom : %s, Prénom : %s\n", e.num, e.nom, e.prenom) ;
      existe = 1;
    }
  }
```

```
  if (! existe) printf("Elément introuvable !\n");
  fclose(fp);
  } else printf("Impossible d'ouvrir le fichier !\n");
}
```

Solution 20 :

```
#include <stdio.h>
main()
{
  int x, y, position ; char nomfichier[10];  size_t nbl;
  FILE *fp;
  // Affichage (lecture) d'un entier
  printf("Donnez le nom du fichier à ouvrir : ");
  scanf("%s", nomfichier);
  printf("Donnez la position de l'entier à lire : ");
  scanf("%d", &position);
  fp = fopen(nomfichier, "r");
  if (fp != NULL){
    y = fseek(fp, (position - 1)* sizeof(int),0);
    if (y == 0){
      nbl = fread(&x, sizeof(x), 1, fp);
      if(nbl == 1) printf("Entier : %d\n", x) ;
    } else printf("Echec de positionnement !\n");
    fclose(fp);
  }
  else printf("Impossible d'ouvrir le fichier !\n");
}
```

Chapitre 9 : Les listes chaînées

1. Introduction

Pour stocker un ensemble d'éléments de même type, il est possible d'utiliser les tableaux. Cependant, le nombre d'éléments du tableau doit être fixé lors de sa déclaration. Les tableaux ne peuvent donc pas être étendus pendant l'exécution du programme. Les listes chaînées peuvent être étendues pendant l'exécution du programme par l'allocation (réservation) d'un nouvel espace mémoire quand c'est nécessaire. Elles peuvent être aussi réduites par une désallocation de l'espace non utile. La construction d'une liste chaînée consiste à regrouper un ensemble d'objets éparpillés en mémoire et liés par des pointeurs.

2. Les pointeurs

Un pointeur est une variable qui contient une adresse mémoire, et non pas la valeur. C'est une variable qui au lieu de contenir la donnée, contient son adresse en mémoire.

Format général : `<type de données> *Nom pointeur ;`

Exemple :

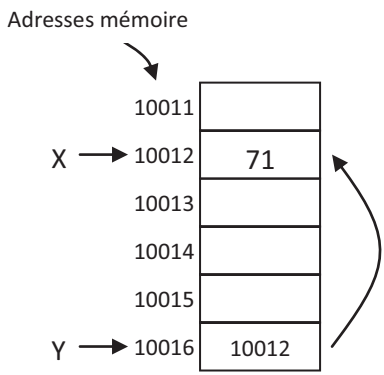
Pour déclarer un pointeur `Y` vers un entier (contient l'adresse d'un entier), on met :

```
Variables
  Entier *Y ;
```

En C, un pointeur est déclaré comme suit :

```
  int *Y ;
```

Si nous avions `X` une variable entière ayant comme valeur `71`, et `Y` un pointeur vers la variable entière `X`, les deux variables `X` et `Y` auraient pu être représentées en mémoire comme suit :



En fait, cette représentation n'est pas tout à fait exacte, car réellement la mémoire est constituée d'un ensemble d'octets, chacun est identifié par une

adresse. La case mémoire correspondant à une variable est constituée d'un ensemble d'octets selon son type. L'adresse d'une variable est l'adresse du premier octet de l'espace mémoire réservé (alloué) à la variable.

Un pointeur peut recevoir la valeur d'un autre pointeur par une opération d'affectation. On peut aussi comparer deux pointeurs pointant vers des objets de même type par les opérateurs de comparaison (=, >, <, >=, <=, <>).

On peut additionner ou soustraire un entier d'un pointeur ; le résultat sera un pointeur. Par conséquent, un pointeur peut être incrémenté ou décrémenté. On peut faire la différence de deux pointeurs ; on aura un entier. Les autres opérations arithmétiques (somme, produit…) ne sont pas acceptées.

L'espace mémoire réservé à une variable de type pointeur dépend du système utilisé. Il peut varier en fonction du matériel et du système d'exploitation. Ça peut être 2 octets, 4 ou même 8 octets.

Les pointeurs mènent à une gestion dynamique de la mémoire.

3. Gestion dynamique de la mémoire

Quand on utilise un tableau dans un programme, on doit fixer sa taille avant l'exécution, et on parle d'une gestion statique de la mémoire. La gestion dynamique permet la réservation (allocation) de l'espace pendant l'exécution du programme grâce à des variables de type pointeur.

Une variable de type pointeur contient l'adresse d'une variable dite *variable pointée*.

Pour la gestion dynamique de la mémoire, on utilise principalement les deux procédures :

- `Allouer(p)` qui permet de réserver un espace mémoire pour une variable pointée par le pointeur `p`. En C, c'est la fonction `malloc(nombre_octets)`, mais on écrit d'une façon plus explicite et sûre : `p=(<type_données>*)malloc(sizeof(<type_données>));`
  `<type_données>` étant le type d'élément vers lequel pointe `p`. La fonction `malloc` retourne l'adresse du premier octet réservé.
  `malloc` est à l'origine de type `void*`, i.e., `malloc` retourne une adresse, mais qu'on ne sait pas sur quel type elle pointe. C'est pourquoi, on convertira ce `void*` en un type plus explicite, tel que `int*`, `float*`, etc., en fonction du besoin, et on met (`<type_données>*`).
  Pour éviter les erreurs lorsqu'on exprime la taille de la mémoire à allouer, on utilisera la fonction `sizeof`, qui renvoie la taille en octets d'une donnée du type spécifié en paramètre.
- `Désallouer(p)` qui permet de libérer l'espace mémoire réservé à une variable pointée par le pointeur `p`. En C, c'est `free(p)`. La fonction `free` est souvent appelée directement par son nom dans un programme C comme une procédure, sans qu'elle figure dans une expression.

Les deux fonctions, `malloc` et `free`, appartiennent à la bibliothèque `stdlib.h`.

Exemple :

```
Algorithme pointeurs
Variables
  Entier *y;
Début
  Allouer(y) ;
  *y ← 7 ;
  Ecrire("La valeur est : ", *y) ;
  Désallouer(y) ;
Fin
```

En C :

```
#include<stdio.h>
#include<stdlib.h>
main()
{ int *y ;
  y = (int*)malloc(sizeof(int));
  *y = 7 ;
  printf("La valeur est : %d\n", *y) ;
  free(y) ;
}
```

Le programme C précédent permet de créer une variable `y` de type pointeur vers un entier. Il alloue un espace pour la variable pointée. Ensuite, il affecte la valeur `7` à la variable pointée. Il affiche cette valeur. Enfin, il libère l'espace réservé à la variable pointée.

Voyons ce que fait l'exemple C suivant :

```
#include<stdio.h>
main()
{ int *y;  int x;
  x = 7;
  y = &x ;  printf("*y = %d\n", *y) ;
  *y = 9 ;  printf("x = %d\n", x) ;
  scanf("%d", &x) ; printf("x = %d\n", x) ;
  scanf("%d", y) ; printf("x = %d\n", x) ;
}
```

Après avoir déclaré un entier `x` et un pointeur vers un entier `y`, le programme C précédent affecte la valeur `7` à la variable `x`. Il affecte l'adresse de la variable `x`, indiquée par `&`, à la variable `y`. Il affiche la valeur de la variable pointée par `y` (`*y = 7`). La variable pointée par `y` aura par la suite `9`. La valeur de `x` sera donc `9`, et le programme l'affiche : `x = 9`. Le programme va encore lire au clavier la valeur de `x` et l'afficher. Ces deux dernières opérations seront répétées deux fois.

L'opérateur de référençage `&`, aussi appelé opérateur d'adressage, permet de connaitre l'adresse d'une variable. Il faut le placer devant la variable considérée. Par exemple, supposons qu'on a une variable `x` déjà déclarée. Alors, l'expression `&x` correspond à l'adresse de cette variable.

Voyons encore un autre exemple :

```
#include<stdio.h>
main()
{ int *y = 7;
   printf("*y = %d\n", *y) ;
   printf("y = %d\n", y) ;
   printf("&y = %d\n", &y) ;
}
```

Cet exemple affichera la valeur de la variable pointée (`*y = 7`), l'adresse de la variable pointée, i.e. le contenu du pointeur (`y = …`), et l'adresse du pointeur, qui est une adresse d'une adresse (`&y = …`).

Remarques :

- Pour dire qu'un pointeur ne pointe vers aucun élément, il faut lui affecter la valeur `NULL` de bibliothèque `stdlib.h`. Une fois créé, un pointeur pointe vers n'importe quel octet de la mémoire. Il est donc recommandé d'initialiser ce pointeur à `NULL`, sinon on risque d'avoir une erreur d'exécution.
- Les deux fonctions `malloc` et `free` utilisent une zone mémoire appelée TAS (en anglais Heap). Quand cette zone est pleine, on aura une erreur d'exécution appelée *fuite de mémoire*.

4. Les listes chaînées

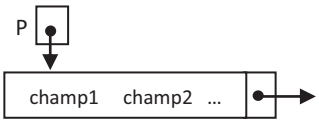
Une liste chaînée est un ensemble d'éléments de même type. Chaque élément est lié à son successeur par un pointeur.

Une liste chaînée est composée d'un ensemble d'éléments (maillons, cellules, nœuds ou composants). Un élément étant une structure (enregistrement) qui contient des données à stocker et un pointeur vers le prochain élément (successeur) de la liste.

```
typedef struct element
{
  <Type_données1>  champ1 ;
  <Type_données2>  champ2 ;
  …
  struct element   *Suivant;
 } Element ;
```

On parle ici d'une structure de données récursive (`Suivant` étant un pointeur vers une structure de même type).

Une variable `P` de type pointeur vers `Element` (`Element *P`) peut être représentée comme suit :

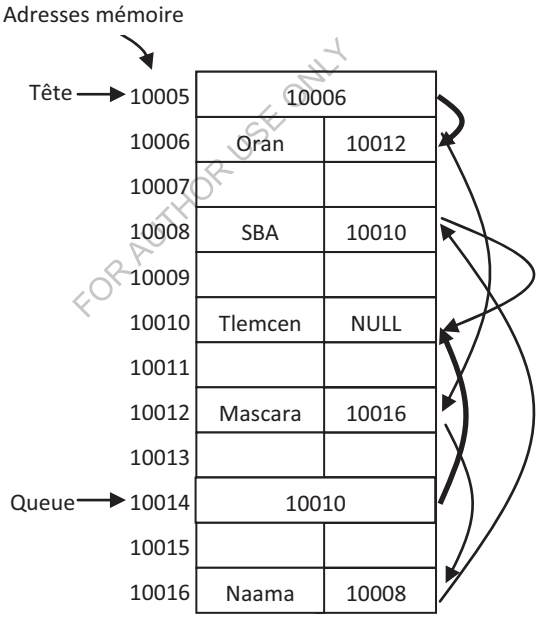


L'adresse de l'élément est rangée dans la variable `P`. L'élément contient deux parties : la partie données contenant les champs : `champ1`, `champ2`…, et le pointeur `Suivant` contenant l'adresse de l'élément suivant (successeur). On accède aux champs de données de l'élément de l'adresse `P` comme suit :
`P->champ1`, `P->champ2`…

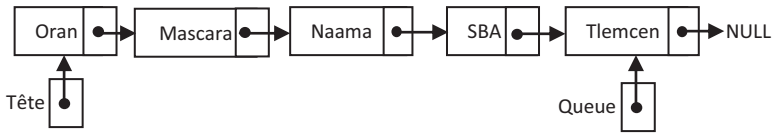
On accède à l'adresse du successeur par `P->Suivant`.

Remarque : Il ne faut pas confondre entre le point (`.`) et la flèche (`->`) pour accéder aux champs d'une structure. Pour accéder aux champs d'une variable de type structure, on utilise un point (eg. `X.num`), mais pour accéder aux champs d'une variable de type pointeur vers structure, on utilise une flèche (eg. `X->num`).

Exemple :
Soit une liste contenant les chaînes de caractères suivantes classées dans cet ordre : `"Oran"`, `"Mascara"`, `"Naama"`, `"SBA"`, `"Tlemcen"`.
La liste peut être représentée en mémoire comme suit :



Une deuxième représentation facile à être assimilée est la suivante :



Il n'est pas possible d'accéder directement à un élément quelconque de la liste. C'est l'adresse (pointeur) du premier élément qui détermine la liste. Cette adresse doit se trouver dans une variable que nous appelons souvent *Tête*, *Sommet* ou *Début* de la liste. Le pointeur vers le dernier élément de la liste est appelé *Queue* de la liste.

5. Opérations sur les listes chaînées

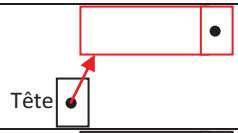
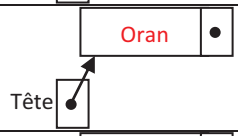
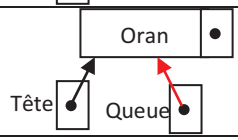
Plusieurs opérations peuvent être effectuées sur une liste.

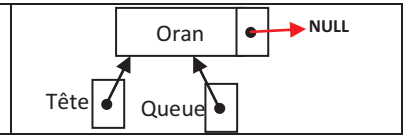
5.1. Créer et remplir une liste

Pour créer et remplir la liste des villes représentée ci-dessus, on doit :

1. Définir la structure `Ville` suivante :
```
typedef struct ville
{
   char Des[20] ;
   struct ville *Suivant ;
} Ville;
```
2. On déclare deux variables `Tete` et `Queue` comme suit :
```
Ville *Tete, *Queue ;
```
3. Le premier élément de la liste peut être créé comme suit :
   - On réserve un espace mémoire pour une variable pointée par `Tete` :
     `Tete = (Ville*)malloc(sizeof(Ville)) ;`
   - Remplir le premier élément de la liste :
     `strcpy(Tete->Des, "Oran") ;`
   - A ce niveau, le premier élément et au même temps le dernier. Par conséquent, la queue pointe vers la tête : `Queue = Tete ;`
   - Le successeur de la queue est `NULL` : `Queue->Suivant = NULL ;`

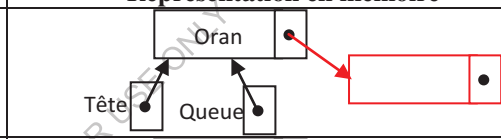
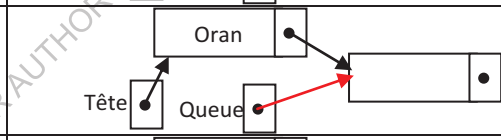
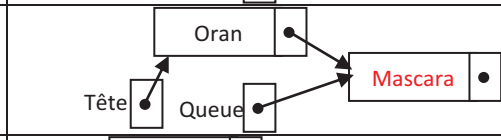
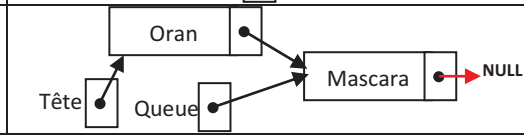
Le tableau suivant décrit le changement d'état de la mémoire selon les instructions du programme :

Instruction	Représentation en mémoire
Tete = (Ville*)malloc(sizeof(Ville)) ;	
strcpy(Tete->Des, "Oran") ;	
Queue = Tete ;	

Queue->Suivant = NULL ;	

4. L'ajout d'un deuxième élément à la fin de la liste se fait comme suit :
   - On réserve un espace mémoire pour une variable pointée par le successeur de la queue :
     ```
     Queue->Suivant = (Ville*)malloc(sizeof(Ville));
     ```
   - La queue va pointer maintenant vers son successeur :
     ```
     Queue = Queue->Suivant ;
     ```
   - Remplir le nouvel élément de la liste :
     ```
     strcpy(Queue->Des, "Mascara") ;
     ```
   - Le successeur de la queue est le `NULL` : `Queue->Suivant = NULL ;`

Le tableau suivant décrit le changement d'état de la mémoire selon les instructions du programme :

Instruction	Représentation en mémoire
Queue->Suivant = (Ville*)malloc(sizeof(Ville));	
Queue = Queue->Suivant ;	
strcpy(Queue->Des, "Mascara") ;	
Queue->Suivant = NULL ;	

5. Pour ajouter le troisième, le quatrième et le cinquième élément, il est possible de substituer l'étape 4 par une procédure permettant l'ajout d'un élément à la fin de la liste. La procédure possède comme paramètre le nom de la ville à ajouter, et elle est écrite comme suit :

```
void Ajouter_Q(char V[20])
{
  Queue->suivant = (Ville*)malloc(sizeof(Ville));
  Queue = Queue->Suivant ;
  strcpy(Queue->Des, V) ;
```

```
      Queue->Suivant = NULL ;
    }
```

Le programme C complet pour créer la liste des villes est alors le suivant :

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
typedef struct ville
{
  char Des[20] ;
  struct ville * Suivant ;
} Ville;
  Ville *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_Q(char V[20])
{
 Queue->Suivant = (Ville*)malloc(sizeof(Ville));
 Queue = Queue->Suivant ;
 strcpy(Queue->Des, V) ;
 Queue->Suivant = NULL ;
}
main()
{
  /* Créer et remplir le premier élément de la liste */
  Tete = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(Tete->Des, "Oran") ;
  Queue = Tete ;
  Queue->Suivant = NULL ;
  /* Ajouter le reste des éléments à la liste */
  Ajouter_Q("Mascara") ;
  Ajouter_Q("Naama") ;
  Ajouter_Q("SBA") ;
  Ajouter_Q("Tlemcen") ;
}
```


5.2. Afficher les éléments de la liste

Pour consulter la liste chaînée que nous avons créée précédemment, il faut se positionner au début de la liste, puis la parcourir élément par élément jusqu'à la fin.

```
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste()
{
  Ville *P ;
  int i ;
  P = Tete;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Suivant;
  }
```

```
}
```

Remarque : Si on remplace la condition (`P != NULL`) par (`P->Suivant != NULL`), le dernier élément de la liste ne sera pas affiché.

