République algérienne démocratique et populaire
Ministère de l'enseignement supérieur et de la recherche scientifique

Université Ibn Khaldoun de Tiaret
Faculté des Mathématiques et de l'informatique
Département d'informatique

Algorithmique et structure de données 1 et 2

Cours & exercices corrigés en langage algorithmique et en langage C

1ère LMD –Mathématiques et Informatique-

Conçu par :

- **Dr. Omar TALBI**

Année universitaire 2020-2021

Préface

OBJECTIFS

Ce polycopié de cours et d'exercices corrigés d'algorithmique et structure de données en Langage Algorithmique (LA) et en langage C présente et explique pas à pas les notions fondamentales de l'algorithmique. Il s'explique par plus de dix années d'enseignement d'algorithmique et de programmation à un public en première année universitaire dans le domaine des Mathématiques et de l'Informatique (MI).

Il présente l'algorithmique en utilisant le LA, acronyme de Langage Algorithmique, un vrai langage de programmation fortement inspiré du langage pascal. Le LA se veut un langage de programmation conçu expressément pour l'enseignement de l'algorithmique car il fait abstraction aux contraintes techniques liées aux spécificités des langages professionnels. Cependant, la traduction d'un programme écrit en LA se fait quasi automatiquement en un programme Pascal, C, Java ou autre. Dans ce polycopié, le langage C a été adopté pour formuler les algorithmes écrits en LA et produire des programmes informatiques exécutables par la machine.

Le niveau d'étude des notions abordées dans cet ouvrage est universitaire, spécifiquement le premier cycle dans les domaines des mathématiques et informatique ou celui des sciences et techniques.

LE CONTENU

Ce polycopié de cours se compose de deux parties, *PARTIE I Algorithmique et Structure de Données1 (ASDD1)* et *PARTIE II Algorithmique et Structure de Données2 (ASDD2)* et d'un *Annexe.*

La *première partie* se décline en six chapitres.

Le *premier chapitre* explique les concepts fondamentaux liés à l'informatique, présente un aperçu sur la structure et le fonctionnement d'un ordinateur et introduit les notions d'algorithme et de programme informatique.

Le *deuxième chapitre* introduit les notions de Langage de programmation et de Langage Algorithmique (LA) et présente les trois parties de mise en forme d'un algorithme par le LA. Il se focalise par la suite sur l'aspect des données et les opérations de base sur ces données. Il présente et explique les instructions de base (l'affectation et les instructions d'Entrée-Sortie), la construction d'un algorithme et sa représentation par un organigramme et leur traduction en langage C.

Le *troisième chapitre* est consacré au traitement conditionnel, alternatif et à choix multiples en LA ainsi que leur traduction en Langage C.

Le *quatrième chapitre* introduit et explique la notion de répétition des traitements à travers les boucles.

Le *cinquième chapitre* présente la notion de type personnalisé, les chaines de caractères et les structures de données de type tableaux à une et deux dimensions, communément appelés vecteurs et matrices.

Le *sixième chapitre* est consacré au type enregistrement, à savoir, sa définition et sa déclaration et sa manipulation. Il donne également un aperçu sur la possibilité de définir un tableau d'enregistrements.

La *deuxième partie* se décline en trois chapitres.

Le *premier chapitre* introduit la notion de sous-programmes, à savoir les fonctions et les procédures et explique les mécanismes de leur fonctionnement. Il aborde également l'utilisation de la récursivité dans les sous-programmes.

Le *deuxième chapitre* introduit les fichiers, leurs définitions, le mode de structuration des données dans un fichier ainsi que leurs manipulations en LA et en langage C.

Le *troisième chapitre* aborde les pointeurs, la gestion dynamique de la mémoire grâce aux variables pointées et les pointeurs de pointeurs, notamment les listes linéaires chainées, doublement chainées, les piles et les files. Aussi la manipulation de ce type de données sera abordée en LA et en langage C.

Annexe

L'*annexe* comporte les solutions des séries d'exercices proposées à chaque fin de chapitre.

