République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Abderahmane Mira de Béjaïa

Faculté des Sciences Exactes

Département d'Informatique

Support de Cours

Pour la première année LMD en Mathématiques et Informatique (MI).

Module :

Algorithmique et Structures de Données 1

Préparé par

Dr. Samra BOULFEKHAR

Maître de conférences, Catégorie A au Département Recherche Opérationnelle, Faculté des Sciences Exactes, Université Abderahmane Mira de Béjaïa.

Année 2021 – 2022

Table des matières

Table des matières

Avant-propos

Ce polycopié est le support écrit du cours " Algorithmique et Structures de Données 1" de la première année Mathématiques et Informatique (MI), Faculté des Sciences Exactes de l'Université A/Mira de Béjaïa, 2021-2022. Il constitue un manuel de cours et d'exercices sur une partie du domaine de programmation permettant la conception et la réalisation d'un petit logiciel, en utilisant des objets élémentaires et des données structurées statiques.
Ce cours se donne pour faire évoluer les étudiant-e-s d'une position d'utilisateur de l'informatique à une position de concepteur, en acquérant les premières bases en algorithmique. Il est pour objectif la maîtrise des compétences suivantes :

- **comprendre** et **examiner** un algorithme préexistant, son fonctionnement ou son but ;
- **modifier** un algorithme pour obtenir un résultat précis ;
- **analyser une situation** : identifier les données d'entrée et de sortie, le traitement, les instructions, etc ;
- **créer une solution algorithmique à un problème donné** : comment écrire un algorithme en " langage courant " en respectant un code, identifier les opérations d'écriture, d'affichage, les boucles, les tests, les tableaux (si on en a besoin), etc ;
- **valider** la solution algorithmique par des traces d'exécution (le déroulement d'un algorithme) ;
- **développer** un programme en Langage C.

Ce support de cours "Algorithmique et Structures de Données" s'intéresse, comme on a dit auparavant, à la présentation des différents éléments principaux et nécessaires pour construire un algorithme et le programmer ainsi qu'à la présentations de deux structures de données statiques, qui sont les tableaux à une seule dimension et les tableaux à deux dimensions .
Il s'articulera autour de cinq chapitres.
Le chapitre I est un chapitre introductif, qui aborde quelques concepts généraux relatifs à l'informatique, à l'algorithmique et aux langages de programmations (pas forcément le langage C).
Le chapitre 2 décrit les différents types d'expression qu'on peut rencontrer dans les algorithmes et la façon de les évaluer, ainsi que les notions de variable, de constante et de type, et les opérations appliquées sur chaque type. Il décrit aussi les différentes instructions de base dans un algorithme, à savoir l'opération d'écriture, d'affectation et d'affichage. Comme il décrit les

symboles nécessaires pour établir une solution graphiques (l'organigramme).
Le chapitre 3 présente les différentes structures conditionnelles disponibles en algorithmique, à savoir, la conditionnelle simple, la conditionnelle complète et la conditionnelle à choix multiples.
Le chapitre 4 décrit en détails les différentes structures itératives (boucles) qui peuvent être utilisées dans un algorithme, à savoir : la boucle Pour, la boucle Tantque et la boucle Repeter.
Le chapitre 5 est dédié aux principales structures de données statiques (tableaux et matrices) ainsi qu'aux chaînes de caractères.
Enfin, ce support de cour se clôturera par une annexe sur les principaux composants du langage C et une liste de références.

Chapitre 1

Introduction à l'informatique

Sommaire

Objectif du chapitre

L'objectif principal de ce chapitre consiste à faire apprendre à l'étudiant :

- Des notions importantes liées aux termes : informatique et algorithmique,
- Les composants de base d'un ordinateur et le rôle principal de chaque composant,
- Les différentes étapes à suivre pour résoudre un problème par un ordinateur.

1.1 Bref historique sur l'informatique

L'informatique est une science qui permet de traiter l'information de façon automatique. Ce terme a été proposé par Philippe Dreyfus en 1962, il résulte de la combinaison des trois premières syllabes du terme "Information" et des deux dernières syllabes du terme "automatique". L'informatique a pour rôle :

- La conception et la construction des ordinateurs,
- Le fonctionnement et la maintenance des ordinateurs,
- L'utilisation des ordinateurs dans les différents domaines d'activités (leur exploitation).

1.1.1 Evolution de l'informatique et des ordinateurs

- En 1643, BLAISE PASCAL a inventé la "PASCALINE", qui est une machine mécanique capable de réaliser des additions et des soustractions.
- En 1812, CHARLES BABBAGE a conçu une machine mécanique pouvant effectuer des calculs numériques compliqués (c'est une machine à base de cartes perforées). En 1860, il définit les grands principes des calculateurs électroniques.
- En 1885, HERMANN HOLLERITH (l'inventeur des cartes perforées) a construit la première machine à cartes perforées et qui a été servie dans l'opération de recensement de la population d'Amérique en 1890.
- En 1946, le premier calculateur électronique est apparu. Il est baptisé "ENIAC" (Electronic Numerical Integrator and Computer). Il est construit sur le principe du binaire 0 et 1 (le courant passe ou ne passe pas). C'est une machine électronique capable de réaliser 5000 additions/seconde et 3000 multiplications/seconde. C'est le premier ordinateur qui utilise le principe de "Programme enregistré" et constitue **la première génération des ordinateurs (1946 - 1957)**. Parmi les ordinateurs inventés à l'époque, nous citons :
  - ENIAC (1946),
  - IBM 701 (1952).
- **Deuxième Génération d'ordinateurs (1958 - 1963) :**
  Ils sont apparus après l'invention du "Transistor" composant électronique capable de réguler le courant. Ces ordinateurs sont 100 fois plus rapides que ceux de la 1^{re} génération, ils consomment moins d'énergie électrique et ils sont moins volumineux (occupent moins d'espaces).
  - **Exemples**
    - PDP I (1960),
    - DEC PDP8 (1964).
- **Troisième Génération d'ordinateurs (1964-1971) :**
  Après l'apparition du "Circuit Intégré", ce type d'ordinateurs utilisent les transistors et

les circuits intégrés.

- **Exemples**
  - IBM 360 (1964),
  - IBM 7030 (1961).

- Quatrième Génération d'ordinateurs :

Les ordinateurs ou tout simplement les micro-ordinateurs sont construits autour de microprocesseur, ils constituent les ordinateurs de la quatrième génération.

- **Exemples de microordinateurs**
  - MICRAL 8008 (1973),
  - ALTAIR 8008 (1973),
  - APPLE1 (1976),
  - IBM PC (1981),
  - PENTUIM, etc.

Actuellement l'évolution des ordinateurs tend à exploiter le laser, les fibres optiques, la biochimie ou ordinateur moléculaire.

1.1.2 Définitions

Dans cette sous-section, nous définissons quelques termes importants liés au terme informatique.

- **Système Informatique**

  Un système informatique est un ensemble de moyens informatiques et de télécommunications, *matériels* et *logiciels*, ayant pour finalité de collecter, traiter, stocker, acheminer et présenter des données aux utilisateurs.

  - **Remarque**

    Tous les termes écris en *Italique* seront décrits en détail dans ce qui suit.

- **Matériels Informatiques**

  Il s'agit des ressources matérielles qui sont exploitées pour exécuter les différents logiciels destinés à satisfaire les besoins des utilisateurs : ordinateurs, imprimantes, scanners, etc. Les concepts de système informatique et *d'ordinateur* ne doivent pas être confondus. L'ordinateur n'est que l'un des composants, certes central, des systèmes informatiques. Il en existe beaucoup d'autres, parmi lesquels on peut citer les matériels réseau, les capteurs et actionneur, etc.

- **Ordinateur**

  Un ordinateur est une machine de traitement automatique des données selon un *programme* enregistré en *mémoire*. Il comporte un ou des *processeurs*, une mémoire et des

périphériques d'entrée, de sortie et de stockage. Les programmes sont nécessairement exprimés dans le *langage machine* propre au processeur qui les exécute.

- **Logiciel**
  Un logiciel est un ensemble de programmes, conçu par un développeur de logiciels, répondant aux besoins fréquents de *l'utilisateur* dans un domaine d'activités bien déterminé. On distingue plusieurs types de logiciels tels que les logiciels de base comme le système d'exploitation ou les programmes d'application (Traitement de texte, comptabilité, etc).
- **Développeur de logiciels**
  Un développeur (également connu sous le nom de programmeur, codeur ou ingénieur logiciel) est un professionnel de l'informatique qui utilise des *langages de programmation* pour créer des logiciels informatiques.
- **Utilisateur**
  En informatique, le terme utilisateur (anciennement un opérateur ou un informaticien) est employé pour désigner une personne qui utilise un système informatisé (ordinateur) mais qui n'est pas nécessairement informaticien (par opposition au développeur par exemple).

1.1.3 Architecture de base d'un ordinateur

Dans cette partie, nous décrivons rapidement l'architecture de base d'un ordinateur et les principes de son fonctionnement.
L'architecture, dite architecture de Von Neumann, décompose l'ordinateur en quatre parties distinctes (voir la figure 1.1) :

- **Le processeur**,
- **La mémoire centrale**,
- **Les dispositifs / périphériques d'entrée-sortie**,
- Les différents composants sont reliés par **des bus**.

1. **Le processeur** : il est grossièrement divisé en deux parties, l'unité de contrôle et l'unité de traitement.
   - **L'unité de contrôle** est responsable de la lecture en mémoire principale et du décodage des instructions, de la gestion des différents constituants de l'ordinateur (contrôler les échanges, gérer l'enchaînement des différentes instructions, etc).
   - **L'unité de traitement**, aussi appelée l'Unité Arithmétique et Logique (UAL), exécute toutes les opérations arithmétiques et logiques (addition, soustraction, multiplication, division, comparaison, etc.) qui manipulent les données.

   Ces deux unités communiquent avec la mémoire principale, la première pour lire les instructions, la seconde pour recevoir/transmettre des *données binaires (bits)*, mais ils communiquent également avec les différents périphériques (clavier, souris, écran, etc).

2. **La mémoire centrale** : elle contient à la fois les données et le programme exécuté par l'unité de contrôle. La mémoire centrale se divise en :
   - **Mémoire volatile ou RAM** (Read Access Memory) qui contient les programmes et les données en cours de traitement.
   - **Mémoire permanente ROM** (Read Only Memory) qui stocke les programmes et les données de base de la machine.

   La mémoire centrale se mesure actuellement par milliers de mégaoctets, tel que :
   - 1 bit : (état électronique 1 ou 0) = unité élémentaire de l'information.
   - 1 octet = 1 caractère = 8 bits = 2^3 bits.
   - 1 Kilooctets = 2^{10} octets =10^{24} bits.
   - 1 Megaoctets = 2^{10} Kilooctets = 2^{20} octets.
   - l Gigaoctets = 2^{10} Megaoctets = 2^{20} Kilooctets = 2^{30} octets.
3. **Les dispositifs / périphérique d'entrée-sortie** : on appelle "périphérique" tout matériel électronique pouvant être raccordé à un ordinateur. Voici quelques exemples :
   - **L'écran** : permet d'afficher le contenu de l'ordinateur.
   - **Le clavier et la souris** : sont indispensables pour entrer en communication avec l'ordinateur.
   - **Les haut-parleurs** : permettent d'écouter les fichiers sons. Certains écrans disposent de haut-parleurs intégrés.
   - **La webcam** : est une petite caméra à poser sur un ordinateur qui permet de filmer des images et de les transférer sur l'ordinateur. Elle peut également être intégrée dans l'écran de l'ordinateur.
   - **L'imprimante** : permet d'imprimer sur papier des fichiers. Certaines imprimantes ont également une fonction "scanner" qui sert à convertir une page "papier" en fichier lisible par l'ordinateur.
   - **Le modem** : est un petit boîtier qui permet de se connecter à Internet.
   - **Le disque dur externe** : est un disque dur embarqué dans un boîtier solide et qui se connecte à un ordinateur en USB. Il peut être utilisé pour sauvegarder des données de votre ordinateur ou enregistrer des fichiers plus lourds (films) et pouvoir les transporter.
   - **La clé USB et la carte mémoire** : permettent de stocker des données sur un petit support transportable. Les cartes mémoires sont généralement destinées à être placées dans des appareils photo, caméras, smartphone, etc.

   Les périphériques **d'entrée-sortie** sont souvent classés en 2 catégories :
   - **Périphériques d'entrée** : servent à fournir des informations (ou des données) au système informatique. Exemples : clavier, souris, scanner, webcam, microphone, etc.
   - **Périphériques de sortie** : servent à faire sortir des informations du système informatique. Exemples : écran, imprimante, casque, etc.
   - **Remarque**
     - Il existe également des périphériques d'entrée et de sortie, ils permettent de fournir

des informations, mais également sortir des informations de l'ordinateur. Exemples : clé USB, carte mémoire, modem, etc.

- Lesmémoires externes ou auxiliaires (Disques durs, disquettes, flashdisque, CD, DVD, etc.) sont appelées, aussi, les périphériques de stockage ou les périphériques d'entrée et de sortie en même temps.

4. **Les Bus** : les informations échangées entre la mémoire et le processeur circulent sur des bus. Un bus est simplement un ensemble de **n** fils conducteurs, utilisés pour transporter **n** signaux binaires.

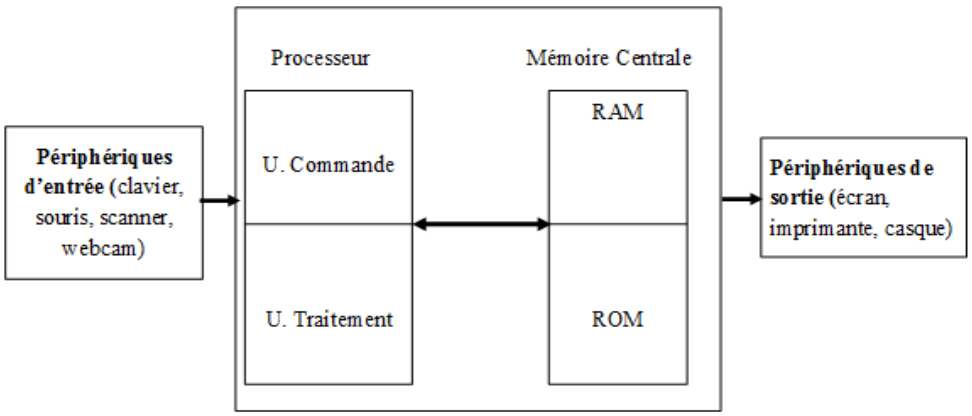


FIGURE 1.1 – Architecture de base d'un ordinateur.

1.2 Introduction à l'algorithmique

Le terme *Algorithme* vient du nom du mathématicien Abou Jaafar Mohamed Ibn Moussa Al Khawarizmi. Un Algorithme est une suite ordonnée d'instructions qui permettent la résolution d'un problème. En effet, La résolution automatique d'un problème s'effectue en plusieurs étapes (voir la figure 1.2) :

FIGURE 1.2 – Etapes de résolution automatique d'un problème.

- **L'étape d'Analyse** : cette étape permet de comprendre le problème en répondant aux 3 questions suivantes :
  1. Quelles sont les entrées (les données) ?
  2. Quelles sont les traitements ou opérations qui permettent d'aboutir aux résultats recherchés à partir des données ?
  3. Quelles sont les sorties (les résultats voulus) ?

  Après la réponse aux questions, on passe à :
- **L'étape de l'écriture de l'Algorithme** : comprend trois parties :
  4. Lecture des données,
  5. Les opérations (traitements),
  6. L'écriture des sorties (résultats).

- **L'étape de Programmation**
  7. Traduire l'algorithme dans un langage de programmation.
- **L'étape d'Edition**
  8. On introduit le programme dans l'ordinateur (saisie au clavier) pour voir les résultats de notre travail.

1.2.1 Définitions

Dans cette sous-section, nous allons définir tous les nouveaux concepts introduits dans ce chapitre et qui ne sont pas encore définis :

- **Instruction**
  Une instruction est une action agissant sur un ou plusieurs objets (données). Chaque ligne dans un algorithme / programme représente une instruction.
- **Programme**
  Un programme est le résultat de la traduction d'un algorithme en utilisant un langage de programmation que la machine peut comprendre. C'est-à-dire, les programmes sont écrits avec un langage respectant des règles bien précises avec un vocabulaire bien défini qui sont traduits ensuite par un compilateur en langage binaire pour être exécuter par l'ordinateur.
- **Compilation**
  La compilation permet de traduire un programme écrit en langage de programmation dit "Langage évolué" en une suite d'instructions exécutables par la machine. un langage évolué est très proche du langage humain.
- **Langage de programmation**
  Un langage de programmation est un ensemble de mots (vocabulaires) et syntaxe (grammaire) stricte (nom ambigüe) permettant d'écrire un programme exécutable par une machine. Parmi les langages de programmation, nous citons : PASCAL, JAVA, C,

C++, MATLAB, etc.

- **Langage binaire**
  Le langage binaire (ou système binaire) est le langage de la machine. Il permet de traiter l'information représentée sous sa forme binaire, il travaille donc sur des données codées de façon *binaire*. Chaque variable binaire peut être représentée par deux états 0 et 1.
- **Langage Algorithmique**
  Un algorithme est la formalisation de la solution d'un problème par l'être humain et il n'est pas destiné pour être interprété par la machine. Les algorithmes sont écrits avec un langage très proche des langages humains, qu'on appelle *langage Algorithmique* (qui sera détaillé dans le chapitre suivant). Ce dernier utilise un vocabulaire et respecte des règles d'écriture universelles.

1.2.2 Caractéristiques d'un algorithme

Pendant la conception d'un algorithme, nous devons prendre en considérations les caractéristiques suivantes :

- Un algorithme doit être **Clair** : l'algorithme ne doit pas présenter des ambiguïtés (instruction interprétable de plusieurs manières) et facile à lire et à comprendre.
- Un algorithme doit être **Correct** : Il faut que l'algorithme exécute correctement les tâches pour lesquelles il a été conçu.
- Un algorithme doit être **Complet** : Il faut que l'algorithme considère tous les cas possibles et donne un résultat dans chaque cas.
- Un algorithme doit être **Fini** : l'algorithme doit se terminer quelle que soit la machine ainsi que le temps et la date d'exécution.
- Un algorithme doit être **Efficace** : l'algorithme doit effectuer le travail demandé avec l'utilisation du minimum de ressources (l'efficatité).