On peut aussi parcourir une liste chaînée par une boucle `for` de la manière suivante :

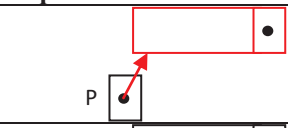
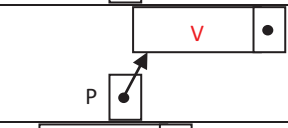
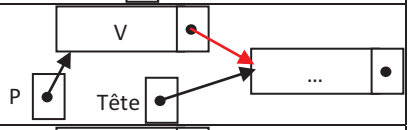
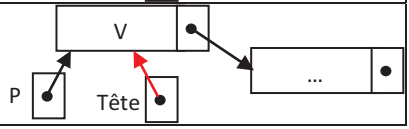
```
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste()
{
  Ville *P ;
  int i ;
  i = 0;
  for (P = Tete; P != NULL; P = P->Suivant) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
  }
}
```

5.3. Ajouter un élément au début de la liste

L'ajout d'un élément au début de la liste se fait comme suit :

- On réserve un espace mémoire pour une variable `P` de type pointeur vers `Ville` : `P = (Ville*)malloc(sizeof(Ville)) ;`
- Remplir le nouvel élément pointé par `P` : `strcpy(P->Des, V) ;`
- On effectue l'enchaînement du nouvel élément avec la liste existante : `P->Suivant = Tete ;`
- La tête va pointer maintenant vers le nouvel élément : `Tete = P;`

Le tableau suivant décrit le changement d'état de la mémoire selon les instructions de la procédure :

Instruction	Représentation en mémoire
P = (Ville*)malloc(sizeof(Ville)) ;	
strcpy(P->Des, V) ;	
P->Suivant = Tete ;	
Tete = P;	

N'oubliez pas de mettre à jour la queue dans le cas où on désir ajouter un élément dans une liste vide.
La procédure permettant l'ajout d'un élément au début de la liste est alors la suivante :

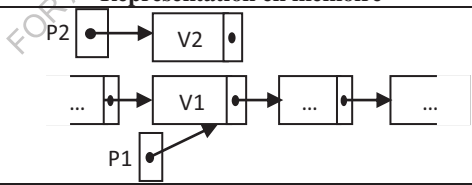
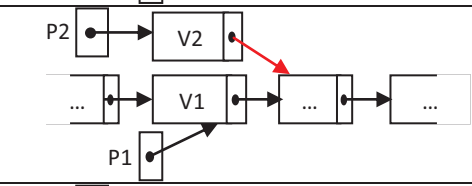
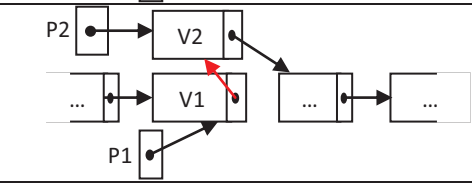
```
/* Procédure d'ajout d'un élément au début de la liste */
void Ajouter_T(char V[20])
{ Ville *P;
  P = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(P->Des, V) ;
  P->Suivant = Tete ;
  Tete = P;
  if (Queue == NULL) Queue = Tete ;
}
```

5.4. Insérer un élément dans la liste

On désir maintenant implémenter une procédure permettant d'insérer un élément après un autre dans une liste (insérer une ville `V2` après `V1`). Pour ce faire, on aura besoin de déclarer deux variables `P1` et `P2` pointant vers une ville : `Ville *P1, *P2`. Le pointeur `P2` va pointer vers l'élément à insérer (un nouvel élément contenant la ville `V2`). Le pointeur `P1` va nous permettre de localiser la ville `V1` dans la liste. Après ça, on doit suivre les étapes suivantes :

- Le successeur de `P2` devient le successeur de `P1` :
  `P2->Suivant = P1->Suivant ;`
- Le successeur de `P1` devient `P2` lui même : `P1->Suivant = P2;`

Le tableau suivant décrit ces deux dernières étapes :

Instruction	Représentation en mémoire
/*Avant l'insertion*/	
P2->Suivant = P1->Suivant ;	
P1->Suivant = P2;	

La procédure permettant d'insérer un élément après un autre est alors la suivante :

```
/* Procédure d'insertion d'une ville V2 après V1 dans la liste */
void Inserer_AP_Ville(char V1[20], char V2[20])
{
  Ville *P1, *P2;
  if (strcmp(Queue->Des, V1) ==0) Ajouter_Q(V2) ;
  else {
  // Se pointer vers l'élément contenant V1
    P1 = Tete;
    while (strcmp(P1->Des, V1) != 0) P1 = P1->Suivant;
  // Créer un nouvel élément contenant V2
    P2 = (Ville*)malloc(sizeof(Ville)) ;
    strcpy(P2->Des, V2) ;
  // Insérer le nouvel élément dans la liste
    P2->Suivant = P1->Suivant ;
    P1->Suivant = P2;
  }
}
```

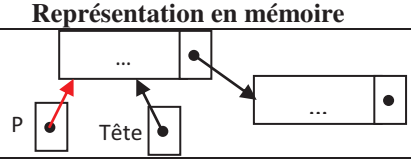
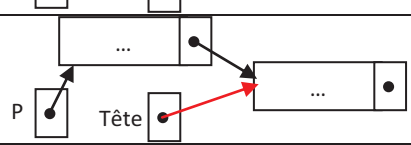
Remarque : Dans la procédure ci-dessus, on suppose que `V1` existe obligatoirement dans la liste. Il faut donc que cette procédure soit enrichie par le traitement du cas où `V1` n'existe pas dans la liste.

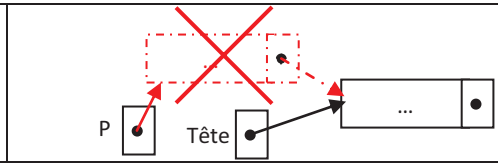
5.5. Supprimer la tête de la liste

Pour supprimer la tête de la liste, on utilise une variable `P` pointant vers une ville : `Ville *P`. Ensuite, il faut suivre les étapes suivantes :

- La variable `P` sera utilisée pour garder l'adresse du premier élément : `P = Tete ;`
- Maintenant, on met à jour l'adresse de la nouvelle tête de la liste : `Tete = Tete->Suivant ;`
- Enfin, on libère l'espace mémoire réservé par l'ancienne tête : `free(P) ;`

Le tableau suivant décrit le changement d'état de la mémoire selon les instructions de la procédure :

Instruction	Représentation en mémoire
P = Tete ;	
Tete = Tete->Suivant ;	

free(P) ;	

La procédure permettant la suppression de la tête de la liste est alors la suivante :

```
/* Procédure de suppression de la tête de la liste */
void Supprimer_T()
{
  Ville *P ;
  if (Tete != NULL)
    if (Tete->Suivant == NULL) {
       free(Tete);
       Tete = NULL;
       Queue = NULL;
       }
    else {
        P = Tete ;
        Tete = Tete->Suivant ;
        free(P) ;
      }
}
```

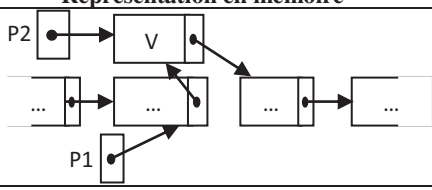

Remarque : Avant de traiter le cas général, nous avons traité les deux cas triviaux : liste vide et liste contenant un seul élément. Un cas trivial est un cas basique et simple qui peut être traité en isolement du cas général.

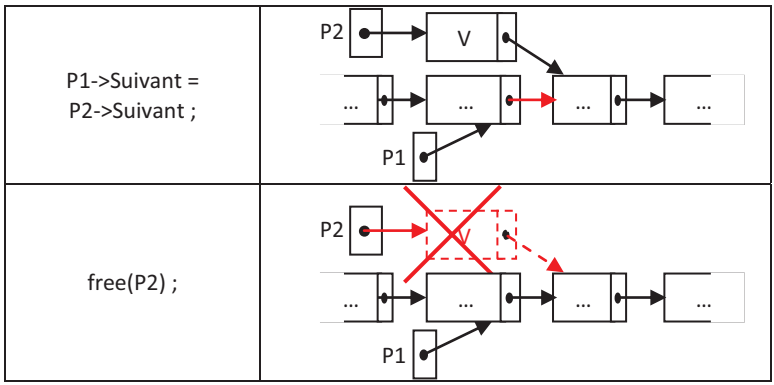
5.6. Supprimer un élément de la liste

La procédure permettant la suppression d'un élément de la liste possède un paramètre `V` indiquant la ville à supprimer. Dans cette procédure, on aura besoin de déclarer deux variables `P1` et `P2` pointant vers une ville : `Ville *P1, *P2`. Le pointeur `P2` va pointer vers l'élément à supprimer. Le pointeur `P1` pointe juste avant `P2`. Après ça, on doit suivre les étapes suivantes :

- Le successeur de `P1` devient le successeur de `P2` :
  `P1->Suivant = P2->Suivant ;`
- Libérer l'espace mémoire occupé par l'élément supprimé : `free(P2) ;`

Le tableau suivant décrit ces deux dernières étapes :

Instruction	Représentation en mémoire
/*Avant la suppression*/	



N'oubliez pas de mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste.

La procédure permettant de supprimer un élément de la liste est alors la suivante :

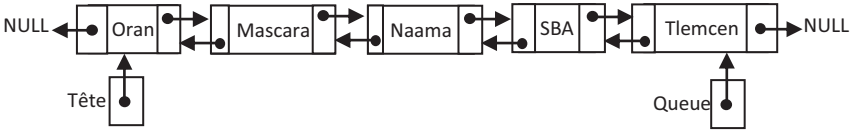
```
/* Procédure de suppression d'un élément de la liste */
void Supprimer(char V[20])
{ Ville *P1, *P2;
  if (strcmp(Tete->Des,V) == 0) Supprimer_T();
  else {
// Se pointer vers l'élément contenant V
       P1 = Tete;
       P2 = P1->Suivant ;
       while (strcmp(P2->Des,V) != 0) {
           P1 = P2;
           P2 = P2->Suivant;
       }
/* Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste */
       if (Queue == P2) Queue = P1 ;
// Supprimer l'élément contenant V
       P1->Suivant = P2->Suivant ;
       free(P2) ;
    }
}
```


Remarque : Dans la procédure ci-dessus, on suppose que `V` existe obligatoirement dans la liste. Il faut donc que cette procédure soit enrichie par le traitement du cas où `V` n'existe pas dans la liste.

6. Les listes doublement chaînées

Il est possible de parcourir une liste dans les deux sens en ajoutant un pointeur permettant l'accès à l'élément précédent. On obtient alors une liste doublement chaînée appelée aussi liste bidirectionnelle, contrairement à une liste simplement chaînée ou unidirectionnelle.

L'ensemble des villes stockées dans une liste doublement chaînée peut être représenté comme suit :



Pour créer la liste doublement chaînée représentée ci-dessus, il faut suivre les étapes suivantes :

1. On définit la structure `Ville` suivante :

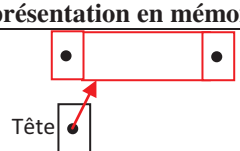
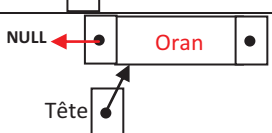
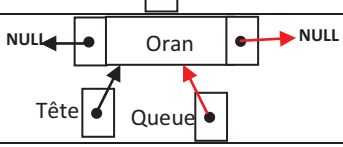
```
typedef struct ville
{
   char Des[20] ;
   struct ville *Precedent, *Suivant ;
} Ville;
```

2. On déclare deux variables `Tete` et `Queue` comme suit :

```
Ville *Tete, *Queue ;
```

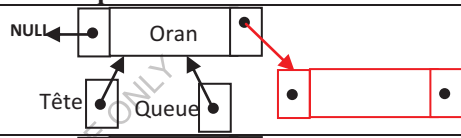
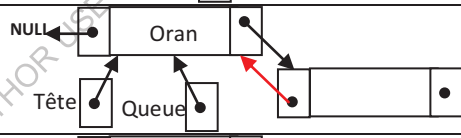
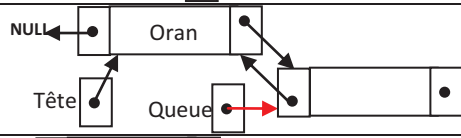
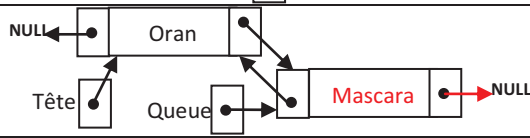
3. Le premier élément de la liste peut être créé comme suit :
   - On réserve un espace mémoire pour une variable pointée par `Tete` :
     `Tete = (Ville*)malloc(sizeof(Ville)) ;`
   - Remplir le premier élément de la liste :
     `strcpy(Tete->Des, "Oran") ;`
     Le premier élément n'a pas de précédent, ce qui se traduit par :
     `Tete->Precedent = NULL ;`
   - A ce niveau, le premier élément et au même temps le dernier. Par conséquent, la queue pointe vers la tête : `Queue = Tete ;`
     Le successeur de la queue est le `NULL` : `Queue->Suivant = NULL ;`

Le tableau suivant décrit le changement d'état de la mémoire selon les instructions du programme :

Instruction	Représentation en mémoire
Tete = (Ville*)malloc(sizeof(Ville)) ;	
strcpy(Tete->Des, "Oran") ; Tete->Precedent = NULL ;	
Queue = Tete ; Queue->Suivant = NULL ;	

4. L'ajout d'un deuxième élément à la fin liste doublement chaînée se fait comme suit :
   - On réserve un espace mémoire pour une variable pointée par le successeur de la queue :
     ```
     Queue->Suivant = (Ville*)malloc(sizeof(Ville));
     ```
   - Le nouvel élément créé sera précédé par la queue :
     ```
     Queue->Suivant->Precedent = Queue ;
     ```
   - La queue va pointer maintenant vers son successeur :
     ```
     Queue = Queue->Suivant ;
     ```
   - Remplir le nouvel élément de la liste :
     ```
     strcpy(Queue->Des, "Mascara") ;
     ```
     Le successeur de la queue est le NULL : `Queue->Suivant = NULL;`

Le tableau suivant décrit le changement d'état de la mémoire selon les instructions du programme :

Instruction	Représentation en mémoire
Queue->Suivant = (Ville*)malloc(sizeof(Ville));	
Queue->Suivant->Precedent = Queue ;	
Queue = Queue->Suivant ;	
strcpy(Queue->Des, "Mascara") ; Queue->Suivant = NULL ;	

5. Pour ajouter le troisième, le quatrième et le cinquième élément, il est possible de substituer l'étape 4 par une procédure permettant l'ajout d'un élément à la fin de la liste. La procédure possède comme paramètre le nom de la ville à ajouter, et elle est écrite comme suit :

```
void Ajouter_Q(char V[20])
{ Queue->Suivant = (Ville*)malloc(sizeof(Ville));
  Queue->Suivant->Precedent = Queue ;
  Queue = Queue->Suivant ;
  strcpy(Queue->Des, V) ;
  Queue->Suivant = NULL ;
}
```

Le programme C complet pour la création de la liste doublement chaînée est alors le suivant :

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
typedef struct ville
{
  char Des[20] ;
  struct ville *Precedent, *Suivant ;
} Ville;
  Ville *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_Q(char V[20])
{ Queue->Suivant = (Ville*)malloc(sizeof(Ville));
  Queue->Suivant->Precedent = Queue ;
  Queue = Queue->Suivant ;
  strcpy(Queue->Des, V) ;
  Queue->Suivant = NULL ;
}
main()
{ /* Créer et remplir le premier élément de la liste */
  Tete = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(Tete->Des, "Oran") ;
  Tete->Precedent = NULL ;
  Queue = Tete ;
  Queue->Suivant = NULL ;
  /* Ajouter le reste des éléments à la liste */
  Ajouter_Q("Mascara") ;
  Ajouter_Q("Naama") ;
  Ajouter_Q("SBA") ;
  Ajouter_Q("Tlemcen") ;
}
```

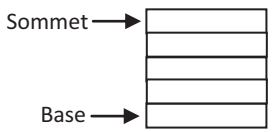

7. Les listes chaînées particulières

En conditionnant l'accès aux listes, on obtient des listes chaînées particulières, à savoir les piles et les files.

7.1. Les piles

Une pile est une liste dont les éléments ne peuvent être ajoutés ou supprimés qu'à partir d'une extrémité appelée sommet de la pile : dernier entré, premier sorti. En anglais, on dit : Last In, First Out (LIFO).

Dans notre vie quotidienne, on trouve par exemple la pile d'assiettes.
La pile peut être représentée tout simplement comme suit :



Le premier élément de la pile est appelé *Sommet* de la pile. Le dernier est appelé *Base* de la pile.

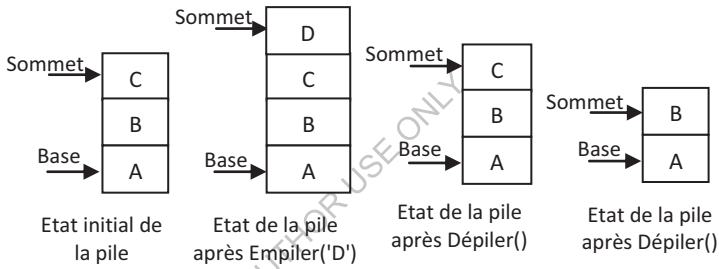
Remarque : Attention ! Il ne faut pas confondre entre la pile (stack) qui est une zone mémoire réservée pour l'exécution des programmes, et la pile étudiée dans cette section.

7.1.1. Primitives d'accès

On définit habituellement les deux primitives suivantes pour accéder à une pile :

- `Empiler(E)` qui ajoute un nouvel élément `E` au sommet de la pile.
- `Dépiler()` qui supprime l'élément sommet de la pile

Exemple : Dans une pile de caractères contenant initialement les éléments : `A`, `B` et `C`, on désir empiler encore le caractère `D`, ensuite dépiler deux caractères. Ces opérations peuvent être représentées comme suit :



7.1.2. Représentation d'une pile par une liste doublement chaînée

Une pile peut être représentée par un tableau, une liste simplement ou doublement chaînée. Dans ce qui suit, nous avons choisi une liste doublement chaînée pour représenter une pile :

- L'opération d'empilement se traduit par un ajout en tête de la liste doublement chaînée.
- L'opération de dépilement se traduit par une suppression de la tête de la liste doublement chaînée.
- Le sommet et la base de la pile étant la tête et la queue de la liste doublement chaînée.

Pour la manipulation d'une pile de caractères, on doit suivre les étapes suivantes :

1. On définit la structure `Pile` comme suit :

```
typedef struct pile
{
  char Car ;
  struct pile *Precedent, *Suivant;
 } Pile ;
```

2. Ensuite, on déclare le sommet et la base de la pile comme suit :

```
Pile *Base, *Sommet ;
```

3. La procédure permettant d'empiler un élément dans la pile est la suivante :

```
/* Procédure permettant d'empiler un élément dans la pile */
void Empiler (char C)
{
  Pile *P ;
  P = (Pile*)malloc(sizeof(Pile));
  P->Car = C ;
  P->Precedent = NULL ;
  P->Suivant = Sommet ;
  if (Sommet != NULL) Sommet->Precedent = P;
  Sommet = P;
  if (Base == NULL) Base = Sommet ;
}
```

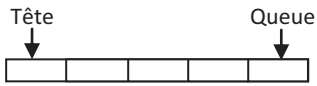
4. La procédure permettant de dépiler un élément de la pile est la suivante :