UTILISATION DU POLYCOPIE DE COURS

La principale utilisation de ce polycopié est d'être pour les étudiants de première année MI un support de cours d'algorithmique pour apprendre à écrire des programmes, entre autres en langage C. Il conviendra de le commencer à son chapitre1 PARTIE1 et de suivre l'ordre des chapitres tels qu'ils apparaissent, et de faire de même pour la PARTIE2.

Un étudiant universitaire ou un enseignant peut aussi utiliser ce livre pour obtenir une clarification ou une définition précise sur un concept donné. Il peut également être utilisé simplement comme source d'exercices.

Table des matières

PARTIE I Algorithmique et Structure De Données 1 (ASDD1)

Cette première partie a pour objectif de présenter les notions d'algorithme et de structure de données. Les Connaissances préalables recommandées sont des notions d'informatique et de mathématiques.

Pour atteindre cet objectif, cette partie se décline en six chapitres :

CHAPITRE 1 : Introduction
CHAPITRE 2 : Algorithme séquentiel simple
CHAPITRE 3 : Les structures conditionnelles (en langage algorithmique et en langage C)
CHAPITRE 4 : Les boucles (en langage algorithmique et en langage C)
CHAPITRE 5 : Les tableaux et les chaînes de caractères
CHAPITRE 6 : Les types personnalisés

Chapitre 1 *Introduction*

Ce chapitre est divisé en quatre sections. La première section est une introduction à la notion de l'informatique. La deuxième section donne un aperçu sur la structure et le fonctionnement d'un ordinateur. La troisième section introduit la notion d'algorithme. La quatrième section propose une série d'exercices sur les concepts traités dans ce chapitre.

1. L'informatique, c'est quoi ?

Le mot informatique est un mot-valise, formé par la fusion de deux mots : information et automatique. En termes simples, ceci veut dire qu'on désire traiter l'information d'une manière automatique. Pour ce faire, on doit disposer d'une machine qui en soit capable. Cette machine est appelée ordinateur.

Nous donnons la définition suivante de l'informatique :

L'informatique est le traitement automatique (logique et ordonné) des informations faisant appel essentiellement à une machine appelée ordinateur.

Pour mieux comprendre cette définition, il nous semble important de définir les deux termes clés qui y sont utilisés à savoir, information et ordinateur.

a. Qu'est-ce qu'une information ?

Une information est le moyen de communiquer tout ce qui peut s'écrire, se présenter ou se dire.

Une information est une donnée possédant un sens.

Information = (donnée, sens)

i. Mais qu'est-ce qu'une donnée ?

Une donnée est représentée par un ou plusieurs caractères, sachant que l'ensemble des caractères, est {Alphabet, signes de ponctuation, symboles spéciaux, etc.}.

⇨ Une donnée ne possède pas forcément un sens.

Exemple

La chaine de caractères *Xwil3?#* est une donnée (*Xwil3?#* n'a pas de sens).

La chaine de caractères *clavier* est une information (*clavier* a un sens, on peut même lui associer une image)

b. Qu'est-ce qu'un ordinateur ?

Nous donnons la définition suivante d'un ordinateur :

Un ordinateur est une machine (calculateur) électronique très rapide qui accepte une information numérique à l'Entrée, la traite suivant le Programme enregistré dans sa Mémoire centrale et produit l'information (le résultat) à la Sortie.

Afin de mieux comprendre cette définition, il est important de définir les termes : Entrée, Programme, Mémoire centrale et Sortie. Cela est l'objet de la section suivante.

2. Structure et fonctionnement d’un ordinateur

Nous présentons dans ce qui suit un schéma haut niveau et simplifié qui montre les étapes à travers lesquelles l’ordinateur parvient à traiter des informations. Nous montrons également les composants essentiels de l’ordinateur qui interviennent dans ce traitement comme illustré par la Figure 1.1.