Chapitre 2

Algorithme séquentiel simple

Sommaire

Objectif du chapitre

A l'issue du cours, l'étudiant-e doit pouvoir construire un algorithme simple, le programmer et comprendre comment se déroule son exécution.

2.1 Structure générale d'un algorithme

Un algorithme est une représentation formelle de la solution d'un problème en utilisant le vocabulaire du langage 'algorithmique'. Les mots du vocabulaire algorithmique sont appelés '**mots clés**'.
Un algorithme est composé de trois parties (voir la figure 2.1) :

- Partie Entête,
- Partie Déclaration,
- Partie Action.

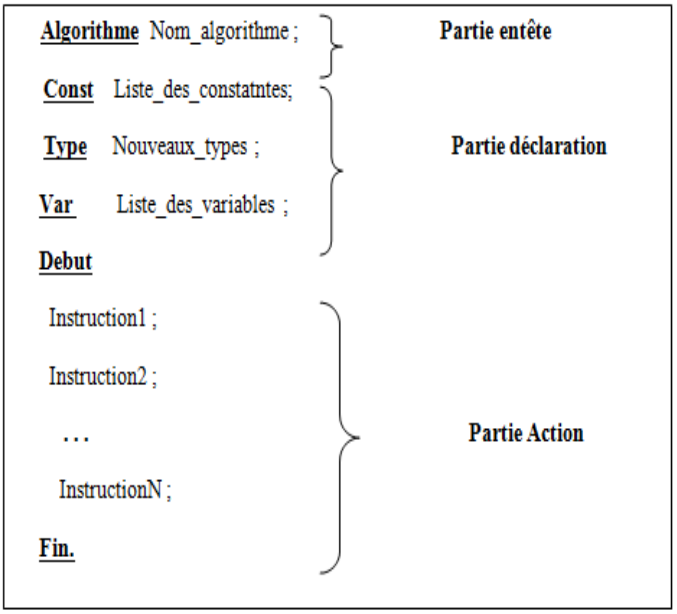


FIGURE 2.1 – Parties d'un algorithme.

2.1.1 Partie Entête

La partie "Entête" permet de donner un nom à l'algorithme à travers **un identificateur** de caractères composé uniquement des caractères alphanumériques (alphabétiques non accentuées (A..Z, a..z) et numériques (0-9)) et du caractère tiret 8 (_). Il doit respecter les règles d'écriture ci-dessous :

- Ne doit pas commencer par un chiffre,
- Ne doit pas être un mot clés,
- Ne contient pas les caractères spéciaux, ni le caractère espace.

- **Remarque**

  - Pour faire la différence entre les mots clés et les identificateurs, nous soulignons les mots clés et nous les écrivons en gras.
  - Un point-virgule (;) doit être mis après chaque déclaration et chaque instruction.
  - L'algorithme doit se terminer par un point (.).

- **Exemple 1**
  Algorithme mon_premier_algo ;

- **Exemple 2**
  Algorithme calcul ;

2.1.2 Partie Déclaration

La partie "Déclaration" contient tous les objets nécessaires dont la machine aura besoin pour résoudre le problème (*plus de détail sur ces objets sera donné dans la Section 2.2*).

2.1.3 Partie Actions

La partie "*Actions*" contient les instructions à exécuter pour solutionner le problème. Cette partie est délimitée par les mots clés '*Debut*' et '*Fin*'.

2.2 Données : constantes et variables

Les objets de base (simples) manipulés dans un algorithme sont :
- Les constantes,
- Les variables.

2.2.1 Constantes

Une constante est un objet dont l'état *reste inchangé* durant toute l'exécution d'un programme. On ne peut jamais modifier sa valeur et celle-ci doit donc être précisée lors de la

définition de l'objet.
Elle est caractérisée par deux caractéristiques :

- Un identificateur fixe (le nom de la constante),
- Une valeur fixe (le contenu de la constante).

- **Déclaration**
  Toute constante doit être déclarée avant qu'elle soit utilisée. Pour déclarer une constante, nous précédons l'identificateur de la constante à déclarer par le mot clé **Const**, puis, nous donnons la valeur souhaitée à cette constante.
  Const Id_const = valeur_cont ;

- **Exemple**
  Algorithme declar_constant ;
  Const P = 3,14 ; ch= "algo" ;

2.2.2 Variables

Une variable est un objet dont la valeur peut être modifiée tout au long de l'exécution du programme. Elle est caractérisée par trois caractéristiques :

- Un identificateur fixe (le nom de la variable),
- Un type fixe (la nature de la variable : ce point sera étudié dans la section suivante (2.3)),
- Une valeur variable (le contenu de la variable).

- **Déclaration**
  Toute variable doit être déclarée avant qu'elle soit utilisée. Sa déclaration se fait de la façon suivante : précéder l'identificateur de la variable à déclarer par le mot clé **Var**, puis, préciser le type souhaité de cette variable.
  Var Id_var : type_var ;

- **Exemple**
  Algorithme declar_var ;
  Var x : entier ; y : reel ;

- **Remarques Importantes**

  - Une variable ou une constante est un espace mémoire qui contient des données soit fixe (constante), soit qui change au fur et à mesure que le programme avance dans son exécution (variable). Cependant, à un instant donné, une variable ne peut contenir

qu'une seule donnée (valeur).
- Quand on déclare plusieurs constantes et plusieurs variables, il n'est pas nécessaire de répéter les mots clés 'VAR' et 'CONST' pour chaque déclaration.
- Quand on déclare plusieurs variables de même type, il n'est pas nécessaire de répéter l'écriture de ce type pour chaque variable. Il suffit de séparer les différentes variables par des virgules et mettre le type une seule fois.

2.3 Types de données

Le type d'une variable définit l'ensemble des valeurs qu'elle peut prendre. Cet ensemble doit être fini.
Un type est caractérisé par :
- ses valeurs
- les opérations qui peuvent s'effectuer sur des variables ayant ce type (voir la section 2.4).

En algorithmique, le type d'une variable peut être un type prédéfini (on l'appelle aussi type de base, type simple, type scalaire ou type élémentaire), comme il peut être défini par l'utilisateur (type intervalle et type énuméré).

2.3.1 Types de base

Dans ce qui suit, nous allons définir les différents types de base en algorithmique et nous allons donner le nombre d'octets occupé par une variable de chaque type. Car, en machine, le type d'une variable *détermine la taille de l'espace mémoire occupé* par cette variable.

2.3.1.1 Type entier

Une variable de type *entier* peut prendre comme valeur l'ensemble des nombres entiers signés. Un entier est représenté sur 16 bits (2 Octets).

- **Exemple**
  Var NB_etudiants, annee : **entier** ;

2.3.1.2 Type reel

Une variable de type reel peut prendre comme valeur l'ensemble des nombres réels. Un réel est représenté sur 4 ou 6 octets.

- **Exemple**
  Var moyenne, prix : **reel** ;

2.3.1.3 Type Booléen

Appelé aussi type logique (*Booléen, c'est par rapport au nom du mathématicien Georges BOOLE*). Une variable de type booléen prend comme valeur *VRAI* ou *FAUX*. En mémoire, les booléens sont codés sur *1 bit*, avec 0 pour FAUX et 1 pour VRAI.

- **Exemple**
  Var existe : **booleen** ;

2.3.1.4 Type Caractère

Il est réservé aux variables contenant *un et un seul caractère*. Il s'agira en l'occurrence des lettres (minuscules et majuscules), des chiffres, des signes de ponctuation et des caractères spéciaux. Un caractère est toujours noté entre guillemet et il est représenté sur un octet.

- **Exemple**
  Var C : **caractere** ;

C est une variable caractère qui peut prendre une valeur caractère, comme : "x ", "C " , "/"ou "2".

- **Remarques importantes**

  – Le blanc (espace) est un caractère,
  – Les valeurs suivantes sont différentes et ne seront pas codés de la même manière dans la mémoire de la machine. Par exemple :
  "1" est un caractère,
  1 est un entier,
  1. est un réel.

  – On considère souvent que les caractères sont ordonnés dans l'ordre alphabétique.

2.3.1.5 Chaîne de caractères

Une variable chaîne de caractère est une suite de caractères. Elle est toujours notée entre guillemets. Le nombre d'octets utilisé pour représenter une chaîne de caractères est égal au nombre des caractères qui la compose.

- **Exemple**
  Var Ch : **chaine** ;

La variable *Ch* peut prendre n'importe quelle chaîne de caractères, comme : "informatique2022", "étudiant" ou "16/11/2021", etc.

2.3.2 Type Enuméré et Type Intervalle

Dans certains cas, on a besoin de définir des nouveaux type inexistant en algorithmique et qui sont nécessaires pour la résolution du problème posé. Dans cette sous-section, nous présentons les deux nouveaux types :

2.3.2.1 Type Enuméré

Le type énuméré consiste à définir la liste complète des valeurs qui peuvent être attribuées à une variable appartenant à ce type énuméré.
La définition du type énuméré se fait de la façon suivante :

- Utiliser le mot clé Type,
- Suivre le mot clé type par l'identificateur du type à définir,
- Entre deux parenthèses, citer toutes les valeurs de ce type en les séparant par des virgules.

Type Id_Type_ Enum = ($valeur_1$, $valeur_2$,...$Valeur_n$) ; En suite, des variables appartenant à ce type énuméré Id_Type_Enum peuvent être définies.

- **Exemple**
  Algorithme Ex_type_enum ;
  Type Couleur = (BLEU, BLANC, VERT, ROUGE, VIOLET) ;
  Var Clr1, Clr2 : Couleur ;

Ces variables Clr1 et Clr2 ne peuvent prendre de valeur que dans la liste BLEU, BLANC, VERT, ROUGE, VIOLET.

- **Remarque importante**
  Les valeurs de la liste sont considérées par le compilateur comme des *constantes entières ordonnées*, qui valent par défaut 0, 1, 2, 3 etc. (dans l'exemple ci_dessus, BLEU vaut 0, BLANC vaut 1 ... VIOLET vaut 4). Les variables de type énuméré appartiennent donc à un *sous_ ensemble des entiers*. On peut les utiliser pour faire tout ce qui est autorisé

sur un entier : indice de tableau (*voir le chapitre 5*), compteur de boucle pour (*voir le chapitre 4*) , etc.

2.3.2.2 Type Intervalle

Le type intervalle est un type dont les données prennent leurs valeurs dans une portion de l'intervalle des valeurs d'un autre type (Entier, Enuméré, Caractère) par l'indication de la borne inférieure et la borne supérieure de l'intervalle. La borne inférieure doit être inférieure de la borne supérieure.
La définition du type intervalle se fait de la façon suivante :
Type Id_Type_ Int = BInf.. BSup ;
Où *BInf* et *BSup* sont des constantes du même type, et sont les bornes inférieures et supérieures de l'intervalle *ID_ Type_ Int* (BInf et BSup sont inclus).
En suite, des variables appartenant à ce type intervalle *Id_ Type_ Int* peuvent ensuite être définies.

- **Exemple**
  Algorithme Ex_type_Int ;
  Type T_Int = 5 .. 70 ;
  Lettre_min= "a" .. "b" ;
  Var X, Y : T_Int ; m, n : Lettre_min ;

2.4 Opérations de base

Dans cette section, nous présentons d'abord les deux concepts (opérande et opérateur) , puis nous détaillons les principaux opérateurs du langage algorithmique ainsi que leurs priorités.

2.4.1 Opérande et Opérateur

Un opérateur est un outil qui permet d'agir sur une variable ou d'effectuer des calculs.

Un opérande est une donnée utilisée par un opérateur.

- **Exemple**
  X + 10

Dans cet exemple + désigne l'opérateur ; *x* et *10* sont les opérandes.

2.4.2 Types d'opérateurs

Le type d'opérateur dépend du type de l'opérande (la valeur de la constante ou de la variable) sur lequel il s'applique.
On distingue trois types d'opérateurs :

1. Opérateurs arithmétiques,
2. Opérateurs logiques,
3. Opérateurs de comparaison.

2.4.2.1 Opérateurs arithmétiques

1. L'addition, notée (+),
2. La soustraction, notée (-),
3. La multiplication, notée (*),
4. La division réelle, noté (/),
5. La division entière, notée (div). L'opérateur div donne le quotient de la division euclidienne,
   - **Exemple**
     9 div 2 = 4.
     9 /2 = 4.5
6. Le reste de division (modulo), noté (mod).
   - **Exemple**
     9 mod 2 = 1.

2.4.2.2 Opérateurs logiques

Les opérateurs logiques /booléens combinent des opérandes booléens pour former des expressions logiques plus complexes.
Les opérateurs logiques usuels sont *ET*, *OU* et *NON* qui sont donnés dans la table de vérité (voir la table 2.1) : Où A et B sont des valeurs / expressions booléennes.

2.4.2.3 Opérateurs de comparaison

Les opérateurs de comparaison, notés (>, <, = , < >, >=, < =). Ils permettent de comparer les valeurs de type entier, reel, caractere et chaine. Le résultat de cette comparaison

A	**B**	**NON (A)**	**A ET B**	**A OU B**
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

TABLE 2.1 – Table de vérité.

est une valeur booléenne (vrai ou faux).

- **Remarque**
  On peut appliquer l'opérateur (+) sur des chaînes de caractères. Dans ce cas, cet opérateur s'appelle opérateur de concaténation, il permet de créer une chaîne de caractères à partir de deux chaînes de caractères (ou plus) en les mettant bout à bout.

- **Exemple**
  'algo' + 'rithmique' donne 'algorithmique'.

2.4.3 Priorités des opérateurs

Lors de l'évaluation d'une expression, on classe les différents opérateurs par ordre de priorité, les opérateurs de plus forte priorité seront réalisés avant ceux de plus faible priorité (chaque opérateur est associée une priorité). La priorité de chaque opérateur permet de définir l'ordre d'exécution des différentes opérations. Dans ce qui suit, nous montrons la priorité de chaque opérateur présenté dans les sous-sections précédentes. Nous commençons par la plus haute priorité et nous descendons vers la priorité la plus basse :

1. ( ),
2. NON,
3. ET, *, /, div, mod,
4. OU, +, -,
5. <, <=, >, >=, =, < >.

- **Remarques importantes**
  - Les expressions sont constituées à l'aide de variables déjà déclarées, de valeurs, de parenthèses et d'opérateurs du (des)type(s) des variables concernées.
  - La machine effectue un seul calcul à la fois, de ce fait lorsqu'une expression contient plusieurs opérateurs de même priorité, l'opérateur le plus à gauche est évalué en premier.

- Pour lever toute ambiguïté ou pour modifier l'ordre d'exécution, on peut utiliser des parenthèses.
- S'il y a plusieurs parenthèses, les plus internes sont prioritaires.
  - **Exemple**
    Soient E1 et E2 deux expressions arithmétique et logique respectivement. Tel que : E1= 5+4*2-6/3 et E2= vrai ou faux et (Non (vrai) et faux).
    L'évaluation de l'expression E1 sera faite comme suit :

    E1= 5+4*2-6/3

    E1= 5+8-6/3

    E1=5+8-2

    E1=13-2

    E1=11.

    L'évaluation de l'expression E2 sera faite comme suit :

    E2= vrai ou faux et (Non (vrai) et faux)

    E2= vrai ou faux et (faux et faux)

    E2= vrai ou faux et faux

    E2= vrai ou faux

    E2= faux.

2.5 Construction d'un algorithme simple

Dans cette section, nous allons étudier les différentes instructions de base dont nous avons besoins pour résoudre un problème simple, comme le calcul de la surface d'un disque, le calcul de la moyenne de deux entiers, etc.

2.5.1 Instructions de base

Les instructions de base dans un algorithme sont les suivantes :
- Instruction de lecture,
- Instruction d'affectation,
- Instruction d'écriture.

2.5.1.1 Instruction de lecture

L'instruction de lecture (saisie) permet d'affecter des valeurs aux différentes variables déclarées et les stokers dans des cases mémoires. Cette opération se réalise en utilisant la syntaxe suivante :

Lire (V1, V2, V, ...Vn) ; où **V1**, **V2**, **V3**, ...**Vn** représentent l'ensemble des variables à lire.

- **Exemple**

  Algorithme Ssaisie ;
  Var A : **entier** ; x, y, z : **reel** ; nom :**chaine** ;
  Debut
  | Lire (A) ; */* donner une valeur entière à la variable A, comme -10, 17 ou 100, etc*/*
  | Lire (x, y, z) ; */* donner trois valeurs réelles aux trois variables x, y et z respectivement*/*
  | Lire (nom) ; */* donner une valeur chaîne à la variable nom, comme "Ali" , "Anes" ou "Imene", etc.*/*
  Fin.

2.5.1.2 Instruction d'affectation

L'affectation permet d'affecter le contenu d'une valeur, d'une constante, d'une variable ou d'une expression à une variable du même type. Elle est symbolisée en algorithmique par "←"

- **Exemple**

  Algorithme affectation ;
  Const B = 10 ;
  Var A, C, D :**entier** ;
  Debut
  | A ← 18 ; */* l'affectation d'une valeur à une variable */*
  | C ← B ; */* l'affectation d'une constante à une variable */*
  | D ← A ; */* l'affectation d'une variable à une variable */*
  | A ← (A+ B)*C-2 ; */* l'affectation d'une expression à une variable */*
  Fin.

2.5.1.3 Instruction d'écriture

Une instruction d'écriture (de sortie /d'affichage) nous permet dans un programme d'afficher un résultat (données traitées) ou bien un message (chaîne de caractères).

- **Exemple**

  Algorithme affichage ;
  Var A :**entier** ;

Debut
| A ← 18 ;
| Ecrire (A) ; */* c'est la valeur 18 qui s'affiche sur l'écran */*
| A ← (A-10)*3 ;
| Ecrire (A) ; */* c'est la valeur 24 qui s'affiche sur l'écran */*
| Ecrire ("A") ; */*c'est la lettre A qui s'affiche sur l'écran */*
| Ecrire ("mon premier affichage") ; */*c'est le message mon premier affichage qui s'affiche sur l'écran*/*
Fin.