```
/* Procédure permettant de dépiler un élément de la pile */
void Depiler()
{
if (Sommet == NULL) puts("Dépilement impossible : Pile Vide.");
else if (Sommet->Suivant == NULL) {
          printf("Elément dépilé : %c\n", Sommet->Car) ;
          free(Sommet) ;
          Sommet = NULL ;
          Base = NULL ;
          }
      else { printf("Elément dépilé : %c\n", Sommet->Car) ;
             Sommet = Sommet->Suivant ;
             free(Sommet->Precedent) ;
             Sommet->Precedent = NULL ;
           }
}
```


7.2. Les files

Une file est une liste dont les éléments sont ajoutés à une extrémité appelée queue, et retirés de l'autre appelée tête : premier entré, premier sorti. En anglais, on dit : First In, First Out (FIFO).

Dans notre vie quotidienne, on trouve par exemple la file d'attente à un guichet. La file peut être représentée tout simplement comme suit :



Le premier élément de la file est appelé *Tête* de la file. Le dernier est appelé *Queue* de la file.

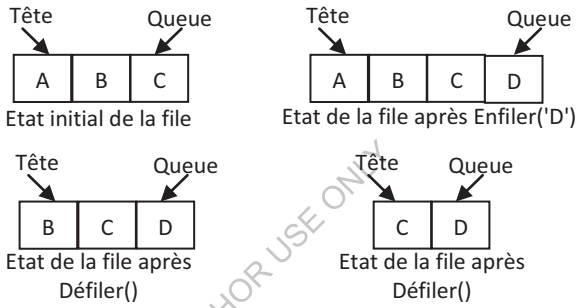
Remarque : Il ne faut pas confondre entre le mot file (en français), et le mot `FILE` (en anglais) qui veut dire fichier et qui est utilisé pour la manipulation des fichiers en C.

7.2.1. Accès à une file

Comme illustré dans l'exemple suivant, on ne peut effectuer un ajout à la file (Enfilement) qu'à travers sa queue. Par contre, la suppression (Défilement) est effectuée à partir de la tête de la file.

Exemple :

Soit une file de caractères contenant initialement les éléments : `A`, `B` et `C`. On désir ajouter le caractère `D`, ensuite supprimer deux caractères. Ces opérations peuvent être représentées comme suit :



7.2.2. Représentation d'une file par une liste doublement chaînée

Comme une pile, la file peut être représentée par un tableau, une liste simplement ou doublement chaînée. Dans ce qui suit, nous avons choisi une liste doublement chaînée pour représenter une file :

- L'opération d'enfilement se traduit par un ajout en queue de la liste doublement chaînée.
- L'opération de défilement se traduit par la suppression de la tête de la liste doublement chaînée.
- La tête et la queue de la file étant la tête et la queue de la liste doublement chaînée.

8. Exercices corrigés

8.1. Exercices

Exercice 1 :

Rappelons que le nom du tableau correspond à l'adresse de la première case du tableau. Rappelons aussi qu'on peut additionner un entier à un pointeur pour avoir un nouveau pointeur (l'adresse de l'emplacement mémoire suivant), donc un pointeur peut être incrémenté. Exploitez ces propriétés pour dire ce qu'affiche le programme suivant :

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

```
main()
{
  int t[4] ={5, 6, 7, 8};
  int *p=t, i;
  for (i=0; i<4; i++){
    printf("%d\n", *p);
    p++; /* Incrémenter le pointeur d'une unité donnera l'adresse
           de l'élément (l'entier dans cet exercice) suivant */
  }
}
```

Exercice 2 :

En utilisant la notion de gestion dynamique de la mémoire, et en exploitant le fait que le nom d'un tableau correspond à un pointeur sur le premier élément du tableau, écrire un programme C permettant de lire un tableau dont le nombre d'éléments est lu au clavier. Le programme doit permettre également d'afficher le contenu de ce tableau.

Exercice 3 :

Qu'affiche le programme C suivant ?

```
#include<stdlib.h>
#include<stdio.h>
main()
{
  int *ad1, *ad2, *ad;
  int n = 10, p = 20;
  ad1 = &n;
  ad2 = &p;
  *ad1 = *ad2 + 2;
  printf("%d\n", n);
  *ad1+=4;
  printf("%d\n", n);
  ad =(int*)malloc(sizeof(int));
  *ad = n + p;
  printf("%d\n", *ad);
}
```


Exercice 4 :

Expliquez ce que fait le programme suivant, partie par partie :

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  int x, *p;
//********************* Pointeur ***************
// Partie 1 : ….
    x = 19;
    p = &x;
    printf("Valeur pointée par p : %d\n", *p);
// Partie 2 : ….
```

```
    p = (int*)malloc(sizeof(int));
    if(p != NULL) {
       *p = 5;
       printf("Valeur pointée par p : %d\n", *p);
       free(p);
       }
//******************** Liste **************
// Partie 3 :    ….
   typedef struct s_Element
   {
     int valeur ;
     struct s_Element *suivant;
   } Element;
// Partie 4 : …..
   Element *Tete, *Queue;
// Partie 5 : …..
   Tete = (Element*)malloc(sizeof(Element));
   Tete->valeur = 80;
   Tete->suivant = NULL;
   Queue = Tete;
// Partie 6 : ….
   Queue->suivant = (Element*)malloc(sizeof(Element));
   Queue = Queue->suivant;
   Queue->valeur = 88;
   Queue->suivant = NULL;
// Partie 7 : ….
   printf("Tete = %d\n", Tete->valeur) ;
   printf("Queue = %d\n", Queue->valeur) ;
}
```


Exercice 5 :

Expliquez la différence entre les deux programmes C :

Programme 1 :

```
#include<stdio.h>
int a, b, *h;
void change1(int *x)
{   x = &a;      }
main()
{
  a=2; b=4;
  h = &b;
  change1(h);
  printf("%d\n", *h);
}
```

Programme 2 :

```
#include<stdio.h>
int a, b, *h;
void change2(int **x)
{  *x = &a;  }
main()
{
  a=2; b=4;
  h = &b;
  change2(&h);
  printf("%d\n", *h);
}
```

Exercice 6 :

Qu'affiche le programme C suivant ?

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

```
typedef struct ville
  {
    char Des[20] ;
    struct ville * Suivant ;
  } Ville;
  Ville *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_Q (char V[20])
{
 Queue->Suivant = (Ville*)malloc(sizeof(Ville));
 Queue = Queue->Suivant ;
 strcpy(Queue->Des, V) ;
 Queue->Suivant = NULL ;
}
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste()
{
  Ville *P ;
  int i ;
  P = Tete;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Suivant;
  }
}
/* Procédure d'ajout d'un élément au début de la liste */
void Ajouter_T(char V[20])
{
  Ville *P;
  P = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(P->Des, V) ;
  P->Suivant = Tete ;
  Tete = P;
  if (Queue == NULL) Queue = Tete ;
}
/* Procédure d'insertion d'une ville V2 après V1 dans la liste */
void Inserer_AP_Ville(char V1[20], char V2[20])
{
  Ville *P1, *P2;
  if (strcmp(Queue->Des, V1) ==0) Ajouter_Q(V2) ;
  else {
  // Se pointer vers l'élément contenant V1
    P1 = Tete;
    while (strcmp(P1->Des, V1) != 0) P1 = P1->Suivant;
  // Créer un nouvel élément contenant V2
    P2 = (Ville*)malloc(sizeof(Ville)) ;
    strcpy(P2->Des, V2) ;
  // Insérer le nouvel élément dans la liste
```

```
    P2->Suivant = P1->Suivant ;
    P1->Suivant = P2;
  }
}
/* Procédure de suppression de la tête de la liste */
void Supprimer_T()
{
  Ville *P ;
  if (Tete != NULL)
    if (Tete->Suivant == NULL) {
       free(Tete);
       Tete = NULL;
       Queue = NULL;
       }
    else {
        P = Tete ;
        Tete = Tete->Suivant ;
        free(P) ;
      }
}
/* Procédure de suppression d'un élément de la liste */
void Supprimer(char V[20])
{
  Ville *P1, *P2;
  if (strcmp(Tete->Des,V) == 0) Supprimer_T();
  else {
// Se pointer vers l'élément contenant V
      P1 = Tete;
      P2 = P1->Suivant ;
      while (strcmp(P2->Des,V) != 0) {
          P1 = P2;
          P2 = P2->Suivant;
      }
    //Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste
      if (Queue == P2) Queue = P1 ;
    // Supprimer l'élément contenant V
      P1->Suivant = P2->Suivant ;
      free(P2) ;
    }
}
main()
{
  /* Créer et remplir le premier élément de la liste */
  Tete = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(Tete->Des, "Oran") ;
  Queue = Tete ;
  Queue->Suivant = NULL ;
  /* Ajouter le reste des éléments à la liste */
  Ajouter_Q("Mascara") ;
  Ajouter_Q("Naama") ;
```

```
  Ajouter_Q("SBA") ;
  Ajouter_Q("Tlemcen") ;
  Ajouter_T("Bechar");
  Inserer_AP_Ville("Bechar", "Blida");
  Supprimer_T();
  Supprimer("Mascara");
  Afficher_Liste();
}
```

Exercice 7 :
Ecrire en C une procédure d'ajout d'un élément au début de la liste doublement chaînée (la liste des villes créée dans le cours).

Exercice 8 :
Ecrire en C une procédure permettant d'insérer un élément après un autre dans une liste doublement chaînée (insérer une ville `V2` après `V1` dans la liste des villes créée dans le cours).

Exercice 9 :
Ecrire en C une procédure permettant de supprimer la tête de la liste doublement chaînée et récupérer l'espace mémoire occupé par cet élément (la liste des villes créée dans le cours).

Exercice 10 :
Ecrire en C une procédure permettant la suppression d'un élément de la liste doublement chaînée (la liste des villes créée dans le cours). La procédure possède un paramètre `V` indiquant la ville à supprimer.

Exercice 11 :
Ecrire en C une procédure permettant d'afficher les éléments de la liste doublement chaînée (la liste des villes créée dans le cours).

Exercice 12 :
Qu'affiche le programme C suivant ?

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
typedef struct ville
{
  char Des[20] ;
  struct ville *Precedent, *Suivant ;
} Ville;
  Ville *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_Q(char V[20])
{
  Queue->Suivant = (Ville*)malloc(sizeof(Ville));
  Queue->Suivant->Precedent = Queue ;
  Queue = Queue->Suivant ;
  strcpy(Queue->Des, V) ;
```

```
  Queue->Suivant = NULL ;
}
/* Procédure d'affichage des éléments de la liste T*/
void Afficher_Liste_T()
{
  Ville *P ;
  int i ;
  P = Tete;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Suivant;
  }
}
/* Procédure d'affichage des éléments de la liste Q*/
void Afficher_Liste_Q()
{
  Ville *P ;
  int i ;
  P = Queue;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Precedent;
  }
}
/* Procédure d'ajout d'un élément au début de la liste */
void Ajouter_T(char V[20])
{
  Ville *P;
  P = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(P->Des, V) ;
  P->Precedent = NULL ;
  P->Suivant = Tete ;
  if (Tete != NULL) Tete->Precedent = P;
  Tete = P;
  if (Queue == NULL) Queue = Tete ;
}
/* Procédure d'insertion d'une ville V2 après V1 dans la liste */
void Inserer_AP_Ville(char V1[20], char V2[20])
{
  Ville *P1, *P2;
  if (strcmp(Queue->Des, V1) ==0) Ajouter_Q(V2) ;
  else {
  // Se pointer vers l'élément contenant V1
    P1 = Tete;
    while (strcmp(P1->Des, V1) != 0) P1 = P1->Suivant;
  // Créer un nouvel élément contenant V2
```

```
    P2 = (Ville*)malloc(sizeof(Ville)) ;
    strcpy(P2->Des, V2) ;
  // Insérer le nouvel élément dans la liste
    P2->Suivant = P1->Suivant ;
    P2->Precedent = P1 ;
    P1->Suivant = P2;
    P2->Suivant->Precedent = P2 ;
  }
}
/* Procédure de suppression de la tête de la liste */
void Supprimer_T()
{
  if (Tete != NULL)
    if (Tete->Suivant == NULL) {
       free(Tete);
       Tete = NULL;
       Queue = NULL;
       }
    else {
        Tete = Tete->Suivant ;
        free(Tete->Precedent) ;
        Tete->Precedent = NULL;
      }
}
/* Procédure de suppression d'un élément de la liste */
void Supprimer(char V[20])
{
  Ville *P;
  if (strcmp(Tete->Des,V) == 0) Supprimer_T();
  else {
// Se pointer vers l'élément contenant V
     P = Tete;
     while (strcmp(P->Des,V) != 0) P = P->Suivant;
    //Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste
     if (Queue == P) Queue = P->Precedent ;
    // Supprimer l'élément contenant V
     P->Precedent->Suivant = P->Suivant ;
    if (P->Suivant != NULL) P->Suivant->Precedent = P->Precedent;
     free(P) ;
    }
}
main()
{ /* Créer et remplir le premier élément de la liste */
  Tete = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(Tete->Des, "Oran") ;
  Tete->Precedent = NULL ;
  Queue = Tete ;
  Queue->Suivant = NULL ;
  /* Ajouter le reste des éléments à la liste */
  Ajouter_Q("Mascara") ;
```

```
  Ajouter_Q("Naama") ;
  Ajouter_Q("SBA") ;
  Ajouter_Q("Tlemcen") ;
  Ajouter_T("Bechar");
  Inserer_AP_Ville("SBA", "Blida");
  Supprimer_T();
  Supprimer("Oran");
  Afficher_Liste_T() ;
}
```

Exercice 13 :

Ecrire un programme C permettant de lire et d'afficher un entier. La lecture étant faite par une fonction qui réserve un espace mémoire pour une variable pointée, lit la valeur de cette variable au clavier et retourne cette valeur au programme appelant.

Exercice 14 :

Ecrire un programme C permettant de créer et d'afficher une liste simplement chaînée. Chaque élément est de type structure `Personne` contenant les informations : `Num` de type entier et `Nom` de type chaîne de caractères. Utilisez une procédure pour l'ajout d'une personne et une autre pour afficher les éléments de la liste. La tête et la queue de la liste étant captés par deux pointeurs `Tete` et `Queue` pointant vers `Personne`.

Quelle instruction doit-on ajouter pour rendre la liste circulaire ? Dans une liste circulaire, il est possible de revenir à un point de départ en parcourant la liste dans un seul sens sans revenir dans le sens inverse.

Exercice 15 :

Ecrire en C une procédure permettant de libérer l'espace mémoire occupé par la liste des personnes déjà créée dans l'exercice 14.

Exercice 16 :

Ecrire en C une fonction entière permettant de calculer la longueur d'une liste. Les éléments de la liste étant de type `Personne` décrit dans l'exercice 14. La fonction doit posséder comme paramètre la tête de la liste.

Exercice 17 :

Ecrire en C une fonction booléenne permettant la recherche d'un élément dans la liste des personnes. La fonction doit posséder comme paramètres la tête de la liste, le numéro et le nom de l'élément à rechercher.

Exercice 18 :

Modifier la fonction de l'exercice précédent de telle sorte que cette fonction retourne un pointeur vers l'élément recherché s'il existe. Dans le cas contraire, elle pointe vers le `NULL`.

Exercice 19 :

Ecrire en C une fonction qui retourne un pointeur vers le $n^{ième}$ élément de la liste des personnes. Si `n` dépasse la taille de la liste, la fonction retourne `NULL`.

Si `n` est inférieur à `0`, la fonction retourne `NULL` aussi. La fonction doit posséder comme paramètres la tête de la liste et le numéro d'ordre, commençant par 0, de l'élément désiré.

Exercice 20 :

Ecrire en C la procédure permettant d'insérer un élément à la $k^{ième}$ position dans une liste de personnes. Si $k<=0$, l'ajout sera en tête de la liste. Si `k` dépasse la longueur de la liste, l'ajout sera en queue de la liste. La procédure aura comme paramètres les informations à insérer, la position, la tête et la queue de la liste.

Exercice 21 :

Ecrire en C une procédure permettant de supprimer l'élément de la $k^{ième}$ position dans une liste de personnes. Si $k<=0$, c'est la tête qui sera supprimée de la liste. Si le `k` dépasse la longueur de la liste, la queue sera supprimée. La procédure aura comme paramètres la position, la tête et la queue de la liste.

Exercice 22 :

Ecrire en C une procédure permettant de renverser et d'afficher une liste de personnes. La procédure doit posséder comme paramètres un pointeur vers la tête et un autre vers la queue de la liste à renverser.

Exercice 23 :

Ecrire en C une procédure récursive permettant d'afficher une liste de personnes.

Exercice 24 :

Ecrire un programme C permettant de créer deux listes chaînées, ayant un nombre d'éléments indéfini. Ensuite, afficher les éléments des deux listes. Chaque élément d'une liste est de type structure `Nombre` contenant une information : `Val` de type entier. Utilisez une procédure pour la création d'une liste. Cette procédure doit posséder la tête et la queue d'une liste comme paramètres. Pour l'ajout d'un élément à une liste, utilisez une procédure possédant comme paramètre la queue de la liste dans laquelle on veut ajouter le nouvel élément. Une autre procédure possédant la tête d'une liste comme paramètre est utilisée pour afficher les éléments de cette liste. La tête et la queue de la première liste étant captés par deux pointeurs `Tete1` et `Queue1`. La tête et la queue de la deuxième liste étant captés par deux pointeurs `Tete2` et `Queue2`.

Exercice 25 :

Ecrire en C une fonction qui retourne la somme des éléments d'une liste de nombres dont la structure est déjà décrite ci-dessus. La fonction possède comme paramètre la tête de la liste en question.

Exercice 26 :

Ecrire en C une procédure permettant, à partir d'une liste de nombres, de créer deux listes de nombres : la première ayant des nombres impairs, la deuxième

avec des nombres pairs. La procédure possède comme paramètres la tête de la liste source, la tête et la queue de la première liste (celle qui va contenir les nombres pairs), et la tête et la queue de la deuxième liste (celle qui va contenir les nombres impairs).

Exercice 27 :

Ecrire en C une procédure permettant de concaténer deux listes. Les éléments de la liste étant de type `Nombre` décrit ci-dessus. La procédure doit posséder comme paramètres un pointeur vers la tête de la première liste (passé par adresse, car c'est lui qui va récupérer le résultat de la concaténation) et un autre vers la tête de la deuxième liste.

Exercice 28 :

Ecrire en C une procédure permettant de trier une liste en ordre croissant. Les éléments de la liste étant de type `Nombre` déjà décrit. La procédure doit posséder comme paramètre un pointeur vers la tête de la liste à trier.

Exercice 29 :

Ecrire en C une procédure permettant d'insérer un élément dans une liste triée en ordre croissant. Les éléments de la liste étant de type `Nombre` déjà défini. La procédure doit posséder comme paramètres la valeur à insérer et un pointeur vers la tête de la liste.

Exercice 30 :

Ecrire en C une procédure permettant de fusionner deux listes déjà triées en ordre croissant. Les éléments de chaque liste étant de type `Nombre` déjà décrit. La procédure doit posséder comme paramètres un pointeur vers la tête de la première liste (passé par adresse pour récupérer le résultat) et un pointeur vers la tête de la deuxième liste.

Exercice 31 :

Reprendre l'exercice 14, mais cette fois-ci en stockant les informations dans une liste doublement chaînée.

Exercice 32 :

Ecrire en C une procédure permettant l'insertion d'un élément avant un autre dans la liste doublement chaînée créée dans l'exercice précédent. La procédure doit posséder comme paramètres le numéro et le nom de l'élément à insérer, ainsi que le numéro et le nom de l'élément avant lequel on désir faire l'insertion.

Exercice 33 :

Ecrire en C une procédure permettant de supprimer les éléments qui se répètent dans la liste doublement chaînée créée ci-dessus. La procédure possède comme paramètre la tête de la liste à traiter. On veut donc modifier la liste de manière à ce que la même personne (`Num` et `Nom`) n'apparaisse qu'une seule fois dans cette liste.

Exercice 34 :
Ecrire un programme C permettant la gestion de la pile de caractères décrite dans le cours : empilement, dépilement et affichage du contenu de la pile.

Exercice 35 :
Ecrire un programme C permettant la gestion de la file de caractères décrite dans le cours : enfilement, défilement et affichage du contenu de la file.

8.2. Corrigés

Solution 1 :
Le programme affiche les éléments du tableau, à savoir :
```
5
6
7
8
```

Solution 2 :
```
#include<stdio.h>
#include<stdlib.h>
main()
{ int *t, i, n;
  printf("Donnez le nombre d'éléments du tableau : ");
  scanf("%d", &n);
  t = (int*) malloc (sizeof(int)*n);
  puts("Donnez les éléments du tableau :");
  for(i=0; i<n; i++)  scanf("%d", &t[i]) ;
  puts("Voici les éléments du tableau :");
  for(i=0; i<n; i++) printf("%d\n", t[i]) ;
}
```


Solution 3 :
Le programme affiche :
```
22
26
46
```

Solution 4 :
Voici ce que fait le programme C partie par partie :
Partie 1 : affecter une adresse à une variable pointeur.
Partie 2 : allocation et désallocation.
Partie 3 : définition de la structure d'un élément de la liste .
Partie 4 : déclarer la tête et la queue.
Partie 5 : créer le premier élément de la liste 80.
Partie 6 : créer le deuxième élément de la liste 88.
Partie 7 : afficher l'élément tête et l'élément queue.

Solution 5 :

Dans le premier programme, le pointeur `*x` est passé par valeur à la procédure `change1`. L'appel de la procédure `change1` avec le paramètre effectif `h` ne change pas la valeur de ce dernier, et elle restera `&b`. Donc, c'est la valeur de `b` qui sera affichée, à savoir `4`.

Dans le deuxième programme, le pointeur `*x` est passé par adresse à la procédure `change2`, c'est pourquoi nous avons mis `int **x`. L'appel de la procédure `change2` avec `&h` changera la valeur de `h`, et elle deviendra `&a`. Par conséquent, c'est la valeur de `a` qui sera affichée, à savoir `2`.

Solution 6 :

Le programme affiche :