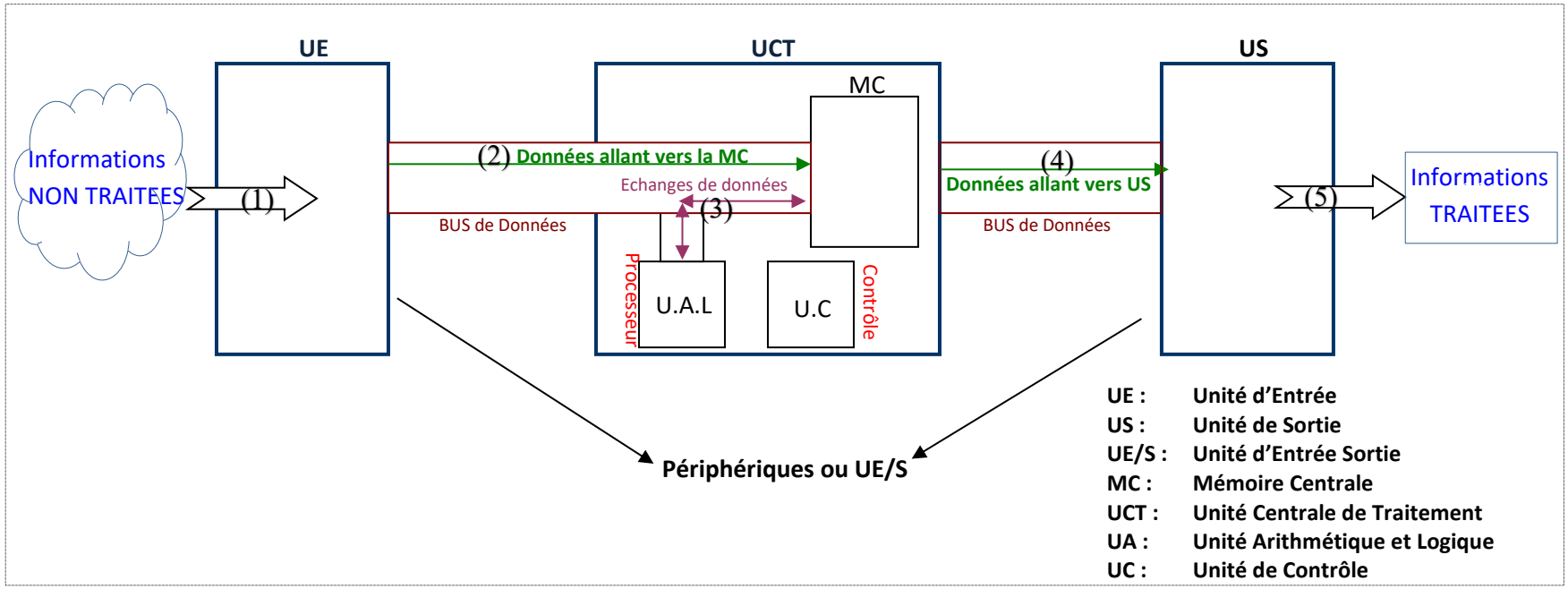


Figure 1.1 : Schéma haut niveau et simplifié du processus de traitement des informations dans un ordinateur.

Pour que les informations non traitées parviennent à être traitées par l'ordinateur, elles doivent suivre les étapes suivantes telles qu'illustrées par la Figure 1.1.

(1) Saisie des Informations NON TRAITEES par l'UE,

(2) ces dernières sont véhiculées par le BUS de données vers l'UCT, plus précisément vers la MC,

(3) l'UAL (processeur) traite les informations (échange de données avec la MC) sous le contrôle de l'UC,

(4) une fois les données traitées, elles sont transportées de la MC vers L'US via le BUS de données et

(5) finalement l'US restitue les Informations TRAITEES.

⇨ Il est très important de savoir, qu'une information/donnée ne peut être traitée que si elle se trouve en mémoire centrale. Cette dernière est par conséquence un composant incontournable dans le processus du traitement de l'information.

A travers ce schéma haut niveau, nous pouvons distinguer deux concepts fondamentaux, à savoir les composants de l'ordinateur (UE/S, UCT, UAL, MC, …) qu'on appelle Matériel en anglais *Hardware* et le traitement des informations/données effectué par ce matériel (Saisie, transfert, échange, calcul arithmétique et logique, …) qu'on appelle Logiciel/Programmes en anglais *Software*.

a. Le Matériel

Chaque composant de l'ordinateur possède une fonction particulière, le Tableau 1.1 en cite les principales.