2.5.2 Déroulement d'un algorithme / Trace d'exécution d'un algorithme

Le déroulement d'un algorithme / Trace d'exécution d'un algorithme (on l'appelle aussi, l'exécution manuelle) permet de suivre le bon déroulement d'un l'algorithme pas à pas afin de s'assurer de son bon fonctionnement. Il s'agit tout simplement d'un tableau qui contient en lignes les instructions et en colonnes les variables et nous donnons des valeurs aux différentes variables après l'exécution de chaque instruction.

- **Exemple**
  Dérouler l'algorithme suivant, en donnant la valeur de chaque variable après l'exécution de chaque instruction.
  Algorithme deroulement ;
  Var A, B, C :**entier** ;
  Debut
  | A ← 2 ;
  | A ← A+2 ;
  | B ← A*2 +A ;
  | C ← 4 ;
  | C ← B-C ;
  | C ← C+A-B ;
  | A ← B-C*A ;
  | A ← (B-A)*C ;
  | B ← (A+C)*B ;
  Fin.

	A	**B**	**C**
A ← 2 ;	2		
A ← A+2 ;	4		
B ← A*2 +A ;	4	12	
C ← 4 ;	4	12	4
C ← B-C ;	4	12	8
C ← C+A-B ;	4	12	0
A ← B-C*A ;	12	12	0
A ← (B-A)*C ;	0	12	0
B ← (A+C)*B ;	0	0	0

TABLE 2.2 – Déroulement de l'algorithme précédent.

2.6 Représentation d'un algorithme par un organigramme

Un algorithme peut être représenté par une solution graphique qu'on appelle *Organigramme*. Ce dernier constitue un diagramme qui permet de montrer les différentes étapes principales de résolution du problème. Il permet aussi de ne pas représenter explicitement les détails triviaux de l'algorithme pour se concentrer sur l'essentiel.
Les opérations dans un organigramme sont représentées par les symboles dont les formes sont normalisées. Ces symboles sont reliés entre eux par des lignes et des fléchées qui indiquent le chemin d'exécution. Pour plus de détails voir la figure 2.2 :

- **Exemple**
  Etablir l'organigramme qui calcule la valeur absolue d'un entier X.

2.7 Traduction en langage C

Comme nous l'avons indiqué précédemment au chapitre 1, un programme est la traduction d'un algorithme dans le langage de programmation utilisé.
Dans ce polycopié, nous utilisons le langage de programmation C. Pour cela, des compléments de ce langage seront fournis au fur et à mesure de l'avancement du cours et en fonction des divers concepts abordés.
En effet, à partir du chapitre suivant de ce polycopié, nous allons accompagner à chaque nouvelle notion en algorithmique sa notion correspondante en Langage C. Nous allons, aussi, présenter les différents composants principaux de ce langage, dans une annexe, à la fin du polycopié afin d'aider l'étudiant à comprendre davantage ces notions.

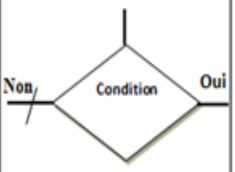
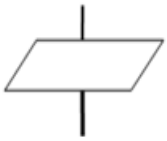
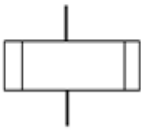
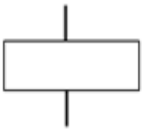
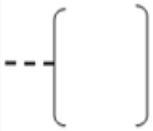
Symbole	Désignation	Symbole	Désignation
	pour représenter le début ou la fin d'un organigramme		Pour représenter un test (une condition qui est vraie ou fausse)
	Pour indiquer une opération d'Entrée /Sortie		Pour représenter un appel d'un sous-programme
	Pour indiquer une opération d'affectation		Représente une ligne de liaison qui relie les différents symboles (de haut en bas)
	Pour représenter un commentaire		Représente une ligne de liaison qui montre un cheminement différent (de bas en haut).

FIGURE 2.2 – Les différents symboles constituant un organigramme.

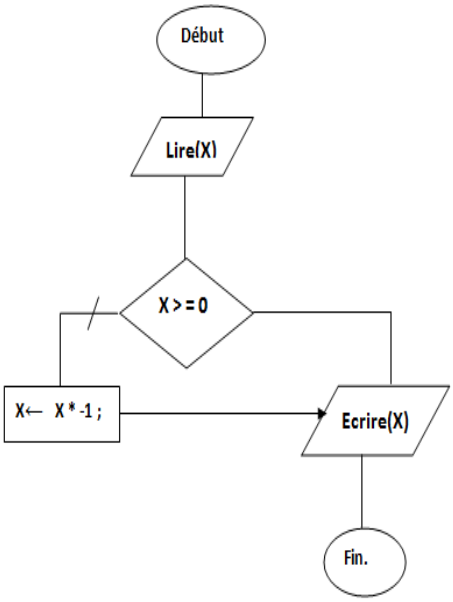


FIGURE 2.3 – L'organigramme qui calcule la valeur absolue d'un entier X.

2.8 Exercices récapitulatifs avec solutions

Exercice 1 :

Soient A et B deux booleens.

Soient x et y deux entiers ;

Compléter le tabeau suivant :

Expression logique (E)	**Complément de E (non(E))**
(A et B) ou Non (A)	
(A ou B) ou (Non (A) et B)	
x = y	
x < y	
x <= y	
x > y	
(x = 0) et (y <> 0)	
(x < 0) ou (y <> 0)	

TABLE 2.3 – Complément d'une expréssion logique (exercice).

Exercice 2 :

En utilisant les priorités des différents opérateurs étudiés, évaluez les expressions suivantes :

- E1= 2*6 + 4*2-10.
- E2= vrai et faux ou vrai
- E3= vrai et (faux ou vrai)
- E4= (15>20) et non ((vrai ou faux) et (vrai et faux))

Exercice 3 :

Ecrire un algorithme permettant de lire les notes obtenues par un étudiant en module Algorithmique (EMD, TD, TP) et de calculer sa moyenne.

Exercice 4 :

Ecrire un algorithme permettant de lire deux variables réelles et de les permuter.

Exercice 5 :

Ecrire un algorithme permettant de lire un entier n et d'afficher son chiffre de dizaine ainsi que son chiffre de centaine.

Solution_Exo1 :

Expression logique (E)	**Complément de E (non(E))**
(A et B) ou Non (A)	(Non(A) ou Non (B)) et A
(A ou B) ou (Non (A) et B)	(Non(A) et Non (B)) et (A ou Non (B))
x = y	X < > y
x < y	x> = y
x <= y	X > y
x > y	X < = y
(x = 0) et (y <> 0)	(x < > 0) ou (y = 0)
(x < 0) ou (y <> 0)	(x > = 0) et (y = 0)

TABLE 2.4 – Complément d'une expréssion logique (solution).

Solution_Exo2 :

L'évaluation de l'expression E1 :

E1= 2*6 + 4*2-10

= 12+4*2-10

= 12+ 8-10

= 20-10

= 10.

L'évaluation de l'expression E2 :

E2= vrai et faux ou vrai

= faux ou vrai

= vari.

L'évaluation de l'expression E3 :

E3= vrai et (faux ou vrai)

= vrai et vrai

= vrai.

L'évaluation de l'expression E4 :

E4= (15>20) et non ((vrai ou faux) et (vrai et faux))

= faux et non ((vrai ou faux) et (vrai et faux))

= faux et non (vrai et (vrai et faux))

= faux et non (vrai et faux)

= faux et non (faux)

= faux et vrai

= faux.

Solution_Exo3 :

```
Algorithme Exo3 ;
Var EMD, TD, TP, Moy :reel ;
Debut
|    Lire(EMD, TD,TP) ;
|    Moy ← 0.4 * (TD+TP) / 2 + 0.6 * EMD ;
|    Ecrire(Moy) ;
Fin.
```

Solution_Exo4 :

```
Algorithme Exo4 ;
Var a, b, c :reel ;
Debut
|    Lire(a,b) ;
|    c ← a ;
|    a ← b ;
|    b ← c ;
|    Ecrire(a, b) ;
Fin.
```

Solution_Exo5 :

```
Algorithme Exo5 ;
Var a, b, n :entier ;
Debut
|    Lire(n) ;
|    n ← n div 10 ;
|    a ← n mod 10 ;
|    b ← (n div 10) mod 10 ;
|    Ecrire('les chiffres demandés sont : ',a ,b ) ;
Fin.
```

Chapitre 3

Structures conditionnelles

Sommaire

Objectif du chapitre

- Comprendre l'utilité d'une structure conditionnelle,
- Connaître les différentes structures conditionnelles disponibles en algorithmique,
- Résoudre des problèmes nécessitant l'utilisation des structures alternatives.

3.1 Introduction

Dans le chapitre précédant, nous avons étudié les actions simples (voir la section 6) où les problèmes traités possèdent des solutions élémentaires constituées d'une suite finie et ordonnée d'actions simples.
En réalité, les problèmes sont plus complexes que ça. La résolution de certains d'entre eux ne peut se faire que sous condition. Comme par exemple le cas suivant :
A la fin d'année, le passage d'un étudiant d'un niveau à un autre dépend de sa moyenne générale obtenue durant l'année universitaire :

SI la moyenne générale supérieure ou égale à 10,

ALORS l'étudiant passera au niveau supérieur.

SINON l'étudiant restera dans le même niveau.

- La moyenne générale obtenue est une *condition.*

  - **Une condition** est une expression booléenne, elle est soit vraie soit fausse. Fréquemment, cette condition se présente sous forme de comparaison en utilisant les opérateurs de comparaison (vus en chapitre 2), c'est à dire : <, >, <=, >=, =, <>.

  La condition est soit "simple", soit "composée" :dans le cas d'une condition composée, on utilise les opérateurs logiques (vus dans le chapitre 2), qui sont : ET, OU, NON.
- L'étudiant passera au niveau supérieur *est l'action à faire si la condition est vraie.*
- L'étudiant restera dans le même niveau *est l'action à faire si la condition est fausse.*

On doit alors trouver une autre structure algorithmique capable de prendre en charge les différents traitements relatifs aux différentes conditions et de déclencher exclusivement le traitement qui respecte une certaine condition.
Cette structure est appelée structure conditionnelle. On distingue plusieurs formes :

1. Structure conditionnelle simple,
2. Structure conditionnelle composée,
3. Structure conditionnelle imbriquée,
4. Structure conditionnelle à choix multiple.

Dans ce qui suit, nous allons détailler chaque structure en mettant l'accent sur sa syntaxe en algorithmique et en langage C ainsi que son organigramme.

3.2 Structure conditionnelle simple

- **Définition**

  Il s'agit d'un traitement qui ne peut s'exécuter que si une condition logique est satisfaite ; dans le cas contraire, rien ne devrait se passer.

- **Syntaxe**

  La syntaxe de la structure conditionnelle simple est donnée dans la table 3.1.

En langage algorithmique	**En langage C**

Si (Condition_Satisfaite) **alors**
Action(s) ;
Finsi ;
	If (Condition_Satisfaite) {
	Action(s) ;
	}

TABLE 3.1 – Syntaxe de la structure conditionnelle simple.

- **Organigramme correspondant**

  La représentation graphique de la structure conditionnelle simple se fait selon l'organigramme suivant :

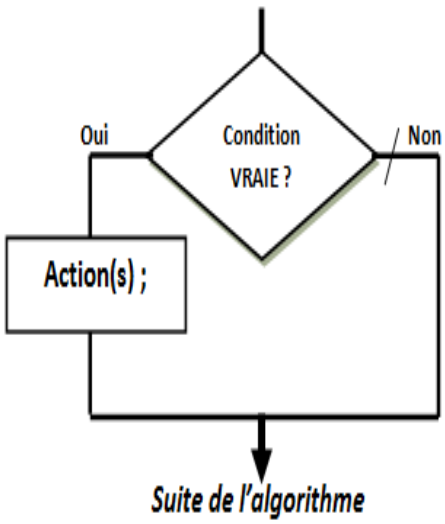


FIGURE 3.1 – Organigramme de la structure conditionnelle simple.

Interprétation : dans cet organigramme, la condition est évaluée. Si elle vaut vrai alors c'est la partie Action(s) qui sera exécutée sinon c'est l'action qui suit l'action conditionnelle dans l'algorithme qui sera exécutée.

- **Exemple**

  Ecrire un algorithme/ un programme en langage C qui affiche la valeur absolue d'un nombre entier A. Puis, établir l'organigramme correspondant.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Valeur_absolue ;	#include<stdio.h>
Var A :**entier** ;	int main()
Debut	{
Ecrire("Donner A") ;	int A ;
Lire(A) ;	printf("Donner A :") ;
Si (A<0) **Alors**	scanf("%d",&A) ;
A ⟵ A*(-1) ;	**if** (A<0) {
Finsi	A= -A ;}
Ecrire ('la valeur absolue de A =', A)	printf("la valeur absolue de A = %d",A) ;
Fin.	return 0 ;
	}

TABLE 3.2 – Solution algorithmique /programme C (Exemple_conditionnelle simple).

- **Solution_Organigramme Correspondant**

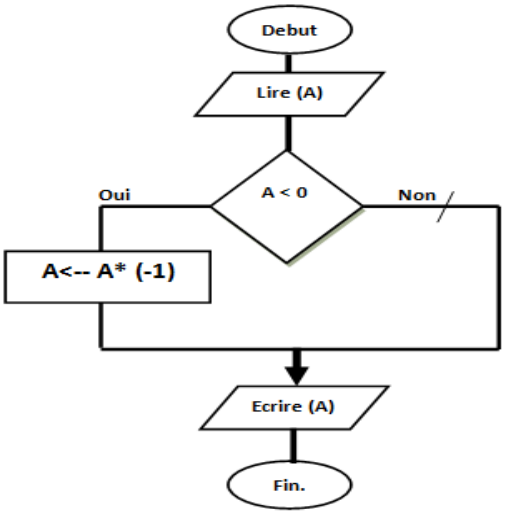


FIGURE 3.2 – Organigramme de la structure conditionnelle simple (solution).

3.3 Structure conditionnelle composée

- **Définition**

  La structure conditionnelle composée (appelée aussi la structure conditionnelle à deux choix ou la structure conditionnelle alternative) est une structure algorithmique qui fait appel au maximum à deux traitements.

- **Syntaxe**

  La syntaxe de la structure conditionnelle composée est la suivante :

Si (Condition_Satisfaite) **alors**	**If** (Condition_Satisfaite) {
Action(s)1 ;	Action(s)1 ;
Sinon ;	}**else**{
Action(s)2 ;	Action(s)2 ;
Finsi ;	}

TABLE 3.3 – Syntaxe de la structure conditionnelle composée.

- **Organigramme correspondant**

  La représentation graphique de la structure conditionnelle composée se fait selon l'organigramme suivant :

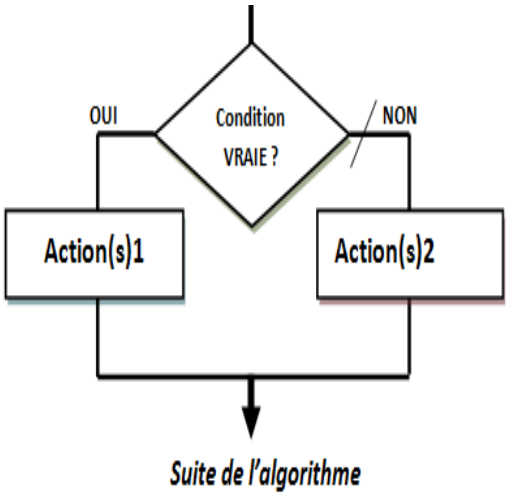


FIGURE 3.3 – Organigramme de la structure conditionnelle composée.

Interprétation : dans cet organigramme, la condition est évaluée. Si elle vaut vrai, alors c'est la partie Action(s)1 qui sera exécutée, sinon c'est la partie Action(s)2 qui sera exécutée.

- **Exemple**

  Ecrire un algorithme/ un programme en langage C qui détermine si un entier A donné par l'utilisateur est pair ou impair. Puis, établir l'organigramme correspondant.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Pair_Impair ;	#include<stdio.h>
Var A :**entier** ;	int main()
Debut	{ int A ;
Ecrire ("Donner la valeur de A :") ;	printf("Donner la valeur de A :") ;
Lire(A) ;	scanf("%d",&A) ;
Si (A mod 2 = 0) **Alors**	**if** (A%2 == 0) {
Ecrire ('A est un nombre pair') ;	printf("A est un nombre pair") ; }
Sinon	else {
Ecrire ('A est un nombre impair') ;	printf("A est un nombre impair") ; }
Finsi ;	return 0 ; }
Fin.	

TABLE 3.4 – Solution algorithmique /programme C (Exemple_conditionnelle composée).

- **Solution_Organigramme Correspondant**

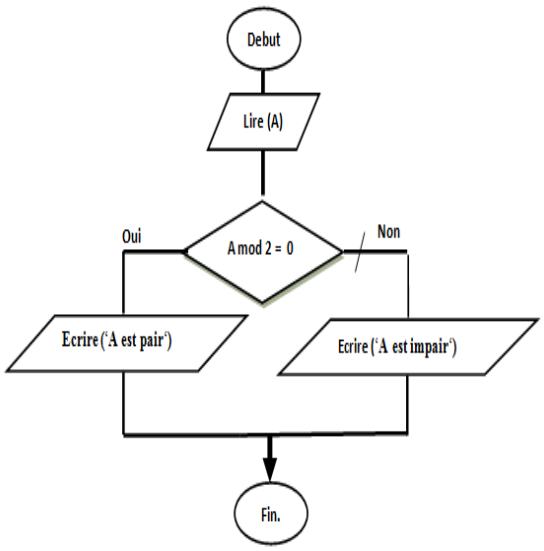


FIGURE 3.4 – Organigramme de la structure conditionnelle composée (solution).

3.4 Structure conditionnelle imbriquée

- **Définition**

  La structure imbriquée est une généralisation de la structure conditionnelle composée, lorsque le nombre de traitements différents est plus grand que deux (c'est-à-dire incluses les unes dans les autres).