```
L'élément num ° 0 de la liste est : Blida
L'élément num ° 1 de la liste est : Oran
L'élément num ° 2 de la liste est : Naama
L'élément num ° 3 de la liste est : SBA
L'élément num ° 4 de la liste est : Tlemcen
```

Solution 7 :

```
/* Procédure d'ajout d'un élément au début de la liste
doublement chaînée des villes */
void Ajouter_T(char V[20])
{
  Ville *P;
  P = (Ville*)malloc(sizeof(Ville)) ;
  strcpy(P->Des, V) ;
  P->Precedent = NULL;
  P->Suivant = Tete ;
  if (Tete != NULL) Tete->Precedent = P;
  Tete = P;
  if (Queue == NULL) Queue = Tete ;
}
```


Solution 8 :

```
/* Procédure d'insertion d'une ville V2 après V1 dans la liste
doublement chaînée des villes */
void Inserer_AP_Ville(char V1[20], char V2[20])
{
  Ville *P1, *P2;
  if (strcmp(Queue->Des, V1) ==0) Ajouter_Q(V2) ;
  else {
  // Se pointer vers l'élément contenant V1
    P1 = Tete;
    while (strcmp(P1->Des, V1) != 0) P1 = P1->Suivant;
  // Créer un nouvel élément contenant V2
    P2 = (Ville*)malloc(sizeof(Ville)) ;
    strcpy(P2->Des, V2) ;
  // Insérer le nouvel élément dans la liste
```

```
    P2->Suivant = P1->Suivant ;
    P2->Precedent = P1 ;
    P1->Suivant = P2;
    P2->Suivant->Precedent = P2 ;
  }
}
```

Solution 9 :

```
/* Procédure de suppression de la tête de la liste doublement
chaînée des villes */
void Supprimer_T()
{
  if (Tete != NULL)
    if (Tete->Suivant == NULL) {
       free(Tete);
       Tete = NULL;
       Queue = NULL;
       }
    else {
        Tete = Tete->Suivant ;
        free(Tete->Precedent) ;
        Tete->Precedent = NULL;
      }
}
```

Solution 10 :

```
/* Procédure de suppression d'un élément de la liste
doublement chaînée des villes */
void Supprimer(char V[20])
{
  Ville *P;
  if (strcmp(Tete->Des,V) == 0) Supprimer_T();
  else {
   // Se pointer vers l'élément contenant V
     P = Tete;
     while (strcmp(P->Des,V) != 0) P = P->Suivant;
   //  Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste
     if (Queue == P) Queue = P->Precedent ;
   // Supprimer l'élément contenant V
     P->Precedent->Suivant = P->Suivant ;
    if (P->Suivant != NULL) P->Suivant->Precedent = P->Precedent;
     free(P) ;
    }
}
```

Solution 11 :

La procédure permettant d'afficher la liste des villes doublement chaînée créée dans le cours de gauche à droite est la suivante :

```
/* Procédure d'affichage des éléments de la liste des villes à
partir de la tête */
void Afficher_Liste_T()
```

```
{
  Ville *P ;
  int i ;
  P = Tete;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Suivant;
  }
}
```

La procédure permettant d'afficher la liste des villes doublement chaînée créée dans le cours de droite à gauche est la suivante :

```
/* Procédure d'affichage des éléments de la liste des villes à
partir de la queue */
void Afficher_Liste_Q()
{
  Ville *P ;
  int i ;
  P = Queue;
  i = 0;
  while (P != NULL) {
   printf("L'élément num ° %d de la liste est : %s\n", i, P->Des);
   i = i + 1 ;
   P = P->Precedent;
  }
}
```


Solution 12 :

Le programme affiche :

```
L'élément num ° 0 de la liste est : Mascara
L'élément num ° 1 de la liste est : Naama
L'élément num ° 2 de la liste est : SBA
L'élément num ° 3 de la liste est : Blida
L'élément num ° 4 de la liste est : Tlemcen
```

Solution 13 :

```
#include<stdio.h>
#include <stdlib.h>
int lire_entier()
 { int *p, resultat;
   p = (int*)malloc(sizeof(int));
   printf("Entrez l'entier : "); scanf("%d",p);
   resultat = *p;
   free(p);
   return resultat;
 }
main()
{ int x;
```

```
  x = lire_entier();
  printf("Voici l'entier : %d\n",x);
}
```

Solution 14 :

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
typedef struct personne
  {
    int Num;
    char Nom[20] ;
    struct personne * Suivant ;
  } Personne;
  Personne *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_E_Q(int N, char M[20])
{
      if(Tete == NULL){
      /* Créer et remplir le premier élément de la liste */
      Tete = (Personne*)malloc(sizeof(Personne)) ;
      Tete->Num = N;
      strcpy(Tete->Nom, M) ;
      Queue = Tete ;
      Queue->Suivant = NULL ;
       }
       else {
      /* Ajout d'un élément à la fin de la liste */
      Queue->Suivant = (Personne*)malloc(sizeof(Personne));
      Queue = Queue->Suivant ;
      Queue->Num = N;
      strcpy(Queue->Nom, M) ;
      Queue->Suivant = NULL ;
       }
}
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste()
{
  Personne *P ;
  P = Tete;
  if (P != NULL) puts("Voici Les éléments de la liste :");
  else puts("Liste vide.");
  while (P != NULL) {
    printf("Num : %d,  Nom : %s\n", P->Num, P->Nom) ;
    P = P->Suivant;
  }
}
main()
{
  Tete = NULL; Queue = NULL;
```

```
  char c, s[20];
  int n;
  do{
   printf("Voulez-vous ajouter un élément à la liste o/n ? ");
   c = getchar(); getchar();
   if(c=='o'){
    printf("Donnez le num de la personne à ajouter : ");
    scanf("%d", &n);
    printf("Donnez le nom de la personne à ajouter : ");
    scanf("%s", s); getchar();
    Ajouter_E_Q(n, s);
    }
   } while(c != 'n');
  Afficher_Liste();
}
```

Pour obtenir une liste simplement chaînée circulaire, il faut ajouter après la création de la liste, l'instruction : `Queue->Suivant = Tete ;`

Solution 15 :

```
/* Procédure de libération de l'espace mémoire occupé par la
liste des personnes */
void Liberer_Liste()
{
  Personne *P ;
  P = Tete;
  while (P != NULL) {
    Tete = Tete->Suivant;
    free(P);
    P = Tete;
  }
  Queue = NULL;
}
```


Solution 16 :

```
/* Fonction de calcul de la longueur de la liste des
personnes*/
int Longueur_Liste(Personne *T)
{ Personne *P ;
  P = T; int L =0;
  while (P != NULL) {
    L++;
    P = P->Suivant;
  }
  return L;
}
```

Solution 17 :

```
/* Fonction de recherche d'un élément dans la liste des
personnes */
int Rechercher_Liste(int N, char M[20], Personne *T)
{
```

```
  Personne *P ;
  P = T; int Exist =0;
  while ((P != NULL) && !Exist)
       if((P->Num==N)&&(strcmp(P->Nom, M) ==0)) Exist = 1;
              else P = P->Suivant;
  return Exist;
}
```

Solution 18 :

```
/* Fonction de recherche d'un élément dans la liste des
personnes */
Personne *Rechercher_Liste(int N, char M[20], Personne *T)
{
  Personne *P, *Exist ;
  P = T; Exist = NULL;
  while ((P != NULL) && (Exist)== NULL)
       if((P->Num==N)&&(strcmp(P->Nom, M) ==0)) Exist = P;
              else P = P->Suivant;
  return Exist;
}
```

Solution 19 :

```
/* Fonction de recherche du nieme élément de la liste des
personnes */
Personne *Nieme_Element_Liste(int N, Personne *T)
{
  Personne *P ;
  if (N>=0) P = T;
  else P = NULL;
  int n=0;
  while ((P != NULL) && (n < N)) {
    P = P->Suivant;
    n++;
  }
  return P;
}
```


Solution 20 :

```
/* Procédure d'insertion d'un élément à la kième position dans
la liste des personnes */
void Inserer_K (int N, char M[20], int k, Personne **T,
Personne **Q)
{
       Personne *P1, *P2; int n=0;
       P1 = (Personne*)malloc(sizeof(Personne));
       P1->Num = N;
       strcpy(P1->Nom, M);
       P1->Suivant = NULL;
       if(*T == NULL){
              //Si la liste est initialement vide
                     *T=P1; *Q=P1;
```

```
            }
    else
       if (k<=0){
            // Ajout en Tete de la liste
                  P1->Suivant = *T;
                  *T = P1;
       }else{
            // Insérer un élément dans la liste
            P2 = *T;
            while((P2->Suivant != NULL)&&(n<k-1)){
                  P2 = P2->Suivant; n++;
            }
            P1->Suivant = P2->Suivant;
            P2->Suivant = P1;
            if (*Q == P2) *Q = P1;
       }
}
```

Remarque : Le premier élément de la liste possède la position 0. Pour insérer la personne `"Mohamed"` dont le numéro est `15` à la position `2`, on appelle la procédure par : `Inserer_K(15, "Mohamed", 2, &Tete, &Queue);`

Solution 21 :

```
/* Procédure de suppression de l'élément de la kième position dans
la liste des personnes */
void Supprimer_K(int k, Personne **T, Personne **Q)
{
  Personne *P1, *P2; int n=0;
  if(*T == NULL){
    //Si la liste est initialement vide
    puts("Liste vide. Suppression impossible.");
       }
    else
      if (k<=0){
      // Suppression de la Tete de la liste
      P1 = *T;
      *T = P1->Suivant;
      if (*Q == P1) *Q = *T;
      printf("Elément Supprimé... Num : %d,  Nom : %s\n", P1->Num, P1->Nom);
      free(P1);
       }else
       if((*T)->Suivant == NULL) {
         free(*T);
         *T = NULL;
         *Q = NULL;
         }else{
         // Supprimer un élément de la liste
         P2 = *T;
         while((P2->Suivant != NULL)&&(n<k-1)){
              P2 = P2->Suivant; n++;
```

```
            }
          P1 = P2->Suivant;
          if(P1 != NULL) P2->Suivant = P1->Suivant;
          if (*Q == P1) *Q = P2;
          printf("Elément Supprimé... Num : %d,  Nom : %s\n", P1->Num, P1->Nom);
          free(P1);
        }
}
```

Remarque : De même que pour l'insertion, la position du premier élément est 0. Pour supprimer l'élément dont la position est `1`, on met :

```
Supprimer_K(1, &Tete, &Queue);
```

Solution 22 :

```
// Procédure de reversement de la liste des personnes
void Renverser(Personne **T, Personne **Q)
{
    Personne *P1, *P2;
    if (*T != NULL) {
    P1 = *T;
    while (P1->Suivant != NULL) {
      P2 = P1->Suivant ;
      P1->Suivant = P2->Suivant ;
      P2->Suivant = *T ;
      *T = P2 ;
    }
    *Q = P1 ;
  }
}
```


Remarque : L'appel de la procédure se fait par :

```
Renverser(&Tete, &Queue);
```

Solution 23 :

```
/* Procédure récursive d'affichage des éléments de la liste
des personnes */
void Afficher_Liste_R(Personne *T)
{
  if (T != NULL) {
    printf("Num : %d,  Nom : %s\n", T->Num, T->Nom) ;
    Afficher_Liste_R(T->Suivant);
  }
}
```

Remarque : L'appel de la procédure se fait par : `Afficher_Liste_R(Tete);`

Solution 24 :

```
#include<stdio.h>
#include <stdlib.h>
typedef struct nombre {
    int Val ;
    struct nombre *Suivant;
} Nombre;
```

```
  Nombre *Tete1, *Tete2, *Queue1, *Queue2 ;
  char c ;
/* Procédure d'ajout d'un élément à la liste */
void Ajouter_Nombre(Nombre **Q)
{
  (*Q)->Suivant = (Nombre*)malloc(sizeof(Nombre));
  *Q = (*Q)->Suivant;
  printf("Donnez la valeur de cet élément : ");
  scanf("%d", &(*Q)->Val);
  (*Q)->Suivant = NULL ;
}
/* Procédure de création d'une liste */
void Creer_Liste(Nombre **T, Nombre **Q)
{
  /* Initialement la liste est vide */
  *T = NULL ;
  *Q = NULL ;
  /* Créer et remplir le premier élément de la liste courante */
  *T = (Nombre*)malloc(sizeof(Nombre));
  printf("Donnez la valeur du premier élément de la liste : ");
  scanf("%d", &(*T)->Val);
  getchar();
  *Q = *T ;
  (*Q)->Suivant = NULL ;
  do{
    printf("Voulez-vous ajouter un autre élément o/n ? ");
    c=getchar();
    if (c == 'o') Ajouter_Nombre(&*Q);
    getchar();
  } while(c != 'n');
}
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste(Nombre *T)
{
 Nombre *P;
 P = T;
 while (P != NULL) {
    printf("%d\n", P->Val) ;
    P = P->Suivant;
  }
}
main() /* Programme principal */
{
  puts("******** Création de la première liste ********");
  Creer_Liste(&Tete1, &Queue1);
  puts("\n******** Création de la deuxième liste ********");
  Creer_Liste(&Tete2, &Queue2);
  puts("\n******** Voici les éléments de la première liste ********");
  Afficher_Liste(Tete1);
  puts("\n******** Voici les éléments de la deuxième liste ********");
```

```
  Afficher_Liste(Tete2);
}
```

Solution 25 :

```
/* Fonction de calcul de la somme des éléments de la liste des
nombres */
int Somme_Liste(Nombre *T)
{
       Nombre *P;
       P = T; int S =0;
       while (P != NULL){
              S+= P->Val;
              P = P->Suivant;
       }
       return S;
}
```

Remarque : Si on veut, par exemple, afficher la somme des éléments de la première liste de l'exercice 24, on met :

```
printf("%d", Somme_Liste(Tete1));
```

Solution 26 :

```
/* Procédure de division d'une liste de nombres en deux listes
paire et impaire */
void Diviser_Liste(Nombre *T, Nombre **T1, Nombre **Q1,
                   Nombre **T2, Nombre **Q2)
{
  Nombre *P;
  P = T;
  while (P != NULL){
    if (P->Val % 2 == 0)
       if (*T1 == NULL){
         *T1 = (Nombre*)malloc(sizeof(Nombre));
         (*T1)->Val = P->Val;
         (*T1)->Suivant = NULL;
         *Q1 = *T1;
       }
       else{
        (*Q1)->Suivant = (Nombre*)malloc(sizeof(Nombre));
        *Q1 = (*Q1)->Suivant;
        (*Q1)->Val = P->Val;
        (*Q1)->Suivant = NULL;
      }
    else if (*T2 == NULL){
           *T2 = (Nombre*)malloc(sizeof(Nombre));
           (*T2)->Val = P->Val;
           (*T2)->Suivant = NULL;
           *Q2 = *T2;
         }
         else{
          (*Q2)->Suivant = (Nombre*)malloc(sizeof(Nombre));
```

```
            *Q2 = (*Q2)->Suivant;
            (*Q2)->Val = P->Val;
            (*Q2)->Suivant = NULL;
          }
      P= P->Suivant;
    }
}
```

Remarque : Pour utiliser cette procédure dans le programme de l'exercice 24, on peut déclarer quatre nouveaux pointeurs comme variables globales : `Nombre *TP, *QP, *TI, *QI`. Les deux premiers pointeurs (`TP` et `QP`) correspondent à la liste qui va contenir les nombres pairs, et les deux autres (`TI` et `QI`) correspondent à la liste qui va contenir les nombres impairs. L'appel d'une telle procédure au niveau du programme principal se fera, par exemple, par : `Diviser_Liste(Tete1, &TP, &QP, &TI, &QI)`. Ensuite, on peut afficher les deux nouvelles listes par : `Afficher_Liste(TP)` et `Afficher_Liste(TI)`.