Composant	Fonction
Processeur (UAL)	Calcul arithmétique et logique
Mémoires	Stockage des données
Carte graphique	Affichage vidéo

Tableau 1.1 : Fonctions principales des composants essentiels d'un ordinateur.

Etant donné que la Mémoire centrale est un composant incontournable dans le processus de traitement des données, nous donnons dans ce qui suit plus de détail sur ce composant.

Nous commençons tout d'abord par définir ce qu'est une mémoire en générale et quelles sont ces caractéristiques.

Nous donnons la définition suivante d'une mémoire :

Une mémoire est une ressource de stockage que l'ordinateur peut employer pour stocker des données/informations temporairement ou sur de longues durées afin de les réutiliser ultérieurement.

Une mémoire est caractérisée par :

- Sa capacité = volume global d'informations (en BIT[1]s qu'on verra par la suite) que la mémoire peut stocker ;
- Sa volatilité/non-volatilité = aptitude à conserver ou non les données en absence d'énergie électrique.
- Le temps d'accès = intervalle de temps entre la demande de Lecture (de l'UE vers la MC) / Ecriture (de La MC vers l'US) et la disponibilité de la donnée.

Le tableau 1.2 donne un récapitulatif sur les types de mémoires

[1] BIT : Abréviation de *Binary Digit*

Types de mémoires	Signification et fonction	Image
Mémoire Vive/Interne/Centrale	⇨ RAM - Random Access Memory. ⇨ Volatile = perd son contenu en l'absence d'énergie électrique. ⇨ Mémorise les informations en cours de traitement. ⇨ Faible capacité de stockage par rapport aux mémoires secondaires. ⇨ Accès aux données très rapide	
Mémoire Morte	⇨ ROM - Read Only Memory ⇨ Non-volatile, ⇨ ROM-BIOS : contient le programme du constructeur de la machine. (BIOS= Basic Input Output System)	
Mémoire Secondaire / Auxiliaire / de Masse/ Physique/ Externe.	⇨ Disque dur, Disquette magnétique (Stockage magnétique) ⇨ CD, DVD (Stockage optique) ⇨ Sauvegarde des informations. ⇨ Grande capacité de stockage ⇨ Accès aux données moins rapide que la RAM.	

Tableau 1.2 : Tableau récapitulatif des types de mémoires.

La Figure 1.2 montre comment qu'est organisée une mémoire centrale et comment que sont stockées les données dans la MC.