- **Syntaxe**

  La syntaxe de la structure conditionnelle imbriquée est la suivante :

Si (Condition1_Satisfaite) **alors**	**If** (Condition1_Satisfaite) {
Action(s)1 ;	Action(s)1 ; }
Sinon ;	**else** {
Si (Condition2_Satisfaite) **alors**	**If** (Condition2_Satisfaite) {
Action(s)2 ;	Action(s)2 ; }
Sinon ;	... ;
... ;	... ;
... ;	... ;
... ;	**If** (ConditionN-1_Satisfaite){
Si (ConditionN-1_Satisfaite) **alors**	Action(s)N-1 ; }
Action(s)N-1 ;	**else**{
Sinon ;	Action(s)N ;}
Action(s)N ;	
Finsi ;	
... ;	
... ;	
... ;	
Finsi ;	
Finsi ;	

TABLE 3.5 – Syntaxe de la structure conditionnelle imbriquée.

- **Remarque Importante**

  Action(s)1, Action(s)2, ...Action(s) N-1 et Action(s)N peuvent être elles même des instructions conditionnelles.

- **Organigramme correspondant**

  La représentation graphique de la structure conditionnelle imbriquée se fait selon l'organigramme suivant :

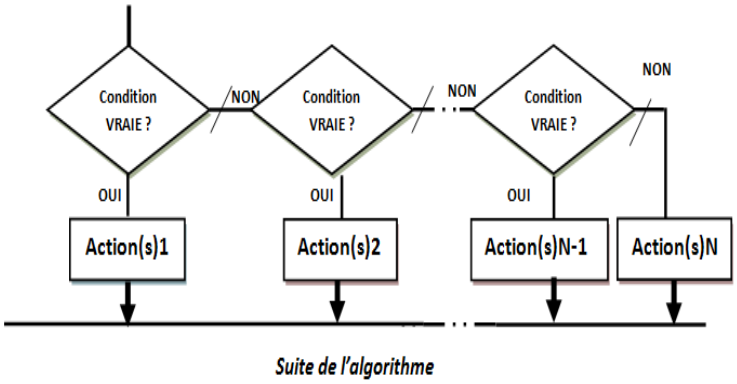


FIGURE 3.5 – Organigramme de la structure conditionnelle imbriquée.

Interprétation :

- A la première Condition vraie, l'Action correspondante est exécutée et l'algorithme passe à l'instruction qui suit la fin si de cette condition.
- Si aucune Condition n'est vraie, c'est l'Action N qui sera exécutée.

- **Exemple**

  Ecrire un algorithme/ un programme en langage C qui détermine le signe d'un entier A (positif, négatif ou nul). Puis, établir l'organigramme correspondant.

- **Solution_Organigramme Correspondant**

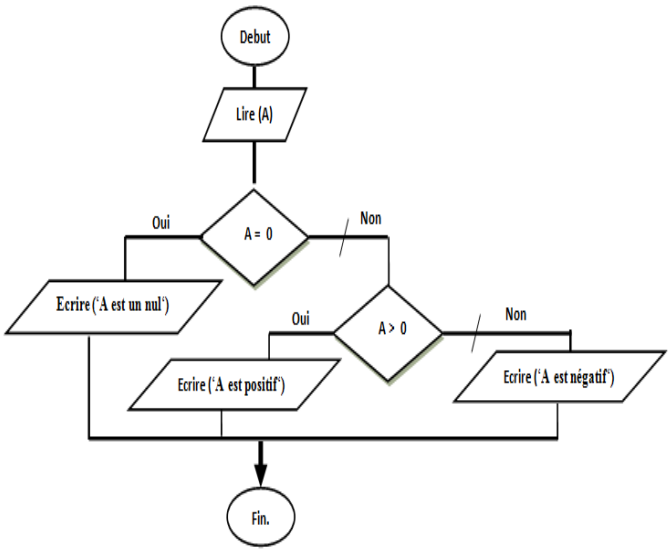


FIGURE 3.6 – Organigramme de la structure conditionnelle imbriquée (solution).

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Signe_nmbre ;	#include<stdio.h>
Var A :**entier** ;	int main()
Debut	{
Lire(A) ;	int A ;
Si (A=0) **Alors**	printf("Donner la valeur de A :") ;
Ecrire ('A est nul') ;	scanf("%d",&A) ;
Sinon	**if** (A == 0) {
Si (A > 0) **Alors**	printf("A est nul") ; }
Ecrire ('A est positif') ;	else {
Sinon	**if** (A > 0) {
Ecrire ('A est négatifl') ;	printf("A est positif") ; }
Finsi ;	else { printf("A est négatif") ; }
Finsi ;	}
Fin.	return 0 ;
	}

TABLE 3.6 – Solution algorithmique /programme C (Exemple_conditionnelle imbriquée).

3.5 Structure conditionnelle à choix multiple

- **Définition**

  La structure conditionnelle à choix multiple est une structure conditionnelle utilisée de préférence dans des structures à plusieurs traitements (Actions) selon des conditions. Le choix d'un traitement (Action) se fait suivant la valeur d'un sélecteur.

- **Syntaxe**

  La syntaxe de la structure conditionnelle à choix multiple est donnée dans la figure 3.7.

- **Remarques importantes**
  - Le sélecteur doit être de type simple discret.
  - Valeur i peut être un ensemble des valeurs qui seront séparées par des virgules ou intervalle des valeurs (Valeur_initiale .. Valeur_finale).
  - La structure conditionnelle à choix multiple est un raccourci d'écriture pour la structure.

Selonque (selecteur) **Faire** Valeur1 : Action(s)1 ; Valeur2 : Action(s)2 ; .... ValeurN : Action(s)N ; **Sinon** ; Action(s)N+1 ; **Fin selonque** ;	**Switch** (selecteur) { **case** Valeur1 : Action(s)1 ; **break** ; **case** Valeur2 : Action(s)2 ; **break** ; .... **case** ValeurN : Action(s)N ; **break** ; **Default** : Action(s)N+1 ; **break** ; }

TABLE 3.7 – Syntaxe de la structure conditionnelle à choix multiple.

- **Organigramme correspondant**

  La représentation graphique de la structure conditionnelle à choix multiple se fait selon l'organigramme suivant :

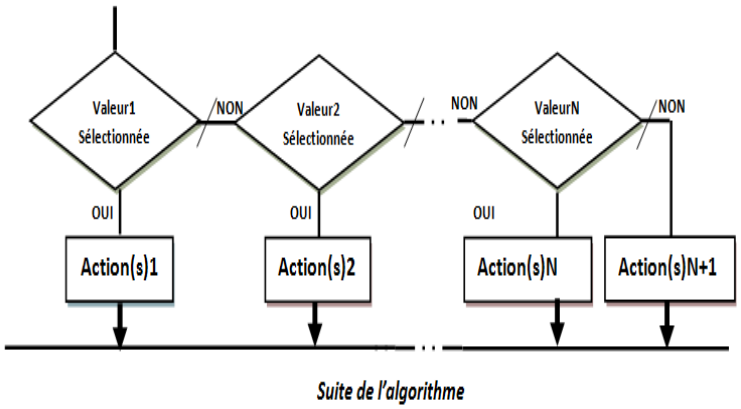


FIGURE 3.7 – Organigramme de la structure conditionnelle à choix multiple.

Interprétation :

- Evaluation de l'expression (le sélecteur), le plus souvent c'est une variable,
- Recherche la valeur du sélecteur dans toutes les valeurs (c'est-à-dire dans l'intervalle Valeur1..ValeurN), en parcourant celles-ci dans l'ordre où elles sont écrites,
- Dès que la valeur est trouvée, l'Action correspondante est exécutée et l'algorithme passe à l'instruction qui suit la fin selon,
- Si la valeur du sélecteur n'est trouvée dans aucune liste, c'est l'Action N+1 qui sera exécutée.

- **Exemple**

  Ecrire un algorithme qui lit un nombre entier A et qui affiche le mois correspond à la valeur de A.

En langage algorithmique	**En langage C**
Algorithme Valeur_absolue ;	#include<stdio.h>
Var A :**entier** ;	int main()
Debut	{
Lire(A) ;	int A ;
Selonque (A) **Faire**	printf("Donner la valeur de A :") ;
1 : Ecrire ("Janvier") ;	scanf("%d",&A) ;
2 : Ecrire ("Février") ;	**Switch**(A) {
3 : Ecrire ("Mars") ;	case 1 : printf("Janvier") ; break ;
4 : Ecrire ("Avril") ;	case 2 : printf("Février") ; break ;
5 : Ecrire ("Mai") ;	case 3 : printf("Mars") ; break ;
6 : Ecrire ("Juin") ;	case 4 : printf("Avril") ; break ;
7 : Ecrire ("Juillet") ;	case 5 : printf("Mai") ; break ;
8 : Ecrire ("Aout") ;	case 6 : printf("Juin") ; break ;
9 : Ecrire ("Septembre") ;	case 7 : printf("Juillet") ; break ;
10 : Ecrire ("Octobre") ;	case 8 : printf("Aout") ; break ;
11 : Ecrire ("Novembre") ;	case 9 : printf("Septembre") ; break ;
12 : Ecrire ("Décembre") ;	case 10 : printf("Octobre") ; break ;
Sinon	case 11 : printf("Novembre ") ; break ;
Ecrire ("aucun mois ne correspond à ce nombre", A) ;	case 12 : printf("Décembre ") ; break ;
Fin selonque ;	Default :
Fin.	printf("aucun mois ne correspond à ce nombre ") ;}
	return 0 ;
	}

TABLE 3.8 – Solution algorithmique /programme C (Exemple_conditionnelle à choix multiple).

3.6 Branchement Conditionnel

- **Définition**

  Le branchement conditionnel est une structure particulière permettant au système d'interrompre l'exécution séquentielle du programme, remonte ou descend à la ligne appelée *étiquette* et poursuit l'exécution à partir de celle-ci (c'est-à-dire, en " sautant " à une adresse identifiée au lieu de poursuivre l'exécution du code séquentiellement).

Pour réaliser un branchement, on doit indique la cible du branchement via une étiquette *<num_ etiq>*, en utilisant l'instruction *aller à <num_ etiq>* (voir la syntaxe ci-dessous).

- **Syntaxe**

  La syntaxe du branchement conditionnel est la suivante :

En langage algorithmique	**En langage C**
Aller à < Etiq>	**goto** < Etiq>
Action1 ;	Action1 ;
Action2 ;	Action2 ;
... ;	... ;
Action N-2 ;	Action N-2 ;
Etiq : Action N-1 ;	Etiq : Action N-1 ;
Action N ;	Action N ;
Action N+1 ;	Action N+1 ;

TABLE 3.9 – Syntaxe du branchement conditionnel.

- **Remarques importantes**

  Une étiquette désigne un seul endroit dans le programme, on ne peut pas indiquer deux endroits avec une même étiquette. Par contre, on peut réaliser plusieurs branchements vers une même étiquette.

- **Exemple**

  Ecrire un algorithme permettant de lire deux nombre entiers A et B différents (c'est-à-dire A < > B) et de les afficher.

- **Solution algorithme / programme C**

  La solution est donnée dans la table 3.10.

3.7 Exercices récapitulatifs avec solutions

Exercice 1 :

Soient A et B deux entiers.

Ecrire un algorithme permettant de ranger dans A la valeur la plus petite et dans B la valeur la plus grande.

Exercice 2 :

Ecrire un algorithme permettant de lire deux entiers A et B et d'afficher le nombre maximum.

En langage algorithmique	**En langage C**

Algorithme branchement ;
Var A, B :**entier** ;
Debut
1 : Ecrire ("Saisir deux valeurs entières différentes") ;
Lire(A, B) ;
Si (A = B) **Alors**
Aller à 1.
Finsi
Ecrire (A, B) ;
Fin.
	#include<stdio.h>
	int main()
	{
	int A, B ;
	1 : printf("Saisir deux valeurs entières différentes ") ;
	scanf("%d",&A) ;
	scanf("%d",&B) ;
	if (A==B) {
	goto <1> ;}
	printf("%d, %d",A, B) ;
	return 0 ;
	}

TABLE 3.10 – Solution algorithmique /programme C (Exemple_Branchement conditionnel).

Exercice 3 :

Ecrire un algorithme permettant de résoudre une équation du second degré : $aX^2 + bX + c = 0$. Tels que a, b et c sont des nombres réels.

Exercice 4 :

Ecrire un algorithme permettant de lire un caractère C et d'identifier sa nature (lettre minuscule, lettre majuscule, chiffre ou autre).

Exercice 5 :

Soit n un entier positif. Ecrire un algorithme permettant de lire n et de calculer la somme de tous les nombres entiers inférieurs ou égaux à n.

Solution_Exo1 :

Algorithme Exo1 ;
Var A, B, C :**entier** ;
Debut
| Lire(A, B) ;
| **Si** (A > B) **Alors**
| | C ← A ;
| | A ← B ;
| | B ← C ;
| **Finsi**
| Ecrire("le nombre le plus petit est ", A, " le nombre le plus grand est ", B) ;
Fin.

Solution_Exo2 :

```
Algorithme Exo2 ;
Var A, B, C :entier ;
Debut
|   Lire(A, B) ;
|   Si (A > B) Alors
|   |   Ecrire("le nombre maximum est A") ;
|   Sinon
|   |   Ecrire("le nombre maximum est B") ;
|   Finsi
Fin.
```

Solution_Exo3 :

```
Algorithme Exo3 ;
Var a, b, c, d, x1, x2 :entier ;
Debut
|   Lire(a, b, c) ;
|   Si (a=0) Alors
|   |   Si (b=0) Alors
|   |   |   Si (c=0) Alors
|   |   |   |   Ecrire("infinité de solutions") ;
|   |   |   | Sinon
|   |   |   |   Ecrire("absurde") ;
|   |   |   Finsi
|   |   Sinon
|   |   |   Ecrire(-c/b) ;
|   |   Finsi
|   Sinon
|   |   d ⟵ b*b-4a*c ;
|   |   Si (d < 0) Alors
|   |   |   Ecrire("cette éqaution n'a pas de solutions") ;
|   |   Sinon
|   |   |   Si (d=0) Alors
|   |   |   |   Ecrire(−b/2 ∗ a) ;
|   |   |   Sinon
|   |   |   |   x1 ⟵ (−b − sqrt(d))/2a ;
|   |   |   |   x1 ⟵ (−b + sqrt(d))/2a ;
|   |   |   |   Ecrire(x1, x2) ;
|   |   |   Finsi
|   |   Finsi
|   Finsi
```

Fin.

Solution_Exo4 :

```
Algorithme Exo4 ;
Var C :caractere ;
Debut
|    Lire(C) ;
|    Selonque (C) Faire
|        | "a"..."z" : Ecrire (" Lettre minuscule") ;
|        | "A"..."Z" : Ecrire (" Lettre majuscule'") ;
|        | 1...9 : Ecrire ("Chiffre") ;
|    Sinon
|        | Ecrire ("Ni lettre, Ni Chiffre") ;
|    Finselonque
Fin.
```

Solution_Exo5 :

```
Algorithme Exo5 ;
Var n, S :entier ;
Debut
|    S ⟵ 0 ;
|    1 : Lire(n) ;
|    Si ((n < 0) ou (n = 0)) Alors
|        | Aller à 1 ;
|    Sinon
|        | 2 : Si (n < > 0) Alors
|        |     | S ⟵ S + n ;
|        |     | n ⟵ n-1 ;
|        |     | Aller à 2 ;
|        | Finsi;
|    Finsi;
|    Ecrire(S) ;
Fin.
```

Chapitre 4

Structures répétitives (les boucles)

Sommaire

Objectif du chapitre

- Comprendre l'utilité d'une structure itérative,
- Connaître les différentes structures itératives disponibles en algorithmique,
- Résoudre des problèmes nécessitant l'utilisation des structures itératives.

4.1 Introduction

Dans les problèmes quotidiens, on ne traite pas uniquement des séquences d'actions, sous ou sans conditions, mais il peut être fréquent d'être obligé d'exécuter un traitement (séquence d'actions), plusieurs fois. En effet, pour saisir les N notes d'un étudiant et calculer sa moyenne, on est amené à saisir N variables, puis faire la somme et ensuite diviser la somme par N. Cette solution nécessite la réservation de l'espace par la déclaration des variables et une série de séquences d'écriture/lecture. Ce problème est résolu à l'aide des structures répétitives.
Une structure répétitive est une structure qui répète un même traitement autant de fois que l'on veut, le nombre de répétition peut :

- Etre connu (fixé à l'avance).
- Dépendre d'une condition permettant l'arrêt et la sortie de la boucle.

Il existe trois types de boucles :

- La boucle Pour,
- La boucle Tant Que,
- La boucle Répéter.

L'utilisation d'une telle ou telle boucle dépend essentiellement de la nature du problème à résoudre.

4.2 Boucle Pour

- **Définition**

  La boucle *pour* (*for* en langage C) permet de répéter une action (ou plusieurs actions) un nombre donné de fois. Elle se caractérise par le fait que l'on connait à l'avance le nombre d'itérations qu'on doit effectuer.

- **Syntaxe**

  La syntaxe de la boule Pour est comme suit :

En langage algorithmique	**En langage C**

Pour cpt allant de V_i à V_f [V_p] **faire**
Action(s) à répéter ;
Finpour ;
	for (/* Expression/Déclaration */ ; /* Condition */ ; /* Expression */) {
	Action(s) à répéter ;
	}

TABLE 4.1 – Syntaxe de la boucle Pour.

cpt : la variable qui joue le rôle d'un compteur.

V_i et V_f : la valeur initiale et la valeur finale du compteur cpt et doivent être connus à l'avance.

V_p : la valeur du pas, c'est la valeur qu'on rajoute au cpt à chaque fois qu'on termine l'exécution des Actions.

Action(s) : les instructions qu'on veut répéter

- **Fonctionnement**

  Le compteur *cpt* est initialisé à la valeur V_i. Si *cpt* est inférieur ou égale à la valeur V_f, les actions de la boucle sont exécutées et la variable *cpt* prend $cpt + V_p$. Le processus recommence ensuite pour exécuter la prochaine itération. Si $cpt > V_f$, l'exécution de la boucle s'arrête et l'exécution de l'algorithme continue après *finpour*.

En langage C, la boucle for se décompose en trois parties (ou trois clauses) :

- une expression et/ou une déclaration qui sera le plus souvent l'initialisation d'une variable. Elle est exécutée uniquement à la première itération.
- une condition d'arrêt de la boucle, lorsque sa valeur devient faux, l'exécution des instructions de la boucle s'arrête.
- une seconde expression, qui consistera le plus souvent en l'incrémentation du compteur de la boucle.