Solution 27 :

```
/* Procédure de concaténation de deux listes de nombres */
void Concat_Liste(Nombre **T1, Nombre *T2)
{
  Nombre *P;
  if ((*T1!=NULL) && (T2 != NULL)) {
    P = *T1 ;
    while (P->Suivant != NULL) P = P->Suivant ;
    P->Suivant = T2 ;
  }
}
```


Remarque : Cette procédure peut être invoquée dans le programme de l'exercice 24, par exemple, par : `Concat_Liste(&Tete1, Tete2);`

Solution 28 :

```
/* Procédure de tri d'une liste de nombres */
void Tri_Liste(Nombre **T)
{
  Nombre *P1, *P2;
  int x;
  P1 = *T ;
  while (P1->Suivant != NULL) {
    P2 = P1->Suivant ;
    while (P2 != NULL) {
      if (P1->Val > P2->Val) {
        x = P1->Val ;
        P1->Val = P2->Val ;
        P2->Val = x ;
      }
      P2 = P2->Suivant ;
    }
    P1 = P1->Suivant ;
```

```
  }
}
```

Remarque : Cette procédure peut être invoquée dans le programme de l'exercice 24, par exemple, par : `Tri_Liste(&Tete1);`

Solution 29 :

```
/* Procédure d'insertion d'un élément dans une liste triée de
nombres */
void Inserer_Val(int V, Nombre **T)
{
  Nombre *P1, *P2;
  /* Créer le nouvel élément */
  P2=(Nombre*)malloc(sizeof(Nombre));
  P2->Val = V ;
  P2->Suivant = NULL ;
  /* Le cas où le nouvel élément est inférieur à la tête */
  if (V <= (*T)->Val) {
    P2->Suivant = *T ;
    *T = P2 ;
  }
  else {
   /* Le cas où le nouvel élément est supérieur à la queue */
    P1 = *T ;
    while (P1->Suivant != NULL) P1 = P1->Suivant ;
    if (V >= P1->Val) P1->Suivant = P2;
      else {
       /* Insertion dans la liste */
        P1 = *T ;
        while (V >= P1->Suivant->Val) P1 = P1->Suivant ;
        P2->Suivant = P1->Suivant ;
        P1->Suivant = P2 ;
      }
    }
}
```


Remarque : Supposant que `Tete1` du programme de l'exercice 24 pointe vers une liste triée. Pour insérer la valeur `5` dans cette liste, on met : `Inserer_Val(5, &Tete1);`

Solution 30 :

```
/* Procédure de fusion de deux listes de nombres */
void Fusionner_Listes(Nombre **T1, Nombre *T2)
{
  Nombre *P1, *P2;
  while (T2 != NULL) {
    /* Isoler le premier élément de la deuxième liste */
    P2 = T2 ;
    T2 = T2->Suivant ;
    P2->Suivant = NULL ;
    /* Le cas où l'élément isolé est inférieur à la tête de la première liste */
```

```
    if (P2->Val <= (*T1)->Val) {
      P2->Suivant = *T1 ;
      *T1 = P2 ;
    }
    else {
      /* Le cas où l'élément isolé est supérieur à la queue de la première liste */
      P1 = *T1 ;
      while (P1->Suivant != NULL) P1 = P1->Suivant ;
      if (P2->Val >= P1->Val) P1->Suivant = P2;
      else {
        /* Insertion de l'élément isolé dans la première liste */
         P1 = *T1 ;
        while (P2->Val >= P1->Suivant->Val) P1 = P1->Suivant ;
         P2->Suivant = P1->Suivant ;
         P1->Suivant = P2 ;
      }
    }
  }
}
```

Remarque : Supposant que `Tete1` et `Tete2` du programme de l'exercice 24 pointent vers deux listes triées. Pour fusionner ces deux listes, on peut mettre : `Fusionner_Listes(&Tete1, Tete2);`

Solution 31 :

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
typedef struct personne
  {
    int Num;
    char Nom[20] ;
    struct personne *Precedent, *Suivant ;
  } Personne;
  Personne *Tete, *Queue ;
/* Procédure d'ajout d'un élément à la fin de la liste */
void Ajouter_E_Q(int N, char M[20])
{
  if(Tete == NULL){
      /* Créer et remplir le premier élément de la liste */
      Tete = (Personne*)malloc(sizeof(Personne)) ;
      Tete->Num = N;
      strcpy(Tete->Nom, M) ;
      Tete->Precedent = NULL;
      Queue = Tete ;
      Queue->Suivant = NULL ;
       }
   else {
      /* Ajout d'un élément à la fin de la liste */
      Queue->Suivant = (Personne*)malloc(sizeof(Personne));
      Queue->Suivant->Precedent = Queue;
```

```
      Queue = Queue->Suivant ;
      Queue->Num = N;
      strcpy(Queue->Nom, M) ;
      Queue->Suivant = NULL ;
       }
}
/* Procédure d'affichage des éléments de la liste */
void Afficher_Liste()
{
  Personne *P ;
  P = Tete;
  if (P != NULL) puts("Voici Les éléments de la liste :");
  else puts("Liste vide.");
  while (P != NULL) {
    printf("Num : %d,  Nom : %s\n", P->Num, P->Nom) ;
    P = P->Suivant;
  }
}
main()
{
  Tete = NULL; Queue = NULL;
  char c, s[20];
  int n;
  do{
    printf("Voulez-vous ajouter un élément à la liste o/n ? ");
    c = getchar(); getchar();
    if(c=='o'){
       printf("Donnez le num de la personne à ajouter : ");
       scanf("%d", &n);
       printf("Donnez le nom de la personne à ajouter : ");
       scanf("%s", s);getchar();
       Ajouter_E_Q(n, s);
       }
  } while(c != 'n');
  Afficher_Liste();
}
```


Pour obtenir une liste doublement chaînée circulaire, il faut ajouter après la création de la liste, les instructions :

```
Queue->Suivant = Tete ; Tete->Precedent = Queue ;
```

Solution 32 :

```
/* Procédure d'insertion d'un élément avant un autre dans une
liste doublement chaînée de personnes */
void Inserer_AV_Personne(int Nu1, char No1[20], int Nu2, char No2[20])
{
  Personne *P1, *P2;
{
  if ((Tete->Num == Nu2) && (strcmp(Tete->Nom, No2) == 0)) {
    P1 = (Personne*)malloc(sizeof(Personne));
    P1->Num = Nu1 ;
```

```
    strcpy(P1->Nom, No1);
    P1->Precedent = NULL ;
    P1->Suivant = Tete ;
    Tete->Precedent = P1;
    Tete = P1;
}
  else {
  /* Se pointer avant l'élément Pr2 */
    P2 = Tete ;
    while ((P2->Suivant->Num != Nu2) && (strcmp(P2->Suivant->Nom, No2) != 0))
              P2 = P2->Suivant;
  /* Créer un nouvel élément contenant Pr1 */
    P1 = (Personne*)malloc(sizeof(Personne));
    P1->Num = Nu1 ;
    strcpy(P1->Nom, No1) ;
  /* Insérer le nouvel élément dans la liste */
    P1->Precedent = P2 ;
    P1->Suivant = P2->Suivant ;
    P2->Suivant->Precedent = P1;
    P2->Suivant = P1;
  }
}
}
```


Remarque : Pour que cette procédure fonctionne correctement dans le programme de l'exercice 31, l'élément avant lequel on veut faire l'insertion doit exister obligatoirement dans la liste. Il est possible d'enrichir cette procédure pour traiter le cas où cet élément n'existe pas dans la liste.

Solution 33 :

```
/* Procédure de suppression des éléments en double dans la
liste des personnes */
void Supprimer_Double(Personne **Tete)
{
  Personne *P1, *P2, *D;
  if(*Tete != NULL)
  if((*Tete)->Suivant != NULL){
    P1 = *Tete;
    while (P1 != NULL){
      P2 = P1->Suivant;
      while(P2 != NULL){
      if((P1->Num == P2->Num) && (strcmp(P1->Nom,P2->Nom) == 0)) {
          D = P2; P2= P2->Suivant;
          D->Precedent->Suivant = D->Suivant;
          if (D->Suivant != NULL)
             D->Suivant->Precedent = D->Precedent;
          free(D);
       }
       else P2 = P2->Suivant;
       }
```

```
      P1 = P1->Suivant;
     }
  }
}
```

Remarque : Cette procédure peut être invoquée dans le programme de l'exercice 31, par exemple, par : `Supprimer_Double(&Tete);`

Solution 34 :

```
#include<stdio.h>
#include <stdlib.h>
  typedef struct pile
  {
    char Car ;
    struct pile *Precedent, *Suivant;
   } Pile ;
  Pile *Base, *Sommet ;
  int Choix ;
  char Element, reponse ;
/* Procédure permettant d'empiler un élément dans la pile */
void Empiler(char C)
{
  Pile *P ;
  P = (Pile*)malloc(sizeof(Pile));
  P->Car = C ;
  P->Precedent = NULL ;
  P->Suivant = Sommet ;
  if (Sommet != NULL) Sommet->Precedent = P;
  Sommet = P;
  if (Base == NULL) Base = Sommet ;
}
/* Procédure permettant de dépiler un élément de la pile */
void Depiler()
{
  if (Sommet == NULL) puts("Dépilement impossible : Pile Vide.");
  else if (Sommet->Suivant == NULL) {
                printf("Elément dépilé : %c\n", Sommet->Car) ;
                free(Sommet) ;
                Sommet = NULL ;
                Base = NULL ;
               }
               else {
                 printf("Elément dépilé : %c\n", Sommet->Car) ;
                  Sommet = Sommet->Suivant ;
                  free(Sommet->Precedent) ;
                  Sommet->Precedent = NULL ;
                 }
}
/* Procédure permettant l'affichage des éléments de la pile */
void Afficher_Pile()
{
```

```
  Pile *P;
  if (Sommet == NULL) puts("Aucun élément : Pile vide.");
  else {
   P = Sommet;
   puts("Les éléments de la pile cités du sommet vers la base :");
   while (P != NULL) {
      printf("%c\n", P->Car) ;
      P = P->Suivant;
    }
  }
  }
main()
{/* Initialement la pile est vide */
  Sommet = NULL ;   Base = NULL ;
  do {
    puts("Quelle Opération désirez-vous ?") ;
    puts("1- Empiler, 2- Dépiler, 3- Afficher.") ;
    scanf("%d", &Choix) ; getchar();
    switch (Choix)
    {
     case 1 : printf("Donnez le caractère à empiler : ") ;
           Element = getchar() ; getchar() ;
           Empiler(Element) ;
           break;
     case 2 : Depiler() ; break;
     case 3 : Afficher_Pile(); break;
     default : puts("Choix non Accepté !") ;
    }
    printf("Voulez-vous un autre test o/n ? ") ;
    reponse = getchar() ; getchar();
  }
  while (reponse != 'n') ;
}
```


Solution 35 :

```
#include<stdio.h>
#include <stdlib.h>
  typedef struct file
  {
    char Car ;
    struct file *Precedent, *Suivant ;
  } File ;
  File *Queue, *Tete ;
  int Choix ;
  char Element, reponse ;
/* Procédure permettant d'ajouter un élément à la file */
void Ajouter_File(char C)
{
  if (Queue == NULL ) {
    Tete = (File*)malloc(sizeof(File));
```

```
    Tete->Car = C ;
    Tete->Precedent = NULL ;
    Queue = Tete ;
    Queue->Suivant = NULL ;
  }
  else {
    Queue->Suivant = (File*)malloc(sizeof(File));
    Queue->Suivant->Precedent = Queue ;
    Queue = Queue->Suivant;
    Queue->Car = C ;
    Queue->Suivant = NULL ;
  }
}
/* Procédure permettant de supprimer un élément de la file */
void Supprimer_File()
{
  if (Tete == NULL) puts("Suppression impossible : File Vide.");
  else if (Tete->Suivant == NULL) {
                printf("Elément Supprimé : %c\n", Tete->Car) ;
                free(Tete) ;
                Tete = NULL ;
                Queue = NULL ;
               }
               else {
                 printf("Elément supprimé : %c\n", Tete->Car) ;
                  Tete = Tete->Suivant ;
                  free(Tete->Precedent) ;
                  Tete->Precedent = NULL ;
                 }
}
/* Procédure permettant l'affichage des éléments de la file */
void Afficher_File()
{
  File *P;
  if (Tete == NULL) puts("Aucun élément : File vide.");
  else {
    P = Tete;
    puts("Les éléments de la file cités de la tête vers la queue :") ;
    while (P != NULL) {
      printf("%c\n", P->Car) ;
      P = P->Suivant;
    }
  }
}
main()
{
/* Initialement la file est vide */
  Tete = NULL ;   Queue = NULL ;
  do{
    puts("Quelle Opération désirez-vous ?") ;
```

```
    puts("1- Ajouter, 2- Supprimer, 3- Afficher.") ;
    scanf("%d", &Choix); getchar();
    switch (Choix)
    {
     case 1 : printf("Donnez le caractère à ajouter à la file :") ;
           Element = getchar() ;getchar();
           Ajouter_File(Element) ;
           break;
     case 2 : Supprimer_File() ; break;
     case 3 : Afficher_File(); break;
     default : puts("Choix non Accepté !") ;
    }
    printf("Voulez-vous un autre test o/n ? ") ;
    reponse = getchar() ; getchar();
   }
  while (reponse != 'n') ;
}
```

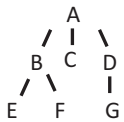
Chapitre 10 : Les arbres

1. Introduction

En informatique, on manipule parfois des structures de données récursives générales appelées les arbres. Il s'agit d'un cas particulier des graphes qui seront vus dans le chapitre suivant.

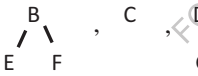
2. Définitions

Un arbre : C'est un ensemble de nœuds (sommets) reliés par des arcs (liens, branches ou arêtes) formant une hiérarchie. Un arbre peut être représenté graphiquement comme suit :



Chaque sommet ou nœud (`A`, `B`, `C`, `D`, `E`, `F`, `G`) possède un certain nombre de fils. Par exemple, `B`, `C` et `D` sont les fils de `A`. On dit aussi que `B`, `C` et `D` sont des frères. Lorsqu'un nœud n'a pas de fils (ici `E`, `F`, `C` et `G`), on dit que c'est une feuille de l'arbre.

Inversement, chaque nœud possède un seul père. Par exemple, `A` est le père de `B`, `C` et `D`. Le nœud particulier qui n'a pas de père (ici `A`) est appelé racine de l'arbre.



sont des sous-arbres de l'arbre ci-dessus

Niveau ou profondeur : Le niveau de la racine de l'arbre est égal à `1`. Le niveau d'un nœud, autre que la racine, est égal au niveau de son père plus `1`. Par exemple, pour l'arbre ci-dessus, les nœuds `E`, `F` et `G`, sont de même niveau (niveau `3`).

Mot des feuilles d'un arbre : Le mot des feuilles d'un arbre est la chaîne formée, de gauche à droite, des valeurs des feuilles de l'arbre. Par exemple, pour l'arbre ci-dessus, le mot des feuilles est égal à : `EFCG`.

Taille d'un arbre : La taille d'un arbre est égale au nombre de nœuds de cet arbre. La taille d'un arbre vide est égale à `0`. La taille de l'arbre précédent est égale à `7`.

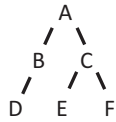
Hauteur d'un arbre : C'est la distance entre la feuille la plus éloignée et la racine. La hauteur d'un arbre vide est égale à `0`. La hauteur de l'arbre précédent est égale à `3`.

Un arbre n-aire : Un arbre n-aire est un arbre pour lequel chaque nœud admet au plus *n* fils. Le *n* est dit arité de l'arbre. L'arbre ci-dessus est donc un arbre 3-aire ou encore ternaire.

3. Arbre binaire

3.1. Définition

Un arbre binaire est un arbre n-aire pour lequel *n* est égal à `2`, i.e., c'est un arbre dans lequel un nœud peut avoir `0`, `1` ou `2` fils. Il peut être représenté sous forme graphique ou sous forme parenthèsée.



Forme graphique

A(B(D,), C(E, F))

Forme parenthèsée

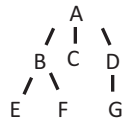
Dans le cas des arbres binaires, pour chaque nœud donné, on parle de fils gauche et de fils droit.

3.2. Passage d'un arbre n-aire à un arbre binaire

Tout arbre n-aire peut être représenté sous forme binaire : le premier fils gauche d'un nœud devient le fils gauche de ce nœud ; le premier frère droit d'un nœud devient le fils droit de ce nœud.

Exemple :

L'exemple suivant illustre la transformation d'un arbre n-aire en un arbre binaire équivalent :



Arbre 3-aire ou ternaire

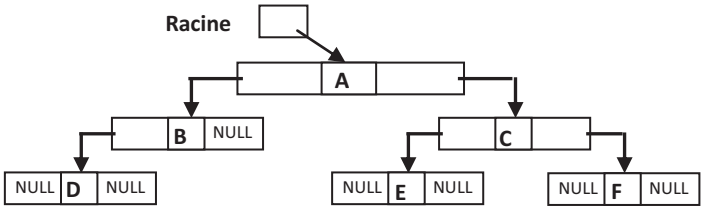
Arbre binaire équivalent

Puisque chaque arbre n-aire peut être représenté par un arbre binaire équivalent, nous limiterons notre étude dans le reste de ce chapitre aux arbres binaires.

3.3. Représentation chaînée d'un arbre binaire

Un arbre binaire peut être représenté en mémoire par une liste chaînée. Chaque nœud de la liste est une structure qui contient des informations à stocker et deux pointeurs vers les nœuds fils.

L'arbre A(B(D,), C(E, F)) peut être représenté par la liste chaînée suivante :



Où racine est une variable de type pointeur contenant l'adresse du nœud racine de l'arbre.

Exercice : Ecrire le programme C permettant de créer l'arbre ci-dessus en utilisant une représentation chaînée.

Solution :