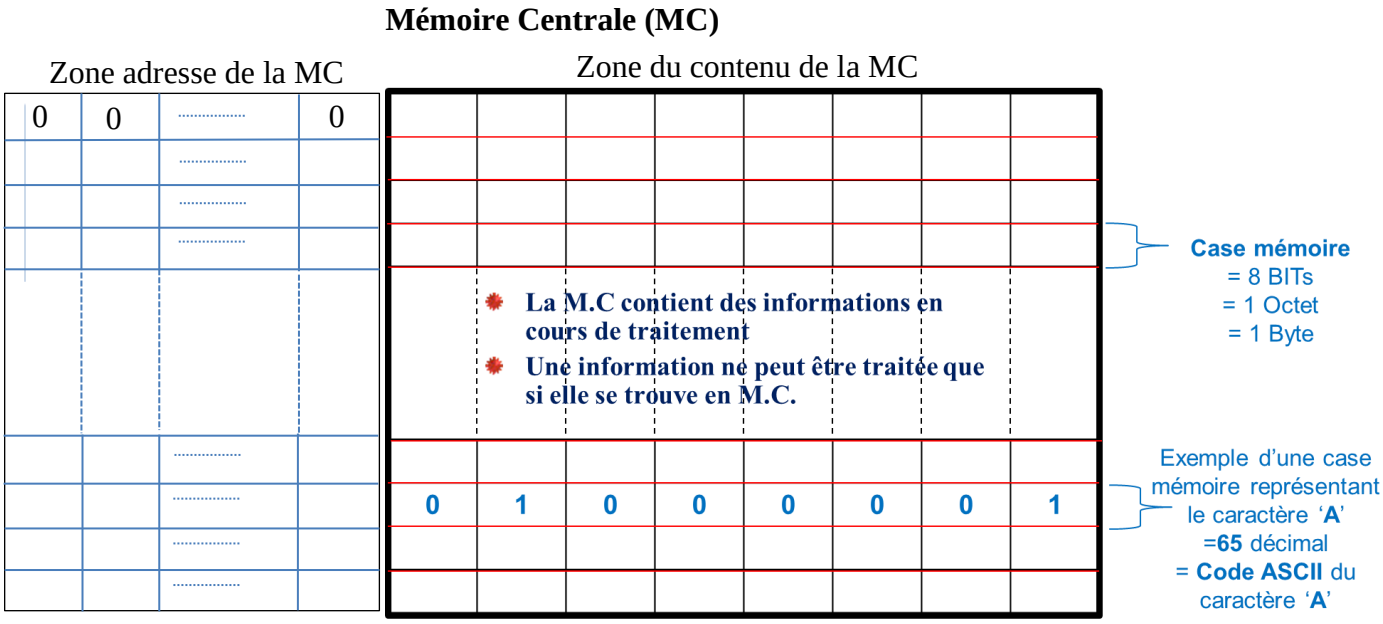
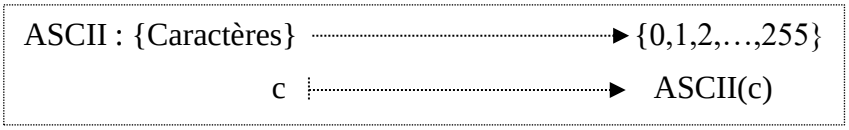


Figure 1.2 : Organisation de la mémoire centrale.

La Figure 1.2 nous montre que :

- Les cellules mémoires sont contiguës (l'une à la suite de l'autre)
- Chaque cellule est repérée par une adresse.
- Une information/donnée est représentée en MC par une suite de 0 et de 1.

Cette représentation binaire obéit à une norme informatique de codage de caractères qui pour chaque caractère associe un nombre et un seul. Cette norme est une application, appelée ASCII, acronyme de American Standard Code for Information Interchange (Code américain normalisé pour l'échange d'information) définie ainsi :



Exemple.

$ASCII('A') = (65)_{10} = (01000001)_2$

Le caractère 'A' a pour code ASCII le nombre 65 en décimale et est représenté en MC par 01000001 son équivalent en binaire.

Une mémoire étant le récipient des données, sa capacité est donc le volume d'informations qu'elle peut stocker. Le Tableau 1.3 donne les valeurs des unités de mesure d'une information.

Unité	Valeur
BIT	1 ou 0
Octet ou Byte	8 BITs
Kilo ou K	$2^{10} = 1024$
Méga	2^{10} K = 2^{20} = 1048576
Giga	2^{10} Méga =2^{10} K=2^{30} = 1 073 741 824
Téra	2^{10} Giga = 2^{20} Méga = 2^{30} K = 2^{40} = 1 099 511 627 776

Tableau 1.1 : Unités de mesure de l'information.

b. Le Logiciel

Afin de pouvoir réaliser le traitement de l'information, le Matériel a besoin d'être piloté (guidé) par un programme informatique.

i. Programme informatique

Un programme informatique est une séquence d'instructions et de données destinées à être exécutées par un ordinateur.

ii. Logiciel

Un logiciel est un ensemble de programmes destinés à accomplir un certain nombre de tâches et de services répondant à des besoins spécifiques.

On distingue différents types de Logiciels selon les services qu'ils offrent à leurs utilisateurs :

- **Système d'exploitation** (MS-DOS, OS, Windows, Unix, …), un logiciel qui permet l'exploitation des ressources de l'ordinateur. Il se situe en tant qu'intermédiaire entre l'utilisateur et l'ordinateur.
- **Logiciels standards**, comme les logiciels de traitement de texte MSWord, de tableurs MSExcel...
- **Progiciels**, logiciels spécifiques, par exemple des programmes réalisant la paie du personnel d'une entreprise, la comptabilité ou la gestion des stocks d'une entreprise, ...).