- **Remarques**

  - Il est totalement interdit de modifier la valeur du compteur cpt dans le corps de la boucle.
  - Les variables *cpt*, V_i et V_f doivent être de même type.
  - La valeur du pas peut être positive ou négative et par conséquent, il faut ; au départ de la boucle ; que $V_i <= V_f$ ou $V_i >= V_f$ selon la positivité ou la négativité de cette valeur.
  - La valeur du pas est égale à 1 par défaut.

- **Organigramme correspondant à la boucle Pour**

  La représentation graphique de la boucle Pour se fait selon l'organigramme donné dans la figure 4.1.

- **Exemple**

  Ecrire un algorithme qui calcule la somme des entiers positifs inférieurs ou égaux à 100.

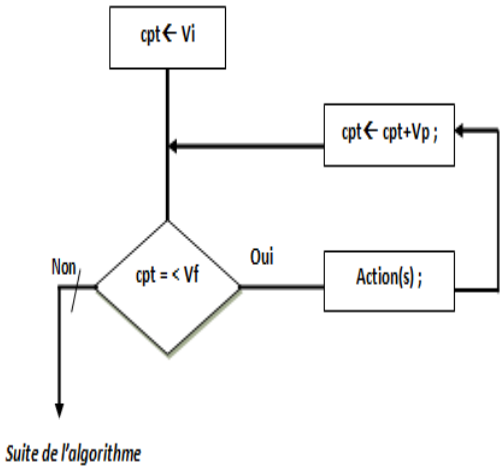


FIGURE 4.1 – Organigramme de la boucle Pour.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Boucle_pour ;	#include<stdio.h>
Var S, i :**entier** ;	int main()
Debut	{
S ⟵ 0 ;	int S, i ;
Pour i allant de 1 à 100 **Faire**	S=0 ;
S ⟵ S+i ;	for(i=1 ;i<=100 ;i++) {
Finpour;	S=S+i ; }
Ecrire ('la somme calculée est', S)	printf("la somme calculée est %d",S) ;
Fin.	return 0 ; }

TABLE 4.2 – Solution algorithmique /programme C (Exemple_Boucle Pour).

- **Solution_Organigramme Correspondant**
  La solution est donnée dans la figure 4.2.

4.3 Boucle Tantque

- **Définition**
  La boucle *tantque* (*while* en langage C) permet de répéter l'exécution d'une action (ou plusieurs actions) tant qu'une condition est vraie. C'est-à-dire, elle permet d'exécuter l'action / les actions de la boucle lorsque le nombre de répétition dépond d'une condition ; on s'arrêtera dès que la condition n'est plus vérifiée.

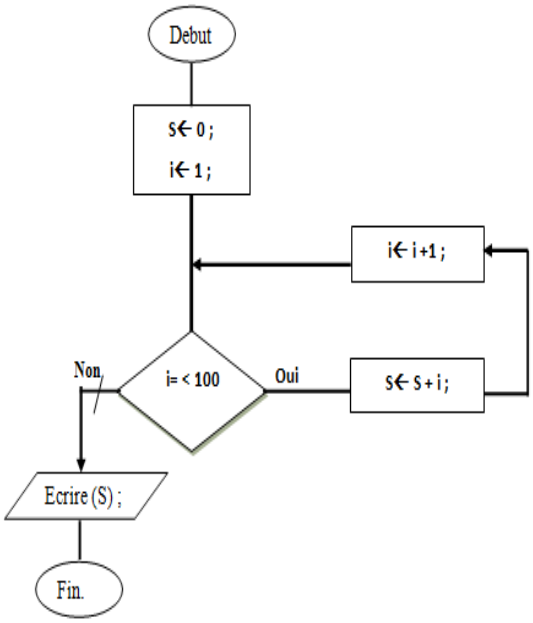


FIGURE 4.2 – Organigramme de la boucle Pour (solution).

- **Syntaxe**

  La syntaxe de la boule Tantque est comme suit :

En langage algorithmique	**En langage C**

Tantque (Condition_Satisfaite) **faire**
Action(s) à répéter ;
FinTantque ;
	while (Condition_Satisfaite) {
	Action(s) à répéter ;
	}

TABLE 4.3 – Syntaxe de la boucle Tantque.

- **Fonctionnement**

  La boucle Tantque fonctionne de la façon suivante. Au début de chaque itération, la condition est évaluée. Si elle est vraie l'ensemble des actions sera exécuté sinon on sort de la boucle. La condition d'accès à la boucle (elle s'appelle aussi la variable de boucle) doit être mise à jour à chaque itération. Donc, après avoir exécuté la partie Action(s), on re-teste la condition. Si elle est toujours vraie, on exécute le bloc d'actions une autre fois. On répète ce traitement jusqu'à ce que la condition soit fausse, et on continue alors l'exécution du reste de l'algorithme (voir la figure 4.3).

- **Organigramme correspondant à la boucle Tantque**

  La représentation graphique de la boucle Tantque se fait selon l'organigramme suivant :

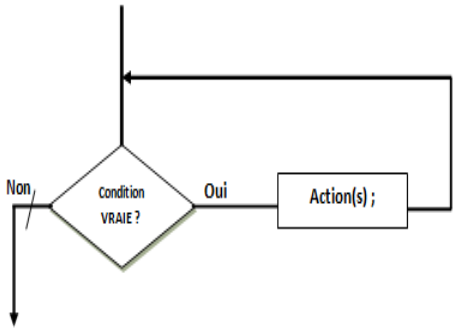


FIGURE 4.3 – Organigramme de la boucle Tantque.

- **Remarques**

  – La condition doit être initialisée avant d'entrer dans la boucle et doit être mise à jour à la fin de chaque itération.
  – Le nombre de répétitions dépendra de la condition.
  – Si la condition n'est pas vérifiée au début, alors la première exécution des action(s) ne sera pas faite du tout.

- **Exemple**

  Ecrire un algorithme qui calcule la somme des N entiers donnés par l'utilisateur. L'algorithme s'arrête quand l'utilisateur tape zéro.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Boucle_TQ ;	#include<stdio.h>
Var N, S :**entier** ;	int main()
Debut	{
Lire(N) ;	int N, S ;
S ⟵ 0 ;	scanf("%d",&N) ;
Tantque (N < > 0) **Faire**	S=0 ;
S ⟵ S+N ;	while (N < > 0) {
Lire(N) ;	S=S+N ;
FinTantque;	scanf("%d",&N) ; }
Ecrire ('la somme calculée est', S) ;	printf("la somme calculée est %d",S) ;
Fin.	return 0 ; }

TABLE 4.4 – Solution algorithmique /programme C (Exemple_Boucle Tantque).

- **Solution_Organigramme Correspondant**

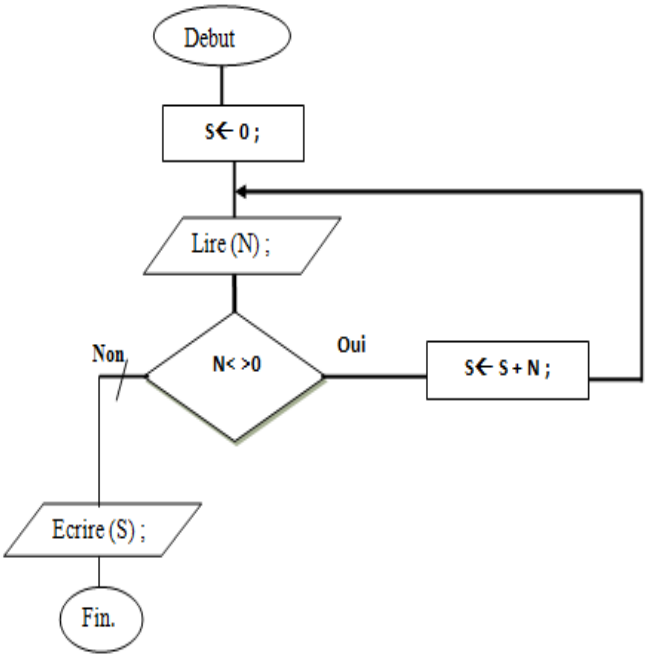


FIGURE 4.4 – Organigramme de la boucle Tantque (solution).

4.4 Boucle Répéter

- **Définition**

  La boucle *répéter* permet d'exécuter un bloc d'instructions d'abord, de façon inconditionnelle, puis de répéter son traitement tant qu'une condition est vraie. Elle est utilisée surtout dans le contrôle de saisie.

  En langage C, cette boucle est introduite par l'instruction *do*, suivie du bloc d'instructions. La condition est exprimée sous la forme d'une expression entre parenthèses, placée après un while.

- **Syntaxe**

  La syntaxe de la boule Repeter est comme suit :

En langage algorithmique	**En langage C**
Repeter Action(s) à répéter ; **Jusqu'à** (Condition_Satisfaite) ;	**do** { Action(s) à répéter ; } while (Condition_Non_Satisfaite)

TABLE 4.5 – Syntaxe de la boucle Repeter.

- **Fonctionnement**

  Le principe de fonctionnement de la boucle Repeter est le suivant :
  La partie Action(s) est exécuté. Puis, la condition est testée. Si elle est fausse, on sort de la boucle et on continue l'exécution normale de l'algorithme (voir la figure 4.5). Si elle est vraie, on exécute de nouveau le bloc, puis on re-teste la condition. Comme pour la boucle Tant que, il est donc nécessaire d'initialiser manuellement la variable de boucle avant Repeter. De plus, cette variable doit apparaître dans la condition, et doit être modifiée dans le bloc d'actions, sinon la boucle sera infinie.

- **Organigramme correspondant à la boucle Repeter**

  La représentation graphique de la boucle Repeter se fait selon l'organigramme suivant :

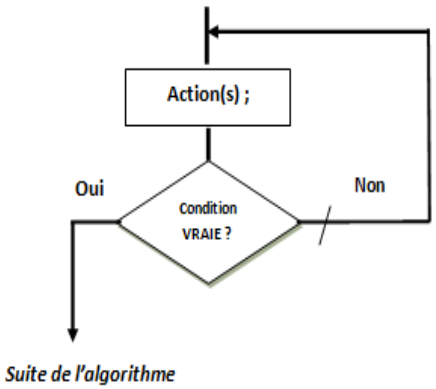


FIGURE 4.5 – Organigramme de la boucle Repeter.

- **Remarque Importante**

  Afin de choisir la boucle la plus adaptative pour résoudre un problème, on applique la règle suivante :
  Si le nombre d'itérations est connu a priori, alors on utilise la boucle *Pour*.Sinon, on utilise la boucle *Répéter* : quand il y a toujours au moins une itération, ou la boucle *Tantque* : quand le nombre d'itérations peut être nul.
  Il y a toujours au moins une exécution du corps. La boucle Répéter permet de répéter un traitement une ou plusieurs fois.
  En d'autres termes, Pour choisir entre Répéter et Tant que il faut se poser la question : faut-il éventuellement ne jamais faire le traitement ? Si oui : il faut utiliser tant que, sinon utiliser la structure Répéter qui exécute au moins une fois l'action.

- **Exemple**

  Ecrire un algorithme permettant de lire un entier positif N et d'afficher son carré.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Boucle_Repeter ;	#include<stdio.h>
Var N, C :**entier** ;	int main()
Debut	{
Repeter	int N, C ;
Lire(N) ;	**do**
Jusqu'à (N > 0)	{ scanf("%d",&N) ;
C ⟵ N*N ;	} **while** (N =< 0)
Ecrire ("le carré de", N, "est", C)	C= N*N ;
Fin.	printf("le carré de %d C %d est %d ", C) ;
	return 0 ;
	}

TABLE 4.6 – Solution algorithmique /programme C (Exemple_Boucle Repeter).

- **Solution_Organigramme Correspondant**

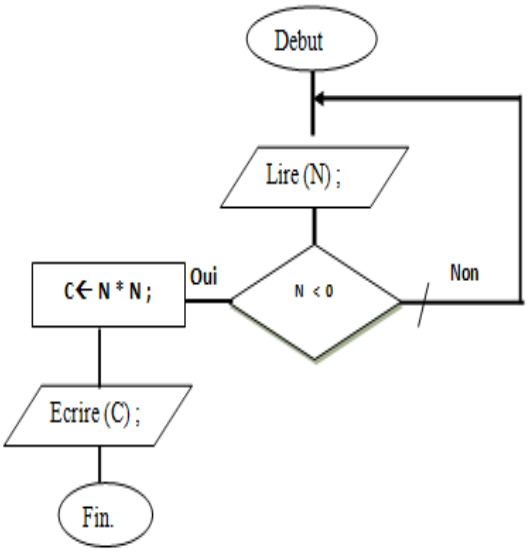


FIGURE 4.6 – Organigramme de la Boucle Repeter (solution).

4.5 Boucles imbriquées

- **Définition**

  On dit que deux boucles sont imbriquées si l'une est contenue dans le bloc d'instructions de l'autre. Supposons que la première boucle s'exécute en n itérations et la boucle interne s'exécute en m itérations, le nombre total d'itération des deux boucles est $n \times m$.

- **Exemple**

  Ecrire un algorithme qui détermine la température moyenne du mois de Mars à Bejaia sachant que les températures dans ce mois ne descendent jamais au dessous de 10° et ne montent jamais au dessus de 30°.

- **Solution algorithme / programme C**

En langage algorithmique	**En langage C**
Algorithme Boucle_Imbriq ;	#include<stdio.h>
Var T, i :**entier** ;	int main()
Debut	{
TM ⟵ 0 ;	int T, i ;
Pour i allant de 1 à 31 **Faire**	TM=0 ;
Repeter	**for**(i=1 ;i<=31 ;i++) {
Lire(T) ;	**do** {
Jusqu'à (T > 11 et T < 31)	scanf("%d",&T) ; }
TM ⟵ TM + T ;	} **while** (T < 10 and T > 30)
Finpour;	TM= TM + T ; }
Ecrire ('la température moyenne est', TM/31)	TM= TM/31 ;
Fin.	printf("la température moyenne est %d ", TM) ;
	return 0 ;
	}

TABLE 4.7 – Solution algorithmique /programme C (Exemple_Boucles Imbriquées).

- **Solution_Organigramme Correspondant**

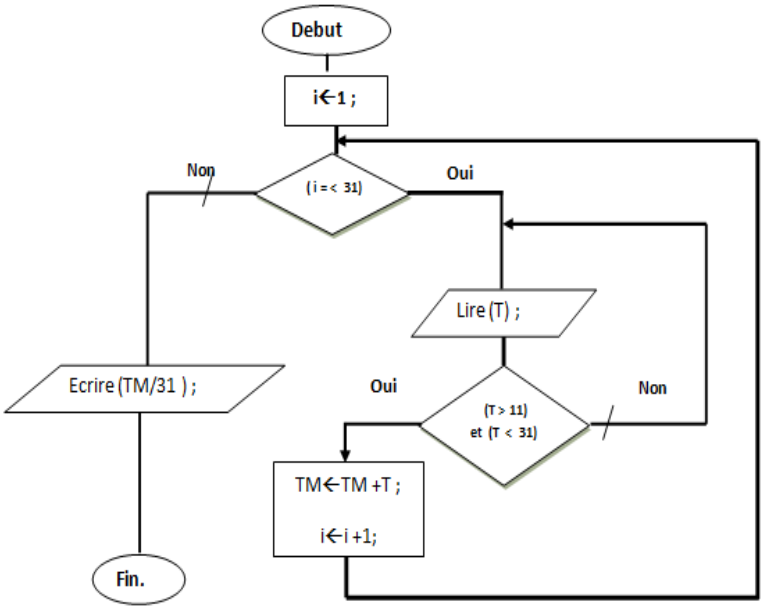


FIGURE 4.7 – Organigramme_Boucles_imbriquées (solution).

4.6 Exercices récapitulatifs avec solutions

Exercice 1 :

Ecrire un algorithme qui affiche tous les diviseurs d'un entier N donné par l'utilisateur.

Exercice 2 :

Ecrire un algorithme qui détermine le chiffre maximum dans un entier N donné par l'utilisateur (Exp. le chiffre maximum dans 2376810 est 8).

Exercice 3 :

Soit N un entier positif composé de 3 chiffres ou plus. Ecrire un algorithme qui calcule la factorielle de N (Exp. Factorielle de 5 = 5 ! = 5*4*3*2*1).

Exercice 4 :

Ecrire un algorithme permettant de calculer la somme de tous les nombres premiers qui appartiennent à l'intervalle [15, 500].

Exercice 5 :

Ecrire un algorithme qui calcule la formule F suivante :
$F = -1/2! + 2/3! - 3/4! + 4/5! - ...n - 1/n!$.