```
#include <stdio.h>
#include <stdlib.h>
/* Structure d'un nœud de l'arbre */
typedef struct arbre {
    char Car ;
    struct arbre *F_gauche, *F_droit;
} Arbre ;
  Arbre *Racine ;
/* Procédure d'ajout d'un fils gauche à un nœud */
void Ajouter_FG(Arbre *R, char C)
{
  R->F_gauche = (Arbre*)malloc(sizeof(Arbre));
  R->F_gauche->Car = C ;
  R->F_gauche->F_gauche = NULL ;
  R->F_gauche->F_droit = NULL ;
}
/* Procédure d'ajout d'un fils droit à un nœud */
void Ajouter_FD(Arbre *R, char C)
{
  R->F_droit = (Arbre*)malloc(sizeof(Arbre));
  R->F_droit->Car = C ;
  R->F_droit->F_gauche = NULL ;
  R->F_droit->F_droit = NULL ;
}
/* Programme Principal */
main(){
  /* Créer et remplir la racine de l'arbre */
  Racine = (Arbre*)malloc(sizeof(Arbre));
  Racine->Car = 'A' ;
  Racine->F_gauche = NULL ;
  Racine->F_droit = NULL ;
  /* Ajouter le reste des éléments à l'arbre */
  Ajouter_FG(Racine, 'B');
  Ajouter_FG(Racine->F_gauche, 'D');
  Ajouter_FD(Racine, 'C');
```

```
  Ajouter_FG(Racine->F_droit, 'E');
  Ajouter_FD(Racine->F_droit, 'F');
}
```

3.4. Parcours d'un arbre binaire

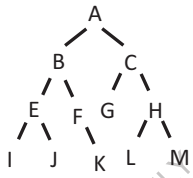
Dans ce cours, on s'intéresse à trois manières classiques pour parcourir un arbre binaire qui sont des cas particuliers du parcours en profondeur : parcours préfixé, parcours infixé et parcours postfixé.

3.4.1. Parcours préfixé (préordre ou RGD)

On visite un nœud, ensuite son fils gauche, ensuite son fils droit.

Exemple :

Soit l'arbre suivant :



Cet arbre peut être traité (en parcours préfixé) dans l'ordre suivant : `A B E I J F K C G H L M`. La procédure C récursive du parcours préfixé d'un arbre binaire s'écrit comme suit :

```
void RGD(Arbre *R)
{
  if (R != NULL) {
    Traiter(R) ;
    RGD(R->F_gauche) ;
    RGD(R->F_droit) ;
  }
}
```


`Traiter` étant une procédure qui effectue le traitement désiré sur le nœud pris comme paramètre.

Exercice : Ecrire en C la procédure permettant d'afficher en RGD l'arbre de caractères créé précédemment.

Solution :

```
/* Procédure d'affichage des éléments d'un arbre en RGD */
void Afficher_RGD(Arbre *R)
{
  if (R != NULL) {
    printf("%c", R->Car);
    Afficher_RGD(R->F_gauche);
    Afficher_RGD(R->F_droit);
  }
}
```

La procédure sera invoquée dans le programme principal par : `Afficher_RGD(Racine);`

3.4.2. Parcours infixé (projectif, symétrique ou encore GRD)

On commence par visiter le fils gauche d'un nœud, ensuite le nœud lui-même, enfin le fils droit de ce nœud.

Pour l'arbre précédent, on traite les nœuds (en parcours infixé) dans l'ordre suivant : `I E J B F K A G C L H M`. La procédure récursive du parcours infixé d'un arbre binaire s'écrit comme suit :

```
void GRD(Arbre *R)
{
  if (R != NULL) {
    GRD(R->F_gauche) ;
    Traiter(R) ;
    GRD(R->F_droit) ;
  }
}
```

3.4.3. Parcours postfixé (ordre terminal ou GDR)

On commence par visiter le fils gauche d'un nœud, ensuite le fils droit de ce nœud, enfin le nœud lui-même.

Pour l'arbre précédent, on traite les nœuds (en parcours postfixé) dans l'ordre suivant : `I J E K F B G L M H C A`. La procédure récursive du parcours postfixé d'un arbre binaire s'écrit comme suit :

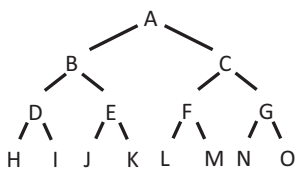
```
void GDR(Arbre *R)
{
  if (R != NULL) {
    GDR(R->F_gauche) ;
    GDR(R->F_droit) ;
    Traiter(R) ;
  }
}
```


3.5. Arbres binaires particuliers

3.5.1. Arbre binaire complet

On dit qu'un arbre binaire est complet si chaque nœud autre qu'une feuille admet deux descendants, et si toutes les feuilles sont au même niveau.

Exemple :



La taille d'un arbre binaire complet est égale à $2^n - 1$, où n est le niveau des feuilles.

3.5.2. Arbre dégénéré

Un arbre est dit dégénéré si tous les nœuds de cet arbre ont au plus un fils.

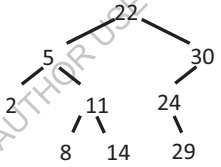
Exemple :

Un arbre dégénéré est équivalent à une liste simplement chaînée.

3.5.3. Arbre binaire ordonné

Soit une relation d'ordre (noté <=) définie sur l'ensemble des valeurs attachées aux nœuds d'un arbre binaire. Un arbre binaire est dit ordonné (appelé aussi arbre binaire de recherche) si la chaîne infixée des valeurs, correspondant au parcours infixé, est ordonnée.

Exemple :



Le parcours infixé de l'arbre ci-dessus donne : `2-5-8-11-14-22-24-29-30`.

Remarques :

- Un arbre binaire de recherche ne peut pas contenir plusieurs fois la même valeur.
- Chaque nœud est supérieur à tous les nœuds se trouvant dans son sous-arbre de gauche (s'il existe), et inférieur à tous les nœuds se trouvant dans son sous-arbre de droite (s'il existe).
- Chaque nœud est compris entre deux nœuds : (1) son prédécesseur, qui est le maximum de son sous-arbre gauche (s'il existe), et (2) son successeur, qui est le minimum de son sous-arbre droit (s'il existe).

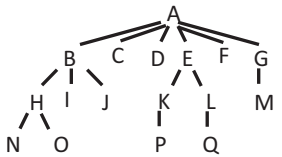
4. Exercices corrigés

4.1. Exercices

Exercice 1 :

Proposez une structure pour stocker un arbre n-aire contenant des valeurs entières.

Exercice 2 :
Transformez l'arbre n-aire suivant en arbre binaire équivalent.



Arbre 6-aire

Exercice 3 :
Ecrire un programme C permettant de créer et d'afficher les éléments d'un arbre binaire de caractères. Pour la création d'un nœud, utilisez une procédure récursive. Pour l'affichage, faites le parcours RGD de l'arbre binaire.

Exercice 4 :
Ecrire en C une fonction récursive booléenne permettant la recherche d'un élément dans un arbre binaire de caractères. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et un caractère indiquant l'élément recherché.

Exercice 5 :
Modifier la fonction de l'exercice précédent de telle sorte qu'elle retourne un pointeur vers l'élément recherché. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et un caractère indiquant l'élément recherché.

Exercice 6 :
Ecrire en C une fonction qui retourne le niveau d'un nœud dans un arbre binaire de caractères. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et le contenu du nœud pour lequel on désire déterminer le niveau.

Exercice 7 :
Ecrire en C une fonction récursive qui retourne le nombre de feuilles d'un arbre binaire. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 8 :
Ecrire en C une procédure qui affiche le mot des feuilles d'un arbre binaire de caractères. La procédure possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 9 :
Ecrire en C une fonction qui retourne la taille d'un arbre binaire. La taille d'un arbre binaire est égale à 1 (l'élément racine) plus la taille du sous-arbre gauche plus la taille du sous-arbre droit. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 10 :
Ecrire en C une fonction booléenne permettant de déterminer si un arbre binaire est dégénéré ou non. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 11 :
Ecrire en C une fonction qui retourne la hauteur d'un arbre binaire. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 12 :
Ecrire en C une fonction booléenne permettant de dire si un arbre binaire est équilibré ou non. Un arbre est équilibré si, pour chacun de ses nœuds, la différence entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit est d'au plus une unité. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 13 :
Soit un arbre binaire d'entiers dont la structure de chaque élément est décrite comme suit :

```
typedef struct arbre
{
    int Val ;
    struct arbre *F_gauche, *F_droit;
  } Arbre ;
```


Ecrire la fonction qui calcule la somme des valeurs des nœuds d'un arbre binaire d'entiers. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 14 :
Ecrire en C une fonction qui détermine la plus grande valeur des nœuds d'un arbre binaire d'entiers (décrit dans l'exercice 13). La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 15 :
Ecrire en C une fonction qui détermine la plus petite des valeurs des nœuds d'un arbre binaire d'entiers. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 16 :
Ecrire en C une fonction booléenne permettant de dire si un arbre binaire d'entiers est ordonné (arbre de recherche) ou non. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire.

Exercice 17 :
Ecrire en C une procédure permettant de libérer l'espace mémoire occupé par un arbre binaire d'entiers. La procédure possède comme paramètre un pointeur vers un nœud de l'arbre binaire passé par adresse.

4.2. Corrigés

Solution 1 :

Une première proposition consiste à définir la structure comme suit :

- Une variable entière correspondant au contenu du nœud.
- Un tableau de pointeurs vers les nœuds fils.

```
typedef struct arbre_naire
{
  int Val ;
  struct arbre_naire *Fils[10] ;
} Arbre_naire;
```

On peut également définir la structure comme suit :

- Une variable entière correspondant au contenu du nœud.
- Un pointeur vers le nœud fils.
- Un pointeur vers le nœud frère.

```
typedef struct arbre_naire
{
    int Val;
    struct arbre_naire *Fils, *Frere;
} Arbre_naire;
```

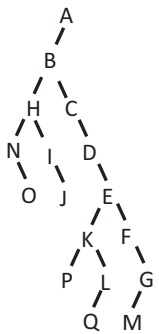
Une autre manière pour définir la structure :

- Une variable entière correspondant au contenu du nœud.
- Un pointeur vers le nœud père.

```
typedef struct arbre_naire
{
    int Val;
    struct arbre_naire *Pere;
} Arbre_naire;
```


Solution 2 :

Arbre binaire équivalent :



Solution 3 :

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

```
typedef struct arbre {
    char Car ;
    struct arbre *F_gauche, *F_droit;
} Arbre ;
  Arbre *Racine ;
/* Procédure récursive pour la création d'un arbre binaire */
void Creer_Noeud(Arbre **R)
{ char C, reponse;
  printf("Donnez la valeur du noeud : ");
  C = getchar();getchar();
  *R = (Arbre*)malloc(sizeof(Arbre));
  (*R)->Car = C ;
  (*R)->F_gauche = NULL ;
  (*R)->F_droit = NULL ;
  printf("Le noeud %c possède t-il un fils gauche o/n ? ", C);
  reponse = getchar();getchar();
  if (reponse == 'o') Creer_Noeud(&(*R)->F_gauche);
  printf("Le noeud %c possède t-il un fils droit o/n ? ", C);
  reponse = getchar();getchar();
  if (reponse == 'o') Creer_Noeud(&(*R)->F_droit);
}
/* Procédure d'affichage des éléments d'un arbre en RGD */
void Afficher_RGD(Arbre *R)
{
  if (R != NULL) {
    printf("%c", R->Car);
    Afficher_RGD(R->F_gauche);
    Afficher_RGD(R->F_droit);
  }
}
/* Programme Principal */
main(){
  /* Créer et remplir l'arbre */
  puts("Créer un arbre binaire : le premier élément est la racine de l'arbre.");
  Creer_Noeud(&Racine);
  puts("Voici l'arbre binaire affiché en RGD :");
  Afficher_RGD(Racine);
}
```


Solution 4 :

```
/* Fonction de recherche d'un élément dans un arbre binaire de
caractères */
int Recherche_Arbre(Arbre *R, char C)
{
 int Trouve = 0;
 if (R != NULL)
  if (R->Car == C) Trouve = 1;
  else Trouve = Recherche_Arbre(R->F_gauche, C) || Recherche_Arbre(R->F_droit, C);
 return Trouve ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
if (Recherche_Arbre(Racine, 'S')) puts("Existe");
else puts("N'existe pas");
```

Solution 5 :

```
/* Fonction de recherche d'un élément dans un arbre binaire de
caractères */
Arbre *Recherche_Arbre(Arbre *R, char C)
{
  Arbre *P ;
  P = NULL ;
  if (R != NULL)
    if (R->Car == C) P = R;
      else {
        P = Recherche_Arbre(R->F_gauche, C) ;
        if (P == NULL) P = Recherche_Arbre(R->F_droit, C);
      }
  return P ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
if (Recherche_Arbre(Racine,'S') != NULL) puts("Existe");
else puts("N'existe pas");
```


Solution 6 :

```
/* Fonction qui détermine le niveau d'un nœud dans un arbre
binaire de caractères */
int Niveau_Noeud(Arbre *R, char C, int np)
{
 int n = 100; // 100 étant la hauteur max qu'on estime pour notre arbre
 if (R != NULL)
 if (R->Car == C) n = np + 1;
 else if (Niveau_Noeud(R->F_droit, C, np+1) < Niveau_Noeud(R->F_gauche, C, np+1))
            n = Niveau_Noeud(R->F_droit, C, np+1);
      else n = Niveau_Noeud (R->F_gauche, C, np+1);
  return n ;
}
```

Remarque : Le nœud `C` doit exister dans l'arbre. `np` veut dire le niveau du père ; lors de l'invocation de la fonction dans le programme, `np` aura `0`. Pour afficher, par exemple, le niveau du nœud ayant comme valeur le caractère `'A'`, on met : `printf("Le niveau du caractère 'A' : %d", Niveau_Noeud(Racine, 'A', 0));`

Solution 7 :

```
/* Fonction de calcul du nombre de feuilles d'un arbre binaire
de caractères */
int Nbr_Feuilles_Arbre(Arbre *R)
{
  int Nbr;
```

```
  if (R == NULL) Nbr = 0;
  else if ((R->F_gauche == NULL)&&(R->F_droit == NULL)) Nbr = 1;
      else Nbr = Nbr_Feuilles_Arbre(R->F_gauche) + Nbr_Feuilles_Arbre(R->F_droit);
  return Nbr ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
printf("Le nombre de feuilles de l'arbre est : %d", Nbr_Feuilles_Arbre (Racine));
```

Solution 8 :

```
/* Procédure qui affiche le mot des feuilles d'un arbre binaire de
caractères */
void Mot_Feuilles_Arbre(Arbre *R)
{
  if (R != NULL)
    if ((R->F_gauche == NULL)&&(R->F_droit == NULL))
        putchar(R->Car);
      else {
         Mot_Feuilles_Arbre(R->F_gauche);
         Mot_Feuilles_Arbre(R->F_droit);
      }
}
```

Remarque : Cette procédure peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit : `Mot_Feuilles_Arbre(Racine) ;`

Solution 9 :

```
/* Fonction qui calcule la taille d'un arbre binaire de
caractères */
int Taille_Arbre(Arbre *R)
{ int T;
  if (R == NULL) T = 0;
  else T = 1 + Taille_Arbre(R->F_gauche) + Taille_Arbre(R->F_droit);
  return T ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
printf("La taille de l'arbre est : %d", Taille_Arbre (Racine)) ;
```

Solution 10 :

```
/* Fonction qui détermine si un arbre est dégénéré ou non */
int Degenere_Arbre(Arbre *R)
{
 int Degenere;
 Degenere = 1 ;
 if (R != NULL)
  if ((R->F_gauche != NULL) && (R->F_droit != NULL)) Degenere = 0;
  else Degenere = Degenere_Arbre(R->F_gauche) && Degenere_Arbre(R->F_droit);
 return Degenere ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
if (Degenere_Arbre(Racine)) puts("Dégénéré");
else puts("Non dégénéré");
```

Solution 11 :

```
/* Fonction qui détermine la hauteur d'un arbre binaire */
int Hauteur_Arbre(Arbre *R)
{
  int H1, H2 ;
  H1 = 0 ;
  H2 = 0 ;
  if (R != NULL) {
        H1 = 1 + Hauteur_Arbre(R->F_gauche) ;
        H2 = 1 + Hauteur_Arbre(R->F_droit);
      }
  if (H1 > H2) return H1;
    else return H2 ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
printf("La hauteur de l'arbre est : %d", Hauteur_Arbre (Racine)) ;
```

Solution 12 :

```
/* Fonction qui détermine si un arbre est équilibré ou non */
int Equilibre_Arbre(Arbre *R)
{
int equilibre;
equilibre = 1 ;
if (R != NULL)
 if (abs(Hauteur_Arbre(R->F_gauche) - Hauteur_Arbre(R->F_droit)) > 1)  equilibre = 0;
 else equilibre = Equilibre_Arbre(R->F_gauche) && Equilibre_Arbre(R->F_droit);
return equilibre ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C de l'exercice 3, par exemple, comme suit :

```
if (Equilibre_Arbre(Racine)) puts("Equilibré");
else puts("Non équilibré");
```

Solution 13 :

```
/* Fonction qui calcule la somme des éléments d'un arbre
binaire d'entiers */
int Somme_Arbre(Arbre *R)
{
  int S;
  if (R == NULL) S = 0;
    else S = R->Val + Somme_Arbre(R->F_gauche) + Somme_Arbre(R->F_droit);
  return S ;
}
```

Remarque : Après avoir créé un arbre binaire d'entiers, la fonction peut être invoquée dans le programme C, par exemple, comme suit :

```
printf("Somme arbre : %d", Somme_Arbre(Racine));
```

Solution 14 :

```
/* Fonction qui détermine le maximum dans un arbre binaire
d'entiers */
int Max_Arbre(Arbre *R)
{
int Max;
if (R != NULL) {
Max = R->Val ;
if (R->F_gauche != NULL)
 if (Max < Max_Arbre(R->F_gauche)) Max = Max_Arbre(R->F_gauche) ;
 if (R->F_droit != NULL)
   if (Max < Max_Arbre(R->F_droit)) Max = Max_Arbre(R->F_droit) ;
}
return Max ;
}
```