Solution_Exo1 :

```
Algorithme Exo1 ;
Var N, i :entier ;
Debut
|    Lire(N) ;
|    Pour i allant de 2 à (N div 2) Faire
|        | Si (N mod i = 0) Alors
|        |    | Ecrire (i) ;
|        | Finsi;
|    Finpour;
Fin.
```

Solution_Exo2 :

```
Algorithme Exo2 ;
Var N, CM :entier ;
Debut
|    Lire(N) ;
|    CM ⟵ 0 ;
|    Tantque (N < > 0) Faire
|        | Si (N mod 10 > CM) Alors
|        |    | CM⟵N mod 10 ;
|        | Finsi;
|        | N⟵N div 10 ;
|    FinTantque;
|    Ecrire('le chiffre max est :', CM) ;
Fin.
```

Solution_Exo3 :

```
Algorithme Exo3 ;
Var N, F, i :entier ;
Debut
|    Repeter
|        | Lire(N) ;
|    Jusqu'à N>99 ;
|    F ⟵ 1 ;
|    Pour i allant de 2 à N Faire
|        | F ⟵ F*i ;
|    Finpour;
|    Ecrire("la factorielle de", N, "est", F) ;
Fin.
```

Solution_Exo4 :

```
Algorithme Exo4 ;
Var N, i :entier ; Bool :booleen ;
Debut
|    Pour i allant de 15 à 500 Faire
|    |    Bool⟵ vrai ;
|    |    i ⟵ 2 ;
|    |    TQ ((i= < (Ndiv2)) et Bool) Faire
|    |    |    Si (N mod i = 0) Alors
|    |    |    |    Bool⟵ faux ;
|    |    |    Finsi;
|    |    |    i ⟵ i + 1 ;
|    |    FinTQ
|    |    Si ( Bool) Alors
|    |    |    Ecrire (N, "est premier") ;
|    |    Sinon
|    |    |    Ecrire (N, "n' est premier") ;
|    |    Finsi
|    Finpour;
Fin.
```

Solution_Exo5 :

```
Algorithme Exo5 ;
Var N, i :entier ;
Debut
|    Repeter
|       | Lire(N) ;
|    Jusqu'à N>1 ;
|    P ⟵ 1 ;
|    F ⟵ 0 ;
|    signe ⟵ -1 ;
|    Pour i allant de 2 à N Faire
|       | P ⟵ P*i ;
|       | X ⟵ (i-1)*signe ;
|       | signe ⟵ signe*signe ;
|       | F ⟵ F+ X/P ;
|    Finpour;
|    Ecrire(F) ;
Fin.
```

Chapitre 5

Tableaux et chaînes de caractères

Sommaire

Objectif du chapitre

- Comprendre l'utilité d'une structure de données statiques,
- Connaître les différentes structures de données statiques disponibles en algorithmique (tableaux et matrices),
- Comprendre comment manipuler une structure de données statiques,
- Comprendre l'utilité des chaînes de caractères et comment les manipuler,
- Résoudre des problèmes nécessitant l'utilisation des structures de données statiques,
- Résoudre des problèmes nécessitant l'utilisation des chaines de caractères et les manipuler via des fonctions et des procédures prédéfinies.

5.1 Introduction

Tous les types étudiés jusqu'à présent sont des types simples (voir le chapitre 2). Une variable de type simple peut prendre une seule valeur, qui correspond à une seule information : un entier, un réel ou un caractère, etc.
Par opposition aux types simples, un type structuré correspond à une collection de plusieurs valeurs. Il existe plusieurs types structurés : tableaux, enregistrements, listes, etc.
Dans ce chapitre, nous nous somme intéressés par les tableaux. En effet, un tableau est une structure de données permettant de regrouper sous un même nom plusieurs valeurs de même type. Il y a deux types de tableaux :

- les tableaux à une dimension appelés également Vecteurs,
- les tableaux à deux dimensions appelés également Matrices.

5.2 Tableaux à une dimension

Un tableau à une dimension est une structure de données statique, composé d'un nombre fixe d'éléments, caractérisé par :

- *Son nom* : un identificateur unique pour tous les éléments,
- *Sa taille* : le nombre d'éléments (cases du tableau) qui est constant et doit être connu à l'avance,
- *Son indice* : l'accès à un élément du tableau se fait à l'aide d'un indice (Numéro de case).

A chaque fois qu'on doit *designer un élément* du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément entre deux crochets.
Le schéma ci-dessous représente un tableau T à une dimension, de 10 éléments.

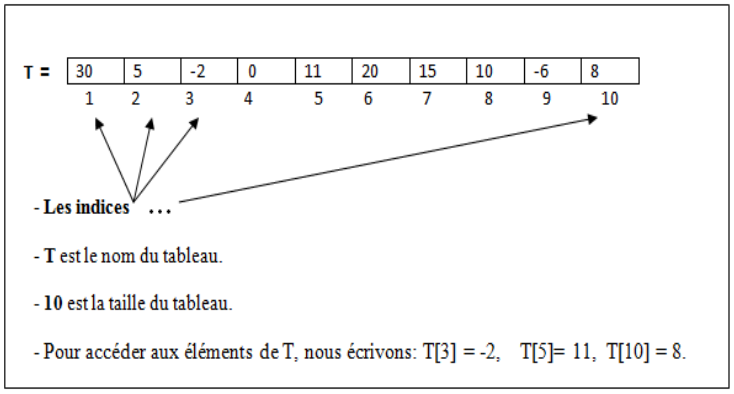


FIGURE 5.1 – Schéma représentant un tableau T de 10 éléments.

- **Remarque**

  Selon les langages de programmation, les tableaux commencent :
  – à l'indice 1 (pseudolangages, Pascal),
  – à l'indice 0 (en C et C++).

5.2.1 Déclaration d'un tableau à une dimension

Un tableau se déclare dans la partie déclaration (*Var*), en précisant le nombre et le type de valeurs qu'il contiendra. Sinon, on définit un nouveau type tableau dans la partie déclaration (*Type*), en précisant le nombre et le type de valeurs qu'il contiendra, puis on déclare une variable tableau dans la partie déclaration (*Var*) de ce nouveau type qu'on vient de définir. Pour déclarer un tableau en algorithmique / en langage C, nous utilisons la syntaxe suivante :

En langage algorithmique	**En langage C**
Var Nom_tab : **Tableau** [Début.. Taille_tab] de Type_Elt ; Ou bien : **Type** Type_tab : **Tableau** [Début.. Taille_tab] de Type_Elt ; **Var** Nom_tab : Type_tab ;	Type_Elt Nom_tab [Taille_tab] ;

TABLE 5.1 – Déclaration d'un tableau à une dimension.

– *Type_tab* : un identificateur choisi par le programmeur indiquant le type Tableau.
– *Nom_tab* : un identificateur choisi par le programmeur indiquant le nom du tableau.
– *Début* et *Taille_tab* : constantes (entiers, caractères, etc), Début est généralement égal à 1 et Taille_tab représente la taille maximale du tableau.
– *Type_Elt* : le type des valeurs du tableau (elles ont toutes le même type) réel, entier, caractère,

- **Exemple**

  On veut déclarer un tableau de Moyennes pour 50 étudiants.

En langage algorithmique	**En langage C**
Var Moy : Tableau [1.. 50] de **reel** ; Ou bien : **Type** Tab_Moy : Tableau [1.. 50] de **reel** ; **Var** Moy : Tab_Moy ;	float Moy [49] ;

TABLE 5.2 – Exemple sur la déclaration d'un tableau à une dimension

5.2.2 Opérations de base sur un tableau

Comme on l'avait mentionné au début de cette section (voir la figure 5.1), l'accès aux éléments d'un tableau se fait en donnant le nom de la variable tableau suivi entre deux crochets ([ ]) de l'indice de l'élément.

- **Exemple**

  Type Tab_Moy = tableau [1..50] de **reel** ;
  Var Moy : Tab_Moy ;
  – Moy [1] : désigne le premier élément du tableau Moy.
  – Moy [50] : désigne le dernier élément du tableau Moy.
  – Moy [51] : n'est pas un élément du tableau Moy.

Les trois opérations de base utilisées pour accéder aux éléments d'un tableau T sont :

1. **Accès en lecture**
   L'accès en lecture d'un tableau T (Remplir T / initialiser T) consiste à entrer des valeurs à partir du clavier et les mémoriser dans les cases de T. La syntaxe permettant de lire un élément de T, qui se trouve à la position i est donnée par : Quand on écrit *Lire (T[i])* :

En langage algorithmique	**En langage C**
Lire (T[i]) ;	scanf ("%d",&T[i]) ;

TABLE 5.3 – La lecture d'un élément dans un tableau à une dimension.

   la valeur entrée par l'utilisateur est enregistrée dans le tableau à la i^{me} case.
   Le fait que les éléments d'un tableau sont indicés, cela nous permet de les parcourir avec une boucle. La boucle pour est généralement la boucle la plus adaptée afin de parcourir un tableau. Car, on connaît le nombre de fois qu'on doit effectuer le traitement (une fois par élément).

2. **Accès en écriture**
   L'accès en écriture au tableau T (Affichage du T) consiste à afficher les valeurs stockées dans les cases de T. La syntaxe permettant d'afficher le contenu d'un élément de T, qui se trouve à la position i est donnée par : Quand on écrit Ecrire (T[i]), la valeur stockée

En langage algorithmique	**En langage C**
Ecrire (T[i]) ;	printf ("%d",&T[i])

TABLE 5.4 – L'affichage d'un élément dans un tableau à une dimension.

   dans le tableau, à la i^{me} case, sera affichée sur l'écran.
   De façon analogue à la lecture, l'affichage de tous les éléments d'un tableau T se fait dans une boucle (généralement la boucle *Pour*).

- **Exemple**

  Pour remplir un tableau T de 50 réels (voir l'exemple précédent), nous procédons de cette façon (voir la table 5.5 :

En langage algorithmique	**En langage C**
Algorithme Lecture_tab ;	#include<stdio.h>
Type Tab_Moy : Tableau [1.. 50] de **reel** ;	int main()
Var Moy : Tab_Moy ; i :**entier** ;	{ int i ; float Moy [49] ;
Debut	for(i=0 ;i<=49 ;i++)
Pour i allant de 1 à 50 **Faire**	scanf ("%d",&Moy[i])
Lire (Moy[i]) ;	}
Finpour;	
Fin.	

TABLE 5.5 – Exemple sur la lecture de tous les éléments d'un tableau à une dimension.

- **Exemple**

  Ecrire un algorithme permettant d'afficher les éléments du tableau T de l'exemple précédent.

En langage algorithmique	**En langage C**
Algorithme Affichage_tab ;	#include<stdio.h>
Type Tab_Moy : Tableau [1.. 50] de **reel** ;	int main()
Var Moy : Tab_Moy ; i :**entier** ;	{ int i ; float Moy [49] ;
Debut	for(i=0 ;i<=49 ;i++)
Pour i allant de 1 à 50 **Faire**	printf ("%d",Moy[i])
Ecrire (Moy[i]) ;	}
Finpour;	
Fin.	

TABLE 5.6 – Exemple sur l'affichage de tous les éléments d'un tableau à une dimension.

3. **Accès par affectation**

   L'affectation d'une nouvelle valeur à un élément, se trouvant à la i^{me} case dans un tableau T, consiste à placer cette valeur dans T à l'indice i (en écrasant l'ancienne valeur qui a été stockée dans cette case). La syntaxe permettant de modifier le contenu d'un élément de T, qui se trouve à la position i par le contenu d'une valeur Val est comme suit :

En langage algorithmique	**En langage C**
T[i]⟵ Val ;	T[i] := Val ;

TABLE 5.7 – L'affictation d'une valeur à un seul élément dans un tableau à une dimension.

- **Exemple1**

  Ecrire un algorithme permettant doubler la valeur de l'élément qui se trouve au milieu de T (T est le tableau de l'exemple précédent).

En langage algorithmique	**En langage C**
Algorithme Affectation_seul ;	#include<stdio.h>
Type Tab_Moy : Tableau [1.. 50] de **reel** ;	int main()
Var Moy : Tab_Moy ; i :**entier** ;	{ int i ; float Moy [49] ;
Debut	Moy[24] := Moy[24] *2 ;
Moy[25]⟵ Moy[25] *2 ;	}
Fin.	

TABLE 5.8 – Exemple sur l'affictation d'une valeur à un seul élément dans un tableau à une dimension.

- **Exemple2**

  Ecrire un algorithme permettant de remplacer chaque valeur paire dans T par zéro (T est le tableau de l'exemple précédent).

En langage algorithmique	**En langage C**
Algorithme Affectation_zero ;	#include<stdio.h>
Type Tab_Moy : Tableau [1.. 50] de **reel** ;	int main()
Var Moy : Tab_Moy ; i :**entier** ;	{ int i ; float Moy [49] ;
Debut	for(i=0 ;i<=49 ;i++)
Pour i allant de 1 à 50 **Faire**	if (Moy[i] % 2 == 0)
Si (Moy[i] mod 2 = 0) **Alors**	Moy[i] :=0 ;
Moy[i]⟵0 ;	}
Finsi;	
Finpour;	
Fin.	

TABLE 5.9 – Exemple sur l'affectation d'un zéro à toute valeur paire dans un tableau à une dimension.

5.3 Tableaux à deux dimensions (Matrices)

Un tableau à deux dimensions appelé également matrice permet d'organiser des informations de même type en lignes et en colonnes. Il est caractérisé donc par :

- *Son nom* : un identificateur unique pour tous les éléments.
- *Son nombre de lignes et son nombre de colonnes* qui doivent être connus à l'avance.
- *Ses deux indices* : l'accès à un élément de la matrice se fait à l'aide de deux indices (le premier représente le numéro de ligne et le deuxième représente le numéro de colonnes).

A chaque fois qu'on doit *designer un élément* de la matrice, on fait figurer le nom de cette matrice, suivi par les deux indices de l'élément entre deux crochets. Les deux indices doivent être séparés par une virgule. Le schéma ci-dessous représente une matrice M avec 3 lignes et 4 colonnes :

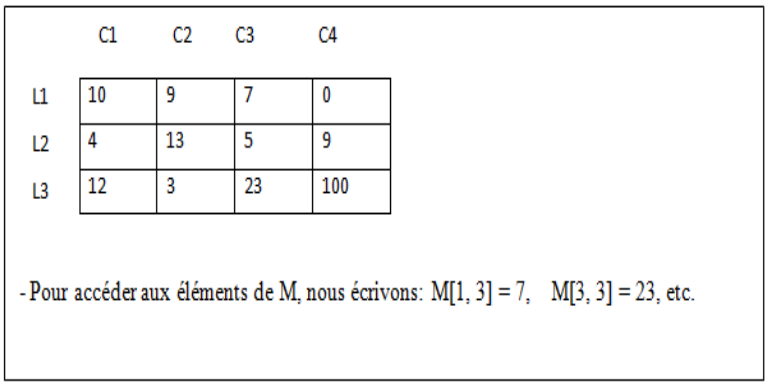


FIGURE 5.2 – Schéma représentant une matrice M de 3 lignes et de 4 colonnes.

5.3.1 Déclaration d'une Matrice

De même qu'un tableau à une seule dimension, la syntaxe de déclaration d'une variable de type matrice peut être l'une des deux suivantes :

En langage algorithmique	**En langage C**
Var Nom_Mat : Tableau [1..nbL, 1..nbC] de Type_Elt Ou bien : **Type** Type_Mat : Tableau [1..nbL, 1..nbC] de Type_Elt ; **Var** Nom_Mat : Type_Mat ;	Type_Elt Nom_Mat [nbL] [nbC] ;

TABLE 5.10 – Déclaration d'un tableau à deux dimensions.

- *Type_ Mat* : un identificateur choisi par le programmeur indiquant le type Matrice.
- *Type_ Mat* : un identificateur choisi par le programmeur indiquant le type Matrice.
- *nbL et nbC* : constantes (généralement des entiers) qui représentent le nombre de lignes et le nombre de colonnes.
- *Type_ Elt* : le type des valeurs de la matrice (elles ont toutes le même type) réel, entier, etc.

- **Exemple**

  On veut déclarer la matrice M de la figure 5.2.

En langage algorithmique	**En langage C**
Var M : Tableau [1.. 3, 1..4] de **entier** ; Ou bien : **Type** Mat : Tableau [1.. 3, 1..4] de **entier** ; **Var** M : Mat ;	int M [3] [4] ;

TABLE 5.11 – Exemple sur la déclaration d'un tableau à une dimension

5.3.2 Opérations de base sur une Matrice

Comme on l'avait mentionné précédemment, un élément de la matrice est identifié par son numéro de ligne et son numéro de colonne.
Si M est une matrice, M [i, j] désigne l'élément de M situé à la i^{me}ligne et j^{me} colonne. Pour y accéder, nous utilisons les trois opérations suivantes :

- Accès en lecture,
- Accès en écriture,
- Accès par affectation.

1. **Accès en lecture**

   L'accès en lecture d'une matrice *M* (Remplir M / Initialiser M) consiste à entrer des valeurs à partir du clavier et les mémoriser dans les cases de *M*. La syntaxe permettant de lire un élément de *M*, qui se trouve à la ligne *i* et à la colonne *j*, est donnée par : Quand on écrit *Lire (M[i,j])* : la valeur entrée par l'utilisateur est enregistrée dans la

En langage algorithmique	**En langage C**
Lire (M[i,j]) ;	scanf ("%d",&M[i][j]) ;

TABLE 5.12 – La lecture d'un élément dans une matrice.

matrice M à la case de la i^{me}ligne et de la j^{me} colonne.
Comme les tableaux à une dimension, *le parcourt d'une matrice* se fait avec la boucle *Pour*. En effet, nous avons besoin de deux boucle Pour imbriquées. La première boucle est utilisée pour parcourir les lignes et la deuxième pour parcourir les colonnes.

- **Exemple**

  Pour remplir une matrice d'entiers M de 100 lignes et 70, nous procédons de cette façon :

En langage algorithmique	**En langage C**
Algorithme Lecture_Matrice ;	#include<stdio.h>
Type Mat_Moy : Tableau [1.. 100, 1..70] de **entier** ;	int main()
Var M : Mat ; i,j :**entier** ;	{ int i, j ; int M [100, 70] ;
Debut	for(i=0 ;i<=99 ;i++)
Pour i allant de 1 à 100 **Faire**	for(j=0 ;j<=69 ;j++)
Pour i allant de 1 à 70 **Faire**	scanf ("%d",&M [i][j])
Lire (M[i, j][i]) ;	}
Finpour;	
Finpour;	
Fin.	

TABLE 5.13 – Exemple sur la lecture de tous les éléments d'une matrice.

2. **Accès en écriture**

L'accès en écriture à une matrice M (Affichage de M) consiste à afficher les valeurs stockées dans les cases de M. La syntaxe permettant d'afficher le contenu d'un élément de M, qui se trouve à la ligne i et à la colonne j, est donnée par : Quand on écrit Ecrire

En langage algorithmique	**En langage C**
Ecrire (M[i,j]) ;	printf ("%d",M[i] [j])

TABLE 5.14 – L'affichage d'un élément dans une matrice.

(M[i,j]), la valeur stockée dans la matrice, au niveau de la case qui se trouve à la ligne i et à la colonne j, sera affichée sur l'écran.
De façon analogue à la lecture, l'affichage de tous les éléments d'une matrice M se fait dans deux boucles *Pour imbriquées.*

- **Exemple**

  Ecrire un algorithme permettant d'afficher les éléments de la matrice M de l'exemple précédent.