Remarque : L'arbre binaire d'entiers ne doit pas être vide. La fonction peut être invoquée dans le programme C, par exemple, comme suit :

```
printf("Max arbre : %d", Max_Arbre(Racine));
```

Solution 15 :

```
/* Fonction qui détermine le minimum dans un arbre binaire
d'entiers */
int Min_Arbre(Arbre *R)
{
int Min;
if (R != NULL) {
 Min = R->Val ;
 if (R->F_gauche != NULL)
  if (Min > Min_Arbre(R->F_gauche)) Min = Min_Arbre(R->F_gauche) ;
  if (R->F_droit != NULL)
   if (Min > Min_Arbre(R->F_droit)) Min = Min_Arbre(R->F_droit) ;
}
return Min ;
}
```

Remarque : L'arbre binaire d'entiers ne doit pas être vide. La fonction peut être invoquée dans le programme C, par exemple, comme suit :

```
printf("Min arbre : %d", Min_Arbre(Racine));
```

Solution 16 :

```
/* Fonction qui détermine si un arbre binaire d'entiers est
ordonné ou non */
int Ordonne_Arbre(Arbre *R)
{
int Ord;
Ord = 1 ;
if (R != NULL) {
```

```
 if (R->F_gauche != NULL) if (Max_Arbre(R->F_gauche) > R->Val) Ord = 0;
 if (R->F_droit != NULL) if (Min_Arbre(R->F_droit) < R->Val) Ord = 0;
 Ord = Ordonne_Arbre(R->F_gauche) && Ordonne_Arbre(R->F_droit) && Ord;
}
return Ord ;
}
```

Remarque : Cette fonction peut être invoquée dans le programme C, par exemple, comme suit :

```
if (Ordonne_Arbre(Racine)) puts("Ordonné");
else puts("Non ordonné");
```

Solution 17 :

```
/* Procédure de libération de l'espace mémoire occupé par un
arbre binaire */
void Liberer_Arbre(Arbre **R)
{
  Arbre *P = *R;
  if (P != NULL){
       Liberer_Arbre(&P->F_gauche);
       Liberer_Arbre(&P->F_droit);
       free(P);
  }
  *R = NULL;
}
```

Remarque : Cette procédure peut être invoquée dans le programme C comme suit : `Liberer_Arbre(&Racine);`

Chapitre 11 : Les graphes

1. Introduction

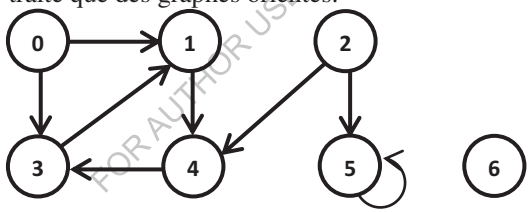
Les graphes sont des structures de données très générales dont les listes et les arbres ne sont que des cas particuliers. Les graphes permettent de modéliser de nombreux problèmes algorithmiques.

2. Définitions

Un graphe est un ensemble de nœuds (sommets) reliés par des liens (arcs). Ce n'est plus un arbre dès qu'il existe deux parcours différents pour aller au moins d'un nœud à un autre.

Un graphe est *connexe* lorsqu'il est possible de trouver au moins un parcours permettant de relier les nœuds deux à deux (un arbre est un graphe connexe ; deux arbres forment un graphe non connexe).

Un graphe est dit *pondéré* lorsque chaque lien est associé à une valeur (appelée poids). On utilisera un graphe pondéré, par exemple, pour répondre à la question : quelle est la route la plus courte pour aller d'une ville à une autre ?
Un graphe est dit *orienté* lorsque les liens sont unidirectionnels. Dans le reste du cours, on ne traite que des graphes orientés.



Exemple d'un graphe orienté

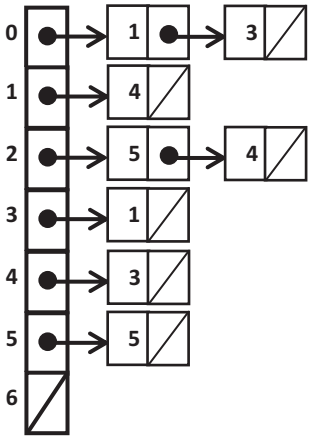
Un graphe est dit *acyclique* s'il ne contient aucun cycle. Un *cycle* est un chemin permettant de revenir à un sommet de départ en passant par tous les sommets du chemin. Un *chemin* étant une suite d'arcs où le nœud cible d'un arc correspond au nœud source de l'arc suivant, mais pas nécessairement pour le dernier arc du chemin. La longueur d'un chemin est égale au nombre d'arcs qui le composent.

Un graphe est dit *fortement connexe* si pour chaque deux sommets i et j, soit $i=j$ ou bien il existe un chemin entre i et j et un autre entre j et i.

3. Représentation d'un graphe

3.1. Liste d'adjacence

On peut représenter un graphe par une *liste d'adjacence* qui est un tableau de N listes avec N est le nombre de sommets. La $i^{ème}$ liste du tableau correspond aux successeurs du $i^{ème}$ sommet.



La liste d'adjacence représentant le graphe présenté précédemment

La structure, permettant de représenter le graphe, est définie en C comme suit :

```
#define nbrs 7
typedef struct succ
{
   int Sommet ;
   struct succ * Suivant ;
  } Succ ;

Succ *G[nbrs] ;
```


3.2. Matrice d'adjacence

Une autre solution consiste à numéroter les *N* sommets et utiliser une matrice carrée *N***N* dite *matrice d'adjacence*, avec la valeur 0 s'il n'y a pas un lien entre les nœuds *i* et *j*, et la valeur 1 s'il y a un lien entre ces deux nœuds. Pour des graphes non orientés, la matrice est symétrique par rapport à la diagonale.

	0	1	2	3	4	5	6
0	0	1	0	1	0	0	0
1	0	0	0	0	1	0	0
2	0	0	0	0	1	1	0
3	0	1	0	0	0	0	0
4	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0

La matrice d'adjacence représentant le graphe présenté précédemment

Cette matrice peut être déclarée en C comme suit :

```
#define nbrs 7
int M[nbrs][nbrs];
```

Une représentation par matrice est surtout intéressante lorsqu'il y a beaucoup de liens (graphe presque complet). La représentation par liste d'adjacence étant moins gourmande en mémoire pour les graphes comportant peu de liens par nœud.

La matrice d'adjacence permet aussi de calculer le nombre de chemins entre deux sommets par $(M^k)_{i,j}$; avec M la matrice d'adjacence et k la longueur du chemin entre les sommets i et j. A partir de là, on peut déduire qu'il existe un chemin entre deux sommets i et j si est seulement si $N_{i,j} <> 0$ avec $N = M + M^2 + ... + M^n$. On peut maintenant définir la *fermeture transitive* d'un graphe commettant un nouveau graphe avec les mêmes sommets, où deux sommets sont reliés par un arc si et seulement s'il existe un chemin entre ces deux sommets dans le graphe original.

La fermeture transitive peut être utilisée pour déterminer les composantes fortement connexes d'un graphe. Pour cela, on doit suivre les étapes suivantes :

1. Déterminer la matrice F correspondant à la fermeture transitive du graphe.
2. Rendre la matrice symétrique par $F_{i,j} = F_{i,j} * F_{j,i}$.
3. On ajoute la relation de réflexivité, i.e. pour chaque sommet i on met $F_{i,i} = 1$.

Les composantes fortement connexes sont obtenues ligne par ligne dans la matrice résultante.

Notons aussi que la représentation par matrice d'adjacence concentre sur les liens entre les sommets. Si chaque sommet du graphe est associé à un ensemble d'informations, il est possible de définir encore un tableau de structure. Chaque élément de ce tableau correspond à un sommet. La structure dépend des informations qu'on veut associer à chaque sommet, par exemple, nom, numéro, etc.

4. Parcours d'un graphe

Un graphe peut être parcouru de deux manières selon l'ordre de visite des nœuds : parcours en profondeur d'abord ou parcours en largeur d'abord.

4.1. Parcours en largeur d'abord

Dans le parcours en largeur d'abord, un sommet s est fixé comme origine et l'on visite tous les sommets situés à distance k de s avant tous les sommets situés à distance $k+1$.

Par exemple, à partir du sommet 0, le parcours du graphe précédent en largeur d'abord permet de visiter les sommets dans l'ordre suivant : `0`, `1`, `3`, `4`, `2`, `5`, `6`.

4.2. Parcours en profondeur d'abord

Le principe du parcours en profondeur d'abord est de visiter tous les sommets en allant d'abord du plus profondément possible dans le graphe.

Par exemple, à partir du sommet 0, le parcours du graphe précédent en profondeur d'abord permet de visiter les sommets dans l'ordre suivant : `0`, `1`, `4`, `3`, `2`, `5`, `6`.

5. Exercices corrigés

5.1. Exercices

Exercice 1 :
Ecrire un programme C permettant de créer et d'afficher le graphe du cours en utilisant une liste d'adjacence.

Exercice 2 :
Ecrire un programme C permettant de créer le graphe du cours en utilisant une matrice d'adjacence, ensuite afficher ce graphe.

Exercice 3 :
Ecrire en C une procédure permettant de transformer la structure du premier exercice en structure du deuxième exercice. Autrement dit, la procédure doit remplir la matrice d'adjacence à partir de la liste d'adjacence déjà créée.

Exercice 4 :
Ecrire en C une procédure permettant d'afficher les nœuds du graphe implémenté sous forme de matrice d'adjacence en utilisant la stratégie du parcours en largeur d'abord à partir du sommet numéro 0.

Exercice 5 :
Ecrire en C une procédure permettant d'afficher les nœuds du graphe implémenté sous forme de matrice d'adjacence en utilisant la stratégie du parcours en profondeur d'abord à partir du sommet 0.

Exercice 6 :
Ecrire un programme C permettant de créer et d'afficher un graphe de `nbrs` sommets représenté par une matrice d'adjacence *M(nbrs*nbrs)*. `nbrs` étant une constante indiquant le nombre de sommets. Le programme doit utiliser deux procédures : une pour la création et une autre pour l'affichage. Vous pouvez choisir, par exemple, *nbrs = 5*.

Exercice 7 :
Ecrire en C une fonction booléenne permettant de tester l'existence d'un chemin entre deux sommets d'un graphe représenté par une matrice d'adjacence *M(nbrs*nbrs)*.

Exercice 8 :
Ecrire en C une procédure permettant de déterminer la fermeture transitive d'un graphe représenté par une matrice d'adjacence *M(nbrs*nbrs)*.

Exercice 9 :
Ecrire en C une fonction permettant de dire si un graphe représenté par une matrice d'adjacence *M(nbrs*nbrs)* est fortement connexe ou non.

Exercice 10 :
Ecrire en C une procédure permettant de déterminer et d'afficher les composantes fortement connexes d'un graphe représenté par une matrice d'adjacence *M(nbrs*nbrs)*.

5.2. Corrigés

Solution 1 :

```
#include<stdio.h>
#include<stdlib.h>
#define nbrs 7
// Définition de la liste d'adjacence
typedef struct succ
{
   int Sommet ;
   struct succ * Suivant ;
  } Succ ;
Succ *G[nbrs] ;
int i;
Succ *P;
/* Procédure d'ajout d'un successeur à un sommet */
void Ajouter_Succ(int S, int Adj){
       if(G[S] == NULL){
         G[S] = (Succ*)malloc(sizeof(Succ));
         G[S]->Sommet = Adj;
         G[S]->Suivant = NULL;
       } else{
             P = G[S];
             while(P->Suivant != NULL) P = P->Suivant;
             P->Suivant = (Succ*)malloc(sizeof(Succ));
             P = P->Suivant;
             P->Sommet = Adj;
             P->Suivant = NULL;
       }
}
/* Procédure d'affichage du graphe */
void Afficher_Graphe()
{
       for(i=0; i<nbrs; i++){
             printf("Sommet %d\n", i);
             if (G[i] != NULL){
                   printf("\tSes successeurs sont : ", i);
                   P = G[i];
                   while(P!=NULL){
                         printf("  %d", P->Sommet);
                         P = P->Suivant;
                   }
                   printf("\n");
             }
       }
}
main(){
    // Initialiser la liste d'adjacence
    for (i=0; i<nbrs; i++) G[i] = NULL;
    // Création du graphe
```

```
    Ajouter_Succ(0,1);
    Ajouter_Succ(0,3);
    Ajouter_Succ(1,4);
    Ajouter_Succ(2,5);
    Ajouter_Succ(2,4);
    Ajouter_Succ(3,1);
    Ajouter_Succ(4,3);
    Ajouter_Succ(5,5);
    // Affichage
    Afficher_Graphe();
}
```

Solution 2 :

```
#include<stdio.h>
#define nbrs 7
// Définition de la matrice d'adjacence
int M[nbrs][nbrs];
/* Procedure d'affichage des éléments du graphe */
void Afficher_Graphe()
{ int i, j;
  for (i=0; i<nbrs; i++){
    printf("Sommet : %d\n", i);
    for (j=0; j<nbrs; j++)
      if (M[i][j] == 1) printf("\tSuccesseur : %d\n", j);
    }
}
main() /* Programme principal */
{
  /* Initialiser la matrice d'adjacence */
  int k, l;
  for (k=0; k<nbrs; k++)
    for (l=0; l<nbrs; l++) M[k][l] = 0;
  /* Création du graphe */
  M[0][1] = 1 ;
  M[0][3] = 1 ;
  M[1][4] = 1 ;
  M[2][4] = 1 ;
  M[2][5] = 1 ;
  M[3][1] = 1 ;
  M[4][3] = 1 ;
  M[5][5] = 1 ;
  /* Afficher le graphe */
  Afficher_Graphe();
}
```

Solution 3 :

```
/* Procédure de transformation d'une liste d'adjacence en
matrice d'adjacence */
void Transformer()
{
  for (i=0; i<nbrs; i++){
```

```
      P = G[i];
      while (P != NULL){
             M[i][P->Sommet] = 1;
             P = P->Suivant;
        }
    }
}
```

Remarque : Pour tester cette procédure, on doit reprendre le programme de l'exercice 1, tout en supprimant la procédure d'affichage du graphe (liste d'adjacence) qui sera remplacée par la procédure d'affichage du graphe (matrice d'adjacence) du programme de l'exercice 2, sans oublier la déclaration de cette matrice (`int M[nbrs][nbrs]`). Maintenant, on peut ajouter la procédure de transformation avec la déclaration de deux variables globales (`int i` et `Succ *P`). Avant d'appeler la procédure `Transformer` et afficher la matrice d'adjacence, les éléments de la matrice doivent être initialisés à `0`.

Solution 4 :

```
/* Procédure d'affichage des éléments du graphe en largeur */
void Afficher_Larg()
{ int i, Imax, Imin ;
// Le tableau Visite indique qu'un nœud a été déjà visité ou non
// Le tableau TLarg va contenir le résultat du parcours en largeur
  int Visite[nbrs], TLarg[nbrs];
/* Initialisation */
  for (i=0; i<nbrs; i++){
    Visite[i] = 0;
    TLarg[i] = 0 ;
  }
  Imin = 0;
  Imax = 0;
  TLarg[0] = 0;
  Visite[0] = 1;
  /* Remplir le tableau TLarg */
  while (Imax < nbrs-1) {
    for (i=0; i<nbrs; i++)
      if ((M[TLarg[Imin]][i] == 1) && ! Visite[i]) {
        Imax++;
        TLarg[Imax] = i;
        Visite[i] = 1;
       }
    Imin++;
    if (Imin == Imax+1) {
      i = 0;
      while ((i<nbrs) && Visite[i]) i++;
      if (i<nbrs) {
        Imax++;
        TLarg[Imax] = i;
        Visite[i] = 1;
```

```
      }
  }
  }
  for (i=0; i<nbrs; i++) printf("%d ", TLarg[i]);
}
```

Solution 5 :

```
/* Le tableau Visite déclaré comme variable globale indique qu'un
nœud a été déjà visité ou non */
int Visite[nbrs];

/* Procédure récursive d'affichage des éléments d'un noeud en
profondeur */
void Afficher_Prof_R(int s)
{
       int i;
       if (! Visite[s]){
              Visite[s] = 1;
              printf("%d ", s);
              for (i=0; i<nbrs; i++) if(M[s][i]==1)
                  Afficher_Prof_R(i);
       }
}
/* Procédure d'affichage des éléments du graphe complet en
profondeur */
void Afficher_Prof()
{
  int i, b;
  /* Initialisation du tableau Visite */
  for(i=0; i<nbrs; i++) Visite[i] = 0;
  do
    {
    b = 0 ;
       for(i=0; i<nbrs; i++) if (! Visite[i]) {
              Afficher_Prof_R(i);
              b = 1;
        }
      }
    while (b) ;
}
```


Solution 6 :

```
#include<stdio.h>
#define nbrs 5
int M[nbrs][nbrs];
/* Procédure de création du graphe */
void Creation()
{ int i, j;
  char c;
for (i=0; i<nbrs; i++)
  for (j=0; j<nbrs; j++){
```

```
  M[i][j] = 0 ;
  printf("Existe-il un arc entre le sommet %d et le sommet %d o/n ? ", i, j) ;
  c = getchar(); getchar();
  if (c=='o') M[i][j] = 1;
  }
}
/* Procédure d'affichage des éléments du graphe */
void Afficher_Graphe()
{ int i, j;
  for (i=0; i<nbrs; i++){
    printf("Sommet : %d\n", i);
    for (j=0; j<nbrs; j++)
      if (M[i][j] == 1) printf("\tSuccesseur : %d\n", j);
    }
}
main() /* Programme principal */
{
  /* Créer le graphe */
  Creation();
  /* Afficher le graphe */
  Afficher_Graphe();
}
```


Solution 7 :

Déterminer l'existence d'un chemin entre deux nœuds revient à calculer la somme : $M^1 + M^2 + ... + M^{nbrs}$.

Avant d'implémenter la fonction désirée, nous allons déclarer une variable globale `H` comme suit :

```
int H[nbrs][nbrs];
```

La variable `H` va être utilisée pour stocker le résultat de la puissance de la matrice `M` par une valeur `n`.