En langage algorithmique	**En langage C**
Algorithme Affichage_Matrice ;	#include<stdio.h>
Type Mat_Moy : Tableau [1.. 100, 1..70] de **entier** ;	int main()
Var M : Mat ; i,j :**entier** ;	{ int i, j ; int M [100, 70] ;
Debut	for(i=0 ;i<=99 ;i++)
Pour i allant de 1 à 100 **Faire**	for(j=0 ;j<=69 ;j++)
Pour i allant de 1 à 70 **Faire**	printf ("%d",M [i][j])
Ecrire (M[i, j][i]) ;	}
Finpour;	
Finpour;	
Fin.	

TABLE 5.15 – Exemple sur l'affichage de tous les éléments d'une matrice.

3. **Accès par affectation**

L'affectation d'une nouvelle valeur à un élément, se trouvant à la i^{me}ligne et de la j^{me} colonne dans une matrice M, consiste à placer cette valeur dans M à la même. La syntaxe permettant de modifier le contenu d'un élément de M, qui se trouve à la ligne numéro i et à la colonne numéro j par le contenu d'une valeur *Val* est comme suit :

En langage algorithmique	**En langage C**
M[i, j]⟵ Val ;	M[i][j] := Val ;

TABLE 5.16 – L'affictation d'une valeur à un seul élément dans une matrice.

- **Exemple1**

  Ecrire un algorithme permettant de remplacer le contenu de la première case de M ainsi que celui de la dernière case de M par un zéro (M est la même matrice sur laquelle on travaille).

En langage algorithmique	**En langage C**
Algorithme Afffectation_element_Matrice ;	#include<stdio.h>
Type Mat_Moy : Tableau [1.. 100, 1..70] de **entier** ;	int main()
Var M : Mat ; i,j :**entier** ;	{ int i, j, M [100, 70] ;
Debut	M[0][0] :=0 ;
M[1, 1]⟵ 0 ;	M[99][69] :=0 ;
M[100, 70]⟵ 0 ;	}
Fin.	

TABLE 5.17 – Exemple sur l'affictation d'un zéro à deux éléments dans une matrice.

- **Exemple2**

  Ecrire un algorithme permettant de remplacer toutes les valeurs de M se trouvant dans des colonnes paires par un zéro.

En langage algorithmique	**En langage C**
Algorithme Affectation_Matrice ;	#include<stdio.h>
Type Mat_Moy : Tableau [1.. 100, 1..70] de **entier** ;	int main()
Var M : Mat ; i,j :**entier** ;	{ int i, j, M [100, 70] ;
Debut	for(i=0 ;i<=99 ;i++)
Pour i allant de 1 à 100 **Faire**	for(j=0 ;j<=69 ;j++)
Pour j allant de 1 à 70 **Faire**	if (j % 2 == 0){
Si (j mod 2 = 0) **Alors**	M[i][j] :=0 ;}
M[i,j]⟵0 ;	}
Finsi;	
Finpour;	
Finpour;	
Fin.	

TABLE 5.18 – Exemple sur l'affectation d'un zéro à toute valeur se trouvant sur une colonne paire dans une matrice.

5.4 Chaînes de caractères

Les chaînes de caractère sont des séquences de caractères délimitées par des guillemets (i.e. ", par exemple "algorithmique").
La taille d'une chaîne de caractère varie entre *0 et 255*. Si *la taille = 0* on dit que la chaîne est vide.
Dans la mémoire centrale, une chaine de caractères est représentée sous forme d'un tableau. La taille de ce dernier est égale au nombre de caractères composant cette chaîne de caractères. A chaque fois qu'on doit designer un caractère de la chaîne, on fait figurer le nom de cette chaine de caractères, suivi du numéro du caractère en question (sa position dans la chaine de caractères) entre deux crochets (voir la figure 5.3).

- **Exemple**

  Soit CH la chaine de caractères "algorithmique". Sa représentation en mémoire centrale est la suivante :

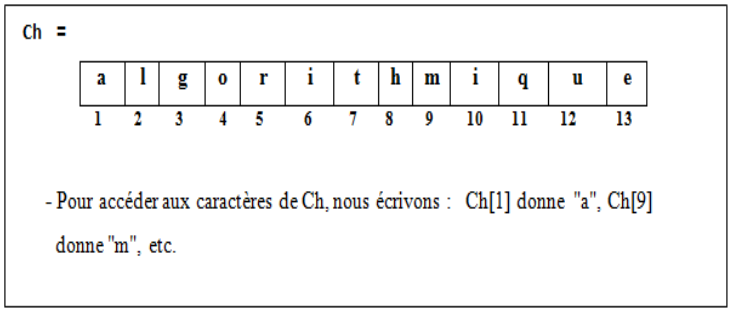


FIGURE 5.3 – Schéma représentant la chaîne de caractères Ch="algorithmique".

5.4.1 Déclaration d'une chaîne de caractères

Une variable chaîne de caractères se déclare en utilisant la syntaxe suivante :

En langage algorithmique	**En langage C**
Var Nom_chaine : **chaine** [taille] ;	char <Nom_chaine> [taille] ;

TABLE 5.19 – Déclaration d'une chaîne de caractères.

taille : indique la taille maximale de la chaîne déclarée. Si cette taille n'est pas précisée, une taille par défaut de 255 caractères est appliquée.

- **Exemple**

  On veut déclarer une chaine de caractères CH de 20 caractères.

En langage algorithmique	**En langage C**
Var CH : **chaine** [20] ;	char <CH> [20] ;

TABLE 5.20 – Exemple sur la déclaration d'une chaîne de caractères.

5.4.2 Opérations de base sur les chaînes de caractères

Dans cette sous-section, nous présentons les différentes opérations de base appliquées sur les chaînes de caractères et qui sont :

- Lecture / Ecriture d'une chaîne de caractères,
- Accès à un caractère / chaîne de caractère par affectation,
- Opérations relationnelles sur les chaînes de caractères,
- Concaténation des chaînes de caractères (cette opération a été étudiée dans le chapitre 2, §2.5.2.)

1. **Lecture / Ecriture d'une chaîne de caractères**
   La lecture et l'écriture d'une variable chaîne de caractères est compatible à celle d'une variable simple et non pas élément par élément comme une variable de type tableau.
   La syntaxe permettant de lire et d'afficher le contenu d'une chaine de caractères, en algorithmique / en langage C, est le suivant :

En langage algorithmique	**En langage C**
Lire (CH) ;	scanf("%s", CH) ;
Ecrire (CH) ;	printf("%s", CH) ;

TABLE 5.21 – Lecture / Ecriture d'une chaîne de caractères.

2. **Accès à un caractère / chaîne de caractère par affectation**
   Le remplacement (ou l'initialisation) d'un caractère se trouvant à la position i dans une chaine de caractères *CH* par un nouveau caractère C se réalise via l'affectation de ce nouveau caractère à la position i (en écrasant l'ancien caractère de la même position). Cette opération se fait suivant la syntaxe ci-dessous :

En langage algorithmique	**En langage C**
CH[i]⟵C ;	CH [i] :=C ;

TABLE 5.22 – Accès à un caractère par affectation.

Quand on remplace le contenu d'une chaine *CH* par un nouveau contenu (celui d'une autre chaine *CH1*), nous affectons directement ce nouveau contenu à la chaine sans utiliser une boucle comme le montre la syntaxe suivante :

En langage algorithmique	**En langage C**
CH⟵CH1 ;	CH :=CH1 ;

TABLE 5.23 – Accès à une chaîne de caractères par affectation.

- **Exemple**
  Soient CH et CH1 deux chaines de caractères.
  CH⟵'Algo' ; C'est l'initialisation de CH par la valeur 'Algo',
  CH1⟵'Math' ; C'est l'initialisation de CH1 par la valeur 'Math',
  CH[2]⟵'M' ; Ici, CH devient 'aMgo',
  CH⟵CH1 ; Ici, CH devient 'Math'.

3. **Opérations relationnelles sur les chaînes de caractères**
   Nous avons étudié les opérateurs relationnels dans le chapitre 2 (voir §2.4.2.3), ces opérateurs permettent la comparaison de deux chaines de caractères en comparant leurs caractères caractère par caractère de la gauche vers la droite selon leur code ASCII (*American Standard Code for Information Interchange :* voir le tableau ci-dessous).

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

FIGURE 5.4 – Numérotation des principaux caractères utilisés en algorithmique.

- **Exemple**

  – ’Programme’ > ’Program’, c’est Vrai.
  – ’a’ < ’A’, c’est faux.
  – ” = ’ ’, c’est faux (le vide est différent du caractère espace).

5.4.3 Fonctions /Procédures prédéfinies sur les chaînes de caractères

Dans cette sous-sections, nous allons présenter quelques fonctions/ procédures prédéfinies et plus utilisées dans les chaines de caractères.
Les fonctions / procédures prédéfinies sont des sous-programmes (cette notion sera étudiée en semestre II), qui réalisent généralement des opérations très fréquemment utilisées pour faciliter la tâche de programmation (par exemple, lire ( ), écrire, etc).
Voici quelques fonctions / procédures prédéfinies en algorithmique (voir la table 5.24).
En langage C, on ne trouve pas les mêmes sous programmes présentés ci-dessus. C’est pour cette raison, nous avons utilisé deux tables séparées :
La première table présente quelques sous programmes prédéfinis et plus connus dans le langage algorithme (Table 5.24).
La deuxième table présente quelques sous programmes prédéfinis et plus connus dans le langage C (Table 5.25).

Fonction / procédure	**Rôle**	**Exemple**
L⟵long(CH) ;	Retourne la longueur L d'une Chaîne de caractères CH.	Si CH = "informatique", aprèss l'exécution de L⟵long(CH), L vaut 12.
CH2⟵ copie(CH1,P,N) ;	Copie N caractères de la chaîne CH1 à partir de la position P et les stocke dans CH2.	Si CH="informatique", N=3, et P = 5. Après l'exécution de CH2⟵copie(CH1,P,N), CH de vient "forma".
P⟵ position(CH1,CH2) ;	Retourne la position de la première occurrence de la chaîne CH1 dans la chaîne CH2. Si CH1 ne contient pas CH2, cette fonction retourne zéro (P=0).	Si CH1="informatique" et CH2 ="mati". Après l'exécution de P⟵position(CH1,CH2), P vaut 6.
Efface(CH,P,N) ;	Efface N caractères de la chaîne CH à partir de la position P.	Si CH1="informatique". Après l'exécution de Efface(CH,2,4), CH devient "imatique".
Insert(CH1,CH2,P) ;	Insère la chaîne CH1 dans la CH2 à partir de la position P.	Si CH1="2022 et CH2 ="informatique. Après l'exécution de Insert(CH1,CH2,13), CH2 devient "informatique2022".

TABLE 5.24 – Quelques fonctions / procédures prédéfinies en algorithmique.

Fonction	**Rôle**	**Exemple**
Strlen (CH) ;	Retourne la longueur d'une Chaîne de caractères CH.	char CH1[20]= "informatique" ; Le résultat à obtenir après l'exécution de Strlen (CH) est 12.
Strepy(CH1,CH2) ;	Ajoute le contenu de la chaine CH1 au contenu de la chaine CH2.	char CH1[20]= "informatique" ; char CH2[20]="2022" ; Le résultat à obtenir après l'exécution de Strepy(CH1,CH2) est "informatique2022".
Strcmp (CH1, CH2) ;	Compare deux chaînes CH1 et CH2 par rapport à leur taille. Elle retourne : *Une valeur positive si la taille de CH1 est plus longue que celle de CH2. * Zéro si elles sont égales * Une valeur négative si la taille de CH1 est plus petite que celle de CH2.	char CH1[20]= "informatique" ; char CH2[20]="2022" ; Le résultat à obtenir après l'exécution de Strcmp (CH1, CH2) est 1.

TABLE 5.25 – Quelques fonctions prédéfinies en langage C.

5.5 Exercices récapitulatifs avec solutions

Exercice 1 :

Soit T un tableau de N éléments entiers. Ecrire un algorithme qui affiche le nombre des occurrences d'un entier V donné par l'utilisateur

Exercice 2 :

Soit T un tableau de N éléments entiers. Ecrire un algorithme qui permet de permuter les éléments de T de cette façon : le dernier élément devient le premier et le premier

élément devient le dernier, l'avant dernier élément devient le deuxième et le deuxième devient l'avant dernier, etc.

- Exemple

T en entrée	5	10	0	-2	13	100	-13

T en sortie	-13	100	13	-2	0	10	5

Exercice 3 :

Soit T un tableau de N éléments entiers. Ecrire un algorithme qui permet de déterminer l'élément maximal dans T.

Exercice 4 :

Soit T un tableau de N éléments entiers. Ecrire un algorithme qui permet de trier les éléments de T dans l'ordre croissant.

Exercice 5 :

Ecrire un algorithme qui lit une matrice de n lignes et m colonnes puis cherche si une valeur entière X lue à partir du clavier existe dans la matrice et afficher sa position.

Exercice 6 :

Ecrire un algorithme qui lit une matrice de n lignes et m colonnes puis compte le nombre d'éléments négatifs, positifs et nuls dans cette matrice.

Exercice 7 :

Soit A et B deux matrices carrées de n lignes et n colonnes.

Ecrire un algorithme qui lit les deux matrices A et B puis calcule le produit de ces deux matrices.

Exercice 8 :

Ecrire un algorithme qui lit une chaine de caractères CH, puis calcule et affiche le nombre de mots dans CH.

Exercice 9 :

Ecrire une fonction qui reçoit comme paramètre une chaîne de caractères et écrit dans un tableau la longueur de chaque mot de cette chaîne.

- Exemple

  Si la chaîne est "Une chaîne de caractères".

  Le tableau des longueurs de mots contiendra les valeurs suivantes :

3	6	2	10

Exercice 10 :

Ecrire un algorithme qui affiche les positions du premier et du dernier caractère du plus long mot d'une chaine de caractères CH.

Solution_Exo1 :

```
Algorithme Exo1 ;
Type Tab = Tableau [1.. 100] de entier ;
Var T : Tab ; N, V, cp, i : entier ;
Debut
|    Lire (V) ;
|    cp ⟵ 0 ;
|    Repeter
|       | Lire(N) ;
|    Jusqu'à (N >= 1 et N =< 100) ;
|    Pour i allant de 1 à N Faire
|    |    Si (T[i] = V) Alors
|    |    |    cp ⟵ cp+1 ;
|    |    Finsi;
|    Finpour;
|    Ecrire(cp) ;
Fin.
```

Solution_Exo2 :

```
Algorithme Exo1 ;
Type Tab = Tableau [1.. 100] de entier ;
Var T : Tab ; N, i : entier ;
Debut
|    Repeter
|       | Lire(N) ;
|    Jusqu'à (N >= 1 et N =< 100) ;
|    i ⟵ 1 ;
|    TQ ( i = < (N div 2)) Faire
|    |    T[i] ⟵ T[n+1-i] ;
|    |    i ⟵ i+1 ;
|    FinTQ;
Fin.
```

Solution_Exo3 :

```
Algorithme Exo3 ;
Type Tab = Tableau [1.. 100] de entier ;
Var T : Tab ; N, max,pos, i : entier ;
Debut
|    Repeter
```

```
|       | Lire(N) ;
|   Jusqu'à (N >= 1 et N =< 100) ;
|   max ⟵ T[1] ; ;
|   pos ⟵ 1 ;
|   Pour i allant de 2 à N Faire
|   |   Si (T[i] > max) Alors
|   |   |   max ⟵ T[i] ;
|   |   |   pos ⟵ i ;
|   |   Finsi;
|   Finpour;
|   Ecrire("la valeur maximale dans T est", max, " et sa position est", pos) ;
Fin.
```

Solution_Exo4 :

```
Algorithme Exo4 ;
Type Tab = Tableau [1.. 100] de entier ;
Var T : Tab ; N, X, i, j : entier ;
Debut
|   Repeter
|       | Lire(N) ;
|   Jusqu'à (N >= 1 et N =< 100) ;
|   Pour i allant de 1 à (N-1) Faire
|   |   Pour j allant de i à N Faire
|   |   |   Si (T[i] > T[j]) Alors
|   |   |   |   X ⟵ T[i] ;
|   |   |   |   T[i] ⟵ T[j] ;
|   |   |   |   T[j] ⟵ X ;
|   |   |   Finsi;
|   |   Finpour;
|   Finpour;
Fin.
```

Solution_Exo5 :

```
Algorithme Exo5 ;
Type Mat = Tableau [1.. 100, 1..70] de entier ;
Var M : Mat ; n, m, X, i, j, Pi, Pj : entier ; trouve : booleen ;
Debut
|   Repeter
|       | Lire (n,m) ;
|   Jusqu'à ((n >= 1 et n =< 100 ) et (m >= 1 et m =< 70)) ;
|   trouve ⟵ faux ; i ⟵ 1 ; j ⟵ 1 ;
|   TQ ((i =< n) et trouve) Faire
```

```
|   |   TQ ((j =< m)et trouve) Faire
|   |   |   Si (X = M[i,j]) Alors
|   |   |   |   Pi ⟵ i ;
|   |   |   |   Pj ⟵ j ;
|   |   |   |   trouve ⟵ vrai ;
|   |   |   Finsi;
|   |   |   j ⟵ j+1 ;
|   |   FinTQ;
|   |   i ⟵ i+1 ;
|   FinTQ;
|   Si (trouve) Alors
|   |   Ecrire ("la valeur", X, " n'existe pas dans la matrice M") ;
|   Sinon;
|   |   Ecrire ("la valeur", X, " existe à la ligne", Pi, "et la colonne", Pj) ;
|   Finsi;
Fin.
```

Solution_Exo6 :

```
Algorithme Exo6 ;
Type Mat = Tableau [1.. 100, 1..70] de entier ;
Var M : Mat ; n, m, i, j, pos, neg, nul : entier ;
Debut
|   Repeter
|   | Lire (n,m) ;
|   Jusqu'à ((n >= 1 et n =< 100 ) et (m >= 1 et m =< 70)) ;
|   pos ⟵ 0 ; neg ⟵ 0 ; nul ⟵ 0 ;
|   Pour i allant de 1 à n Faire
|   |   Pour j allant de 1 à m Faire
|   |   |   Si M[i,j] > 0) Alors
|   |   |   |   | pos ⟵ pos+1 ;
|   |   |   | Sinon;
|   |   |   |   Si M[i,j] < 0) Alors
|   |   |   |   |   neg ⟵ neg+1 ;
|   |   |   |   | Sinon
|   |   |   |   |   nul ⟵ nul ;
|   |   |   |   Finsi ;
|   |   |   Finsi;
|   |   Finpour;
|   Finpour;
|   Ecrire (" le nombre des valeurs positives dans M est", pos) ;
|   Ecrire (" le nombre des valeurs négatives dans M est", neg) ;
```

```
|    Ecrire (" le nombre des valeurs nulles dans M est", nul) ;
Fin.
```