Pour des raisons de lisibilité de programme, nous allons diviser le problème en une procédure qui détermine la puissance de `M` par une valeur `n`, ensuite nous développons la fonction qui détermine l'existence d'un lien entre deux sommets, et cela comme suit :

```
/* Procédure qui calcule la puissance de M par n et range le
résultat dans H */
void M_puissance(int n)
{   int i, j, p, k, s;
    int X[nbrs][nbrs];
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++) H[i][j] = M[i][j];
  for (p=2; p<=n; p++){
    for (i=0; i<nbrs; i++)
      for (j=0; j<nbrs; j++) {
        s = 0;
        for (k=0; k<nbrs; k++) s = s + H[i][k] * M[k][j];
```

```
        X[i][j] = s ;
      }
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++) H[i][j] = X[i][j];
  }
}
/* La fonction qui détermine si un lien existe entre deux
sommets ou non */
int Existe_Chemin(int s1, int s2)
{  int i,j,k, b;
   int X[nbrs][nbrs];
  b = 0;
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++) X[i][j] = M[i][j];
  for (k=2; k<=nbrs; k++) {
    M_puissance(k);
    for (i=0; i<nbrs; i++)
      for (j=0; j<nbrs; j++) X[i][j] = X[i][j] + H[i][j];
  }
  if (X[s1][s2] != 0) b = 1;
  return b;
}
```

Remarque : Maintenant, pour tester cette procédure dans le programme C de l'exercice 2 ou 6, il est possible de mettre, par exemple :

```
if (Existe_Chemin(1, 2)) printf ("Il y a un chemin entre les deux sommets.") ;
else printf("Pas de chemin entre les deux sommets.") ;
```


Solution 8 :

Pour cette solution, nous allons utiliser une variable globale `F` qui va retourner le résultat de la fermeture transitive. Cette variable est déclarée comme suit :

```
int F[nbrs][nbrs];
```

La procédure de la fermeture transitive est la suivante:

```
void Fermeture()
{ int i, j;
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++)
      if (Existe_Chemin(i,j)) F[i][j] = 1;
        else F[i][j] = 0 ;
}
```

Remarque : Après avoir invoqué cette procédure dans le programme C de l'exercice 2 ou 6, on affiche la matrice `F` comme suit :

```
int x,y;
for(x=0; x<nbrs; x++)
for(y=0; y<nbrs; y++) printf("F[%d,%d]=%d\n", x,y,F[x][y]);
```

Solution 9 :

```
/* Fonction qui détermine si le graphe M(nbrs*nbrs) est
fortement connexe ou non */
int Graphe_For_Con()
```

```
{ int i, j, b;
  b = 1 ;
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++)
      if (!(i==j))
        if (!(Existe_Chemin(i,j) && Existe_Chemin(j,i))) b = 0;
  return b;
}
```

Remarque : Après avoir invoqué cette procédure dans le programme C de l'exercice 2 ou 6, on teste si le graphe est connexe ou non comme suit :

```
if (Graphe_For_Con()) printf("Le graphe est fortement connexe.");
else printf("Le graphe n'est pas fortement connexe.");
```

Solution 10 :

```
/* Procédure qui affiche les composantes fortement connexes
d'un graphe M(nbrs*nbrs) */
void Composantes_For_Con()
{   int i, j , k;
    int cfc[nbrs];
  Fermeture();
  for (i=0; i<nbrs; i++)
    for (j=0; j<nbrs; j++) F[i][j] = F[i][j] * F[j][i];
  for (i=0; i<nbrs; i++) F[i][i] = 1;
  /* Affichage des composantes fortement connexes */
  for (i=0; i<nbrs; i++) cfc[i] = 0;
  k = 0;
  for (i=0; i<nbrs; i++)
    if (!cfc[i]) {
     k++;
     printf("Composante fortement connexe num %d contient les sommets :\n", k);
     for (j=0; j<nbrs; j++) if (F[i][j] == 1) {
           printf("%d  ",j);
           cfc[j] = 1;
     }
     printf("\n");
    }
}
```

Références bibliographiques additionnelles

- Djelloul BOUCHIHA. *Algorithmique et programmation en Pascal : Cours avec 190 exercices corrigés*. Editions Universitaires Européennes, Copyright © 2020 International Book Market Service Ltd., Member of OmniScriptum Publishing Group. ISBN : 978-613-9-55434-8. 420 p.
- Djelloul BOUCHIHA. *Programmation Orientée Objet en Java*. Edition errachad, Sidi Bel Abbes, Algérie. Semestre 2/2016. ISBN : 978-9947-65-055-4. 82 p.
- Djelloul BOUCHIHA. *Initiation à l'Algorithmique et à la Programmation en Pascal*. Edition errachad, Sidi Bel Abbes, Algérie. Semestre 2/2019. ISBN : 978-9947-65-075-2. 102 p.
- Djelloul BOUCHIHA. *Notions Avancées sur l'Algorithmique et la Programmation en Pascal*. Edition errachad, Sidi Bel Abbes, Algérie. Semestre 2/2019. ISBN : 978-9947-65-074-5. 90 p.
- Damien BERTHET, Vincent LABATUT. *Algorithmique & programmation en langage C - vol.1 : Supports de cours*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.232. <cel-01176119>
- Damien BERTHET, Vincent LABATUT. *Algorithmique & programmation en langage C - vol.2 : Sujets de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.258. <cel-01176120>
- Damien BERTHET, Vincent LABATUT. *Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.217. <cel-01176121>
- Claude DELANNOY.*Apprendre à programmer en Turbo C*. Editions CHIHAB – EYROLLES -1994. 394 p.
- Claude DELANNOY. *Programmer en langage C : cours et exercices corrigés*. 5e édition 2009. ÉDITIONS EYROLLES. ISBN : 978-2-212-12546-7. 267 p.
- Rémy MALGOUYRES, Rita ZROUR et Fabien FESCHET. *Initiation à l'algorithmique et à la programmation en C: Cours avec 129 exercices corrigés*. Editeur : Dunod, 3e édition (18 juin 2014), ISBN : 978-2100710010. 336 p.

INDEX

'

' (guillemet simple) · 19, 26, 31

!

! (non logique) · 34
!= (différent) · 33

"

" (guillemet double) · 18, 27, 103

#

#define · 24, 25, 36
#include · 18, 31

%

% (code de format) · 19, 29, 30, 39, 103
% (modulo) · 25, 32

&

& (adresse) · 20, 104, 138, 190, 207
& (et bit-à-bit) · 34
&& (et logique) · 34

(

(…) (parenthèses) · 11, 33, 59, 129, 130, 134

*

* (multiplication) · 11, 25, 26, 32
* (pointeur) · 138, 205

,

, (virgule) · 16, 129

.

. (accès au champ) · 209

/

/ (division) · 11, 25, 26, 32
/*…*/ (commentaire) · 27, 36
// (commentaire) · 36

;

; (point-virgule) · 16, 27, 59, 162, 166

\

\ (caractère spécial) · 18, 185

^

^ (ou exclusif bit-à-bit) · 34

{

{ … } (accolades) · 18, 59, 74

|

| (ou bit-à-bit) · 34
| | (ou logique) · 34

~

~ (complément à un) · 34

+

+ (addition) · 11, 16, 25, 26, 32
+ (identité) · 32, 40
++ (incrémentation) · 40

<

< (inférieur) · 11, 33, 206
<< (décalage à gauche) · 35
<= (inférieur ou égal) · 11, 33, 206
<> (différent) · 11, 206

=

= (affectation) · 18, 26, 33, 39
= (égal) · 11, 206
== (égal) · 18, 33, 39

>

> (supérieur) · 11, 33, 206
>= (supérieur ou égal) · 11, 33, 206
>> (décalage à droite) · 35

←

← (affectation) · 16, 26

–

– (opposé) · 32
– (soustraction) · 11, 25, 26, 32
– – (décrémentation) · 41
–> (accès au champ) · 209

A

`abs` · 132
accès direct · 186
accès formaté · 186
accès non-formaté · 186
accès par bloc · 186, 189

accès séquentiel · 186
Ackermann · 148
adresse d'une variable · 25
adresse mémoire · 24, 138, 164, 205
affectation · 26, 162, 165, 206
algorithme · 15
`Algorithme` · 16
algorithmique · 15
`aller à …` · 62
allocation dynamique · 139, 205, 206
`Allouer` · 206
année bissextile · 64
arbre · 254, 269
arbre binaire · 255
arbre binaire complet · 258
arbre binaire de recherche · 259
arbre binaire dégénéré · 259
arbre binaire équilibré · 261
arbre binaire ordonné · 259
arbre des appels · 143
arbre n-aire · 255
arc · 254, 269
`arctg` · 11
arête · 254
arguments · 128
arité de l'arbre · 255
ASCII · 12, 30
assembleur · 21
assignation · 26
automatique · 10

B

bibliothèque · 132
BIOS · 12
bit · 12
bloc · 59, 126, 129, 135, 161
booléen · 11, 32, 33
`Booléen` · 26
boucle infinie · 73, 74, 77
boucle Pour · 75
boucle Répéter · 74
boucle Tant que · 73
boucles · 73
boucles imbriquées · 77
branche · 254
`break` · 62, 77
buffer · 183, 186
byte · 12

C

caractère · 12, 30
`Caractère` · 26
caractères spéciaux · 18
carte mère · 10
cas trivial · 216
case mémoire · 25, 206
cast · 37
CD-ROM · 12, 13
cellule · 208
chaîne de caractères · 27, 103
chaîne miroir · 148
champs · 161, 164, 184, 209
`char` · 30, 31
chemin · 269
clavier · 12
clé USB · 12, 13, 183
code · 24
code binaire · 21
code de format · 19
code machine · 21
code source · 21
commentaire · 36

compilateur · 21, 35, 130
complément à un · 35
complexité spatiale · 30
complexité temporelle · 30
composantes fortement connexes · 271
compteur · 76
condition · 58, 75
`const` · 25, 36
constante littérale · 36
constante symbolique · 36
constantes · 24, 166, 167
constantes textuelles · 36
constates numériques · 36
`continue` · 77
conversion · 37
conversion explicite · 37
conversion implicite · 37
corps de l'algorithme · 24, 27
corps de la procédure · 129
`cos` · 39, 132
CPU · 21
critère d'arrêt · 143
`ctype.h` · 111
cycle · 269
cylindre · 12

D

débordement de pile · 144
débordement de tableau · 93
`Début` · 16, 27
décalage à droite · 35
décalage à gauche · 35
déclaration · 24, 25, 41
décrémentation · 74, 76, 206
défilement · 223
dégradation de l'information · 37
dépassement de capacité · 35
dépilement · 221
déroulement d'un algorithme · 42, 47
désallocation · 205
`Désallouer` · 206
dichotomie · 97
disque dur · 12, 13, 183
`do … while…` · 75
données · 24, 128
`double` · 29
duplicate case value · 62
DVD · 13

E

écran · 12
`Ecrire` · 16, 27
effet de bord · 129
empilement · 221
`Empiler` · 221
emplacement de mémoire · 24
enfilement · 223
ENIAC · 10
enregistrement · 208
en-tête · 129, 130
`Entier` · 25
entrées-sorties · 27, 39
`enum …` · 166
énumération · 166
énumération des étapes · 16, 27
`EOF` · 131, 185
équation du second degré · 64
équations logiques · 11
erreur d'exécution · 21, 33, 208
erreur de compilation · 21, 33, 40, 56, 62, 103, 134
Error · 21

espace mémoire · 25, 35, 144, 205
ET · 11, 26, 34
et bit-à-bit · 35
expressions arithmétiques · 11, 26
expressions logiques · 11, 26, 58, 73
extern · 137

F

factorielle · 78, 142, 145
FAUX · 11, 26, 32, 33
fclose · 185
feof · 187
fermeture transitive · 271
fgetc · 186
fgets · 187
Fibonacci · 79, 108, 147
fichier · 15, 183
fichier d'en-tête · 132
fichiers binaires · 186, 189
fichiers textes · 186, 189
FIFO · 222
file · 222, 223
FILE · 184, 223
fils droit · 255
fils gauche · 255
Fin · 16, 27
Flag · 97
float · 29
fonction · 130, 136, 168
fonction variadique · 142
fonctions de bas niveau · 183
fonctions de haut niveau · 183
fonctions mathématiques · 11, 39
fopen · 184
for … · 76
formatage · 12
forme graphique · 255
forme parenthèsée · 255
fprintf · 188
fputc · 187
fputs · 188
frames · 139
fread · 189
free · 206
fscanf · 188
fseek · 190
ftell · 191
fuite de mémoire · 208
fwrite · 189

G

gestion dynamique · 206
gestion statique · 206
getchar · 31
gets · 105
goto … · 63
graphe · 269
graphe acyclique · 269
graphe connexe · 269
graphe fortement connexe · 269
graphe orienté · 269
graphe pondéré · 269

H

HARDWARE · 10
hauteur d'un arbre · 254
header file · 132
heap · 208

I

identificateur · 24, 27, 35, 103, 166
`if …` · 59
`if … else …` · 60
incrémentation · 74, 76, 206
indentation · 36
indice · 74, 94, 96, 104
information · 10
informatique · 10
initialisation · 19, 96, 163
instruction expression · 41
instructions · 20, 59, 128
`int` · 18, 29
itération · 75
itérative · 142

L

langage de description d'algorithme · 15
langage de programmation · 16, 17
langage machine · 21
language C · 17
LDA · 15
lien · 254, 269
LIFO · 220
`Lire` · 16, 27
lisibilité du code source · 36
liste chaînée · 205, 208, 255
liste circulaire · 231
liste d'adjacence · 269
liste doublement chaînée · 217
`ln` · 11
`long` · 29
`long double` · 29
lvalue · 36, 93

M

maillon · 208
`main` · 17, 134, 138
`malloc` · 206
mantisse · 29
`math.h` · 39, 132
matrice · 102
matrice d'adjacence · 270
medium · 139
mémoire · 11
mémoire auxiliaire · 12
mémoire centrale · 11
menu · 64
mode caractère · 186
mode chaîne · 186
module · 128
mot des feuilles · 254
mots clés · 36, 131
mots réservés · 36

N

niveau · 254
nœud · 208, 254, 269
nœud feuille · 254
nœud fils · 254
nœud frère · 254
nœud père · 254
nœud racine · 254
nom · 24, 35, 129, 130, 166
nombre d'apparition · 109, 110
nombre premier · 108, 144
nombres consécutifs · 108

NON · 11, 26
notation scientifique · 29
NULL · 184, 208

O

octet · 12
opérandes · 11
opérateur conditionnel · 41
opérateur d'adressage · 207
opérateur de référençage · 207
opérateurs arithmétiques · 11, 25, 26, 32
opérateurs binaires · 32
opérateurs bit-à-bit · 34
opérateurs d'affectation élargie · 41
opérateurs de comparaison · 11, 25, 26, 33, 206
opérateurs logiques · 11, 33
opérateurs unaires · 32
opération arithmétique · 20
opération de départ · 75
opération logique · 20
opérations · 20, 24
opérations de base · 26
ordinateur · 10
ordre de force · 37
ordre de priorité · 11, 33, 38
organigramme · 28
OU · 11, 26, 34
ou bit-à-bit · 35
ou exclusif · 35

P

paramètres · 128, 129, 139
paramètres effectifs · 129, 130, 138
paramètres formels · 129, 130, 138
parcours en largeur · 271
parcours en profondeur · 257, 271
parcours infixé · 258
parcours postfixé · 258
parcours préfixé · 257
pas de progression · 75
passage des paramètres · 137
passage par adresse · 138, 141, 164
passage par référence · 138
passage par valeur · 138, 164
passage par variable · 138
PC · 10
périphériques · 12
permutation · 44, 53, 146
perte de précision · 37
PGCD · 79, 148
pile · 139, 144, 146, 220
piste · 12
poids · 269
point d'appel · 126, 130
point décimal · 36
pointeur · 95, 138, 184, 205, 208
portée d'une variable · 41, 135
Pour … · 75
pow · 39, 132
printf · 18, 27, 103, 131, 139
procédure · 129, 136, 163
procédure appelante · 134
procédure appelée · 134
processeur · 11
produit matriciel · 109
profondeur · 254
programmation modulaire · 128
programme · 17, 20
programme appelant · 130
programme principal · 17

prototype · 134
pseudo-code · 15
putchar · 31
puts · 18, 103

R

RAM · 11, 12, 13
Recherche dans un tableau · 97
recherche dichotomique · 97
récursion · 142
récursivité · 142
Réel · 25
règle de trois · 13
répertoire · 15
Répéter … · 74
return · 130
ROM · 12

S

scanf · 18, 27, 104, 131, 139
secteur · 12
segment de données · 139
Selon … · 60
sens d'associativité · 33, 39
séquence · 58, 73
short · 29
Si … · 58
Si … Sinon … · 59
signed · 29
sin · 11, 39, 132
size_t · 189
sizeof · 35, 96, 190, 206
small · 139
SOFTWARE · 10
sommet · 254, 269
sous-programme · 126, 128
sous-programme appelant · 134
sprintf · 106
sqrt · 39, 132
sscanf · 106
stack · 139, 221
stack overflow · 144
static · 137
stdarg.h · 142
stdio.h · 18, 31, 184, 189
stdlib.h · 132, 207, 208
strcat · 105
strcmp · 105
strcpy · 104
stricmp · 106
string.h · 104
strlen · 104
strncat · 105
strncmp · 106
strncpy · 104
strnicmp · 106
struct … · 162
structure · 161, 208
structure conditionnelle composée · 59
structure conditionnelle multiple · 60
structure conditionnelle simple · 58
structures conditionnelles · 58
structures de contrôle · 26
structures récursives · 208, 254
switch … · 61
système d'exploitation · 14

T

tableau · 93, 140, 205
tableau à deux dimensions · 102

tableau de chaînes de caractères · 107
tableau de situation · 42, 47
taille d'un arbre · 254, 258
tampon · 183, 185
`tan` · 39, 132
`Tant que …` · 73
tas · 139, 208
technique de Flag · 97
tiny · 139
`tolower` · 111
`toupper` · 111
traitement · 10, 24
transposée d'une matrice · 109
transtypage · 37
tri à bulles · 101
tri d'un tableau · 100
tri par sélection · 100
troncature · 39
type · 24, 25, 129, 130
type logique · 11
type numérique · 11, 25
type symbolique · 26
`typedef` · 161
types personnalisés · 161
types simples · 25
types structurés · 26, 93

U

UAL · 11
UC · 11
union · 164
`union …` · 165
unité centrale · 10
`unsigned` · 29

V

`va_arg` · 142
`va_list` · 142
`va_start` · 142
valeur · 24
valeur absolue · 132
valeur finale · 76
valeur initiale · 19, 76
variable de contrôle · 75
variable locale statique · 137
variable pointée · 206
variables · 24
`Variables` · 16
variables globales · 135, 136, 137, 139
variables locales · 135, 138, 139
vecteur · 102
virgule flottante · 29
`void` · 129
`VRAI` · 11, 26, 32, 33

W

Warning · 21, 134
`while …` · 73

X

XOR · 35