Solution_Exo7 :

```
Algorithme Exo7 ;
Type Mat = Tableau [1.. 100, 1..100] de entier ;
Var A, B, C : Mat ; n, i, j,k, S : entier ;
Debut
|    Repeter
|       | Lire (n) ;
|    Jusqu'à ((n >= 1 et n =< 100 )) ;
|    Pour i allant de 1 à n Faire
|    |    Pour j allant de 1 à n Faire
|    |    |    ⟵ 0 ;
|    |    |    Pour k allant de 1 à n Faire
|    |    |    |        S⟵ S+ A [i,j]*B[j,k] ;
|    |    |    Finpour ;
|    |    |    C[i, j] ⟵ S ;
|    |    Finpour;
|    Finpour;
Fin.
```

Solution_Exo8 :

```
Algorithme Exo8 ;
Var CH, CH1 : chaine ; P, cp : entier ;
Debut
|    Lire (CH) ;
|    CH1 ⟵ CH ; /* garder CH dans CH1 pour ne pas la perdre */
|    cp ⟵ 1 ; /* CH contient au moins un mot */
|    Repeter
|    | P ⟵ position(CH1, ' ') ; /* chercher des espaces dans CH*/
|    |    Si (p < > 0) Alors /* à chaque fois on trouve un espace, cela veut dire qu'on a un mot*/
|    |    |    cp ⟵ cp+1 ;
|    |    |    Efface (CH1,1, P) ; /* on efface de CH1 le mot et l'espace qui le suit*/
|    |    Finsi;
|    Jusqu'à P=0 ; /* quand on sot de cette boucle, CH1 contient seulement le dernier mot de CH*/
|    Ecrire ("le nombre de mots dans", CH, "est", cp) ;
Fin.
```

Solution_Exo9 :

```
Algorithme Exo9 ;
Type Tab = Tableau [1.. 100] de entier ;
Var T : Tab ; CH, CH1 : chaine ; P, i : entier ;
```

```
Debut
|   Lire (CH) ;
|   P ⟵ position(CH, ' ') ;
|   Si (p=0) Alors /* CH contient un seul mot qui est CH elle-même*/
|   |   T[1] ⟵ long(CH) ;
|   Sinon
|   |   CH1 ⟵ CH ; */garder CH dans CH1 pour ne pas la perdre*/
|   |   i ⟵ 0 ;
|   |   Repeter
|   |   |   P⟵ position(CH1, ' ') ;
|   |   |   i⟵ i+1 ;
|   |   |   T[i]⟵ P-1 ; */la longueur du premier mot = P -1 */
|   |   |   Efface (CH1,1, P) ; /*on efface de CH1 le mot et l'espace qui le suit*/
|   |   Jusqu'à P=0 ; /* quand on sort de cette boucle, CH1 contient seulement le dernier mot de CH*/
|   |   i ⟵ i+1 ;
|   |   T[i] ⟵ long(CH1) ;
|   Finsi ;
Fin.
```

Solution_Exo10 :

```
Algorithme Exo10 ;
Var CH, CH1, Motmax : chaine ; P, Lmax : entier ;
Debut
|   Lire (CH) ;
|   P ⟵ position(CH, ' ') ;
|   Si (P = 0 ) Alors */CH contient un seul mot qui est CH elle-même*/
|   |  | Motmax ⟵ CH ;
|   |  | Lmax ⟵ long[Motmax] ;
|   Sinon
|   |  | CH1 ⟵ CH ;
|   |  | P ⟵ position(CH1, ' ') ;
|   |  | Motmax ⟵ copie(CH1,1,P-1) ; /* Initialiser Motmax par le premier mot dans CH */
|   |  | Lmax ⟵ long[Motmax] ;
|   |  | Efface (CH1,1, P) ;
|   |  | Repeter
|   |  |   |   P ⟵ position(CH1, ' ') ;
|   |  |   |   Si (Lmax < (P-1)) Alors
|   |  |   |   |   Lmax ⟵ (P-1) ;
|   |  |   |   |   Motmax ⟵ copie(CH1,1,P-1) ;
|   |  |   |   Finsi;
|   |  |   |   Efface (CH1,1, P) ;
```

```
|       |   | Jusqu'à P=0 ; /* quand on sort de cette boucle, CH1 contient seulement le dernier mot de CH*/
|       | Si (Lmax < long (CH1)) Alors  /* comparer Lmax avec la taille du dernier mot dans CH qui est CH1*/
|       |   |     Motmax ⟵ CH ;
|       |   |     Lmax ⟵ long[Motmax] ;
|       | Finsi ;
|   Finsi ;
|   Ecrire ("le premier caractère du plus long mot", Motmax, "est", Motmax[1]) ;
|   Ecrire ("le dernier caractère du plus long mot", Motmax, "est", Motmax[Lmax]) ;
Fin.
```

Annexe A

Introduction au langage C

A.1 Introduction

Le langage de programmation est un intermédiaire entre l'homme et la machine, il permet de faire effectuer des tâches à une machine programmable en utilisant des concepts proches de la pensée humaine. Nous en citons quelques uns, tels que : Fortran, Basic, Pascal, Java et C. Le langage C fait partie de la famille des langages de programmation de haut Niveau. Il a été développé à partir de 1972, dans les Laboratoires *Bell*, en tant que langage de programmation système pour écrire un système d'exploitation.
Dancs cette annexe, nous allons présenter les composants élémentaires du langage C, utilisés au long de ce polycopié de cours (à partir du chapitre3), permettant aux étudiants de développer leurs programmes.

A.2 Notions de base liées aux langages de programmation

Cette section aborde quelques concepts généraux relatifs aux langages de programmations, pas forcément le langage C, qui sont :

- **Programme** : séquence d'instructions destinées à être exécutées par un ordinateur.
- **Instruction** : action que l'ordinateur connait et peut réaliser, elle représente une ligne dans un programme.
- **Code source** : programme exprimé dans un langage de programmation compréhensible par un être humain et ne pouvant pas être exécuté directement par l'ordinateur.
- **Code binaire** : programme exprimé en langage machine, pouvant être directement exécuté par l'ordinateur.

– **Compilation** : action de transformer un code source en code binaire, de manière à obtenir un fichier exécutable à partir d'un fichier source.

A.3 Structure d'un programme C

Un programme C est décrit par un fichier source, qui doit être écrit suivant la structure ci-dessous :

```
# include<stdio.h>
# include<stdlib.h>
int main()
{
return 0;
}
```

Le mot "**include**" en anglais signifie "inclure" en français. Ces lignes demandent d'inclure des fichiers au projet, c'est-à-dire d'ajouter des fichiers pour la compilation. Il y a 2 lignes, donc 2 fichiers inclus. Ces fichiers s'appellent **stdio.h** et **stdlib.h**. Ce sont des fichiers qui existent déjà, des fichiers sources tout prêts. On les appelle des librairies (ou aussi bibliothèques), ces fichiers contiennent du code tout prêt qui permet d'afficher du texte à l'écran.
La bibliothèque standard est constituée de plusieurs sous bibliothèques. A chaque sous bibliothèque est associé un fichier en-tête qui a pour extension **.h**.

– **stdio.h** permet de gérer les entrées-sorties,
– **ctype.h** permet d'effectuer des tests su les caractères et la conversion minuscule majuscule,
– **string.h** permet la manipulation des chaînes de caractère,
– **math.h** permet l'utilisation des fonctions mathématiques,
– **stdlib.h** permet la conversion des types, l'allocation de la mémoire.

Un programme en langage C est constitué de fonctions, il ne contient quasiment que ça. Dans la troisième ligne (de la structure ci-dessus), le mot "**main**" est le nom d'une fonction particulière qui signifie "principal". C'est avec cette fonction principale qu'un programme C commence.
Chaque fonction possède un entête composé de :

– son type,
– son nom (identificateur),
– une liste de paramètres (qui peut être vide).

Cet entête est suivi d'un bloc, délimité par deux accolades { et }. Dans ce bloc on trouve une zone de déclaration de variables (type suivi du nom de la variable). Puis la suite d'instructions que doit exécuter la fonction.
La dernière instruction est l'instruction "**return**", elle signifie la fin de la fonction et elle permet d'indiquer le résultat à renvoyer.

A.4 Composants élémentaires d'un programme C

Un programme C est une collection de constantes, de variables et de fonctions pouvant être définie dans la fonction principale main.

A.4.1 Commentaires

Les commentaires débutent par /* et se terminent par */.

- **Exemple**
  /∗ Ceci est un commentaire ∗/

A.4.2 Identificateurs et mots-clés

Comme expliqué dans le chapitre 2, le but d'un identificateur est de donner un nom à une entité du programme (variable, constante, etc).
Les identificateurs sont formés d'une suite de lettres, de chiffres et du signe souligné à condition que le premier caractère ne soit pas un chiffre. Les lettres formant les identificateurs peuvent être majuscules ou minuscules, mais doivent faire partie de l'alphabet anglais : les lettres accentuées sont interdites.
Un certain nombre de mots ne peuvent pas être utilisés comme identificateurs. On les appelle **mots-clés**. Il s'agit essentiellement des mots réservés pour le langage C.

- **Exemple**
  Quelques mots clés du langage C : **Auto, break, case, char, const, continue, default, do, double, else, float, for, goto, if, int, long, register, return, short, signed, switch, void, while**, etc.

A.4.3 Constantes et variables

Les données d'un programme sont stockées en mémoire et sont manipulées par l'intermédiaire d'**identificateurs** de variables.

- **Exemple**
  La déclaration : int N réserve de l'espace en mémoire pour un entier dont l'identificateur (nom qui permet de le manipuler) est N.

Si l'identificateur est associé à une valeur qui ne change pas au cours de l'exécution du programme, dans ce cas, on parle d'une **constante**. La définition de cette constante dans un

programme C se fait en utilisant la syntaxe suivante :
define XXXXX valeur
Où *XXXXX* est l'identificateur et valeur est la valeur associée.

- **Exemple**
  Pour définir une constante P de valeur 3.14, on écrira :
  # define P 3.14

A.4.4 Types fondamentaux

En langage C, toute variable doit donc être déclarée avant d'être utilisée. La déclaration d'une variable nécessite la réservation d'une zone mémoire pour stocker le contenu de cette variable. La taille de la zone mémoire réservée dépend du type de la variable déclarée.
Il existe trois types de base pour les variables en C :

- **char** : définit des variables de type caractère et ne stocke qu'un seul caractère à chaque foi, qui est codé sur 1 octet. Un caractère peut être une lettre, un chiffre ou tout autre élément du code ASCII.
- **int** : selon la valeur que l'on est amené à affecter à un entier et la présence ou non du signe, on peut opter pour int, short,unsigned int, etc.
  1. le type short (signed ou unsigned), codé avec 2 octets,
  2. le type int (signed ou unsigned), codé avec 2 ou 4 octets,
  3. le type long (signed ou unsigned), codé avec 4 ou 8 octets.
- **float** : selon la valeur que l'on est amené à affecter à un nombre à virgule flottante simple ou double précision, on peut opter pour float, double, long double.
  1. le type float, codé avec 4 octets,
  2. le type double, codé avec 8 octets,
  3. le type long double, codé avec 10 octets.

A.4.5 Opérateurs

En langage C, nous distinguons les opérateurs suivants :

A.4.5.1 Opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire **-** (changement de signe) ainsi que les opérateurs binaires :
+ addition,
- soustraction,
$*$ multiplication,
/ division,
% reste de la division (modulo).

- **Remarque**
  Les opérateurs unaires ont la plus forte priorité dans les calculs.

A.4.5.2 Opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation **--** s'utilisent aussi bien en suffixe **(i++)** qu'en préfixe **(++i)**. Toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i.

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

A.4.5.3 Opérateurs relationnels

Il s'agit des opérateurs d'égalité ($==$), d'inégalité (**!**=) ou de comparaison : $<$; $<=$; $>$; $>=$ (respectivement : inférieur, inférieur ou égal, supérieur, supérieur ou égal).
Ces opérateurs renvoient la valeur 0 (faux) ou la valeur 1 (vrai) suivant le résultat de l'opération (les variables booléennes n'existent pas de manière explicite en C).

A.4.5.4 Opérateurs logiques

Ces opérateurs s'appliquent à des opérandes qui ont la valeur logique FAUX s'ils valent 0 et la valeur logique VRAI sinon.
Il s'agit de la négation (!) du ET logique (&&) et du OU logique (||).

- **Exemple**
  Si A vaut 1 et B vaut 0, alors A && B vaut 0.

A.4.5.5 Opérateur d'affectation

L'affectation simple (=) affecte à la variable située à gauche du =, la valeur de l'expression située à droite.

- **Exemple**
  A=2*5+3 affecte la valeur 13 à la variable A.

L'affectation combinée (+=, -=, *=, /=, etc) est un raccourci d'écriture qui affecte à la variable de gauche le résultat de l'opération indiquée avant le signe = entre l'expression de droite et la variable elle-même.

- **Exemple**
  X+=3 est équivalent à écrire X=X+3.
- **Remarque**
  On peut affecter une valeur à une variable au moment de sa déclaration. On dit qu'on initialise la variable.
  - Exemple int nbr=2, triple=3*nbr ;
  char lettre='A' ; /* 'A' désigne le caractère A */

A.4.6 Fonctions d'entrées sorties printf et scanf

Il s'agit des fonctions de la librairie standard stdio.h utilisées avec les unités classiques d'entrées sorties, qui sont respectivement le clavier et l'écran.

A.4.6.1 Fonction d'entrées "scanf"

La fonction **scanf** permet de saisir des valeurs de variables formatées à partir du clavier. Elle est composée d'un format et des identificateurs de variables à saisir. Sa syntaxe est la suivante :
scanf("format", & Noms_Variables) ;

- Le symbole & est obligatoire devant les identificateurs Noms_Variables, car scanf attend des adresses et non des valeurs, sauf devant un identificateur de chaîne de caractères.
- La partie "format" est une chaîne de caractères, qui peut contenir du texte, des séquences d'échappement et des spécficateurs de format.
- Les spécificateurs de format commencent toujours par le symbole "%" et se terminent par un ou deux caractères qui indiquent le format de saisie.

Format	**Pour les variables de type :**	**Signification**
%d	int	entier
%u	unsigned int	entier non signé
%f	double	réel
%lf	long double	réel
%c	unsigned car	caractère
%s	car*	chaîne de caractère

TABLE A.1 – Quelques formats de saisie/d'affichage pour les fonctions scanf /printf.

- **Remarque**
  Il existe d'autres fonctions d'entrées qui permettent de saisir des caractères et des chaînes de caractères :
  - La fonction **getchar** : permet de saisir un caractère au clavier.
  - La fonction **gets** : permet de saisir une chaîne de caractère au clavier.

A.4.6.2 Fonction de sortie "printf"

La fonction **printf** est une fonction d'impression formatée, ce qui signifie que les données sont converties selon un format particulier choisi. Sa syntaxe est la suivante :
Printf ("format", expr1, expr2,...,exprN) ;

- expr1,..., exprN : représentent les variables et les expressions dont les valeurs sont à représenter.
- Comme expliqué précédemment, la partie "format" indique la manière dont les valeurs des expressions expr1, ..., exprN sont affichées.
- Les conversions de format sont spécifées par un caractère précédé du signe % (voir la Table A.1).

A.5 Conclusion

Dans cette annexe, nous avons présenté les composants fondamentaux du langage C, qui complètent les structures principales de ce langage vues dans le chapitre 3, le chapitre 4 et le chapitre 5 (comme, les structures de contrôle, les structures itératives, etc).

Le langage C reste un des langages les plus utilisés actuellement pour programmer une grande partie des logiciels que vous connaissez (Word et Excel sont écrits à partir de C) : il est à la fois facile à utiliser et très efficace.

Bibliographie

1. Algorithmique-Cours : MCours.com.
2. E. Thiel "Cours d'Algorithmes et programmation en Pascal", Faculté des Sciences de Luminy, 2004.
3. J. Lonchamp "Introduction aux systèmes informatiques : Architectures, composants, mise en œuvre", Dunod (ISBN 978-2-10-075944-6), 2017.
4. H. Ghilas " Cours Algorithmiques et Structures de Données 1", Département de MI, Université de Béjaia, 2020.
5. K. Bachi " Cours Algorithmiques et Structures de Données 1", Département de MI, Université de Béjaia, 2021.
6. L. Poinsot "Cours Architecture et Système : chapitre 1- Architecture de base d'un ordinateur", Université Paris 13 - Institut Galilée, 2008.
7. M. Delest "Notes de cours ; Initiation à l'algorithmique", Université de Bordeaux, France, 2007.
8. N. Rebouh "Cours Algorithmiques 1", Département de MI, Université de Béjaia, 2011.
9. R. Ouzeggane " Cours Algorithmiques ", Département de Technologie, Université de Béjaia, 2016.
10. S. Boukerram "Cours Algorithmiques et Structures de Données", Département d'Informatique, Université de Béjaia, 2021.
11. S. Sadouki " Cours Algorithmiques ", Département des mines et géologie, Université de Béjaia, 2019.
12. T. Cormen, C. Leiserson et R. Rivest "Introduction à l'algorithmique, Dunod, 1994.
13. http ://public.iutenligne.net/informatique/algorithme-et-programmation.
14. http ://www.google.dz/url ?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CCwQFjACahUKEwiCxbHtm7PHAhXIbhQKHdpRAA8&url=http%3A%2F%2Ffdrouillon.free.fr%2Falldocs%2F_LivreC_CPP%2FChap6_%2520Strct%2520donnees%2520 listes%2520%26%2520algo.pdf &ei=iHPTVYLhOcjdUdqjgXg&usg=AFQjCNG8hDhJGLV9MQhOGlvLpt1zM3LpIQ&sig2=LF1B6YVr7lIA7L9XGK2YhQ.