⇨ Dans ce cours nous nous intéressons à l'écriture des programmes informatiques.

Pour écrire un programme informatique, nous utilisons un **Langage de programmation**. Mais avant cela, nous devons trouver l'**algorithme** qui lui correspond. Dans ce qui suit, nous allons aborder la notion d'algorithme.

3. Algorithme

L'origine du mot Algorithme vient de l'arabe الخُوَارِزْمِيّ, al-Ḵuwārizmiyy, nom du mathématicien perse Al-Khwarizmi déformé par le grec ancien par le mot arithmós qui veut dire « nombre ».

a. Exemples introductifs

Nous introduisons la notion d'algorithme à travers les exemples suivants.

Exemple 1. ***Envoi d'un document*** (*Extrait du mode d'emploi d'un télécopieur).*

1. **Insérez** le document dans le chargeur automatique.
2. **Composez** le numéro de fax du destinataire à l'aide du pavé numérique.
3. **Enfoncez** la touche envoi pour lancer l'*émission*.

} Tâche = ***envoi d'un document par fax.***

Exemple 2. ***Trouver son chemin*** *(Extrait d'un dialogue entre un touriste égaré et un autochtone).*

1. Pourriez-vous m'indiquer le chemin de la gare, s'il vous plait ?
2. Oui bien sûr : vous **allez tout droit** jusqu'au prochain carrefour, vous **prenez à gauche** au carrefour et ensuite vous **prenez** la troisième à droite, et vous verrez la gare juste en face de vous.
3. Merci

} Tâche= ***aller à la gare.***

Exemple 3. ***Résolution de l'équation du second degré $ax^2 + bx + c = 0$ dans l'ensemble des réels.***

1. **Si** $b^2 - 4ac > 0$, $x1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$, $x2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$
2. **Si** $b^2 - 4ac = 0$, $x1 = \frac{-b}{2a}$, $x2 = \frac{-b}{2a}$
3. **Si** $b^2 - 4ac < 0$, **Pas de solution dans l'ensemble des réels.**

} Tâche= ***Résolution d'une équation de degré 2 dans R.***

De ces trois exemples précédents, nous pouvons dire que pour réaliser une **tâche** nous devons effectuer une suite d'**actions**, sachant qu'une **action** est une **instruction** qui agit sur une ou plusieurs données. Le tableau 1.4 illustre ceci, dans le cas de l'exemple 1.

# Action	Nom de l'action	Données sur lesquelles agit l'action	Tâche à réaliser
1	**Insérer**	1. Document 2. Chargeur automatique	**Envoi d'un document par Fax**
2	**Composer**	3. Numéro de fax 4. Pavé numérique	
3	**Enfoncer**	5. Touche d'envoi	

Tableau 1.4 : Actions qui agissent sur les données pour réaliser la tâche.

Les suites d'actions 1, 2 et 3 agissent sur leurs données respectives et permettent de réaliser les tâches désirées, elles sont appelées des **pseudo-algorithmes**. En d'autres termes des algorithmes formulés en langage naturel.

b. Définitions

Un algorithme peut être défini par l'une des définitions suivantes :

« Une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents ».

« Le résultat d'une démarche logique de résolution d'un problème ».

« Un procédé qui pour un problème posé à partir de données en entrée conduit au résultat désiré par un nombre fini d'actions ».

Un algorithme doit satisfaire les propriétés suivantes :

- **Généralité**. Un algorithme doit toujours être conçu de manière à envisager toutes les éventualités d'un traitement.
- **Finitude**. Un algorithme doit s'arrêter au bout d'un temps fini.
- **Définitude**. Toutes les opérations d'un algorithme doivent être définies sans ambiguïté.
- **Répétitivité**. Généralement, un algorithme contient plusieurs itérations, c'est à dire des actions qui se répètent plusieurs fois.
- **Efficacité**. Idéalement, un algorithme doit être conçu de telle sorte qu'il se déroule en un temps minimal et qu'il consomme un minimum de ressources.

La figure 1.3 illustre la notion d'algorithme qui vient en amont de l'écriture du programme informatique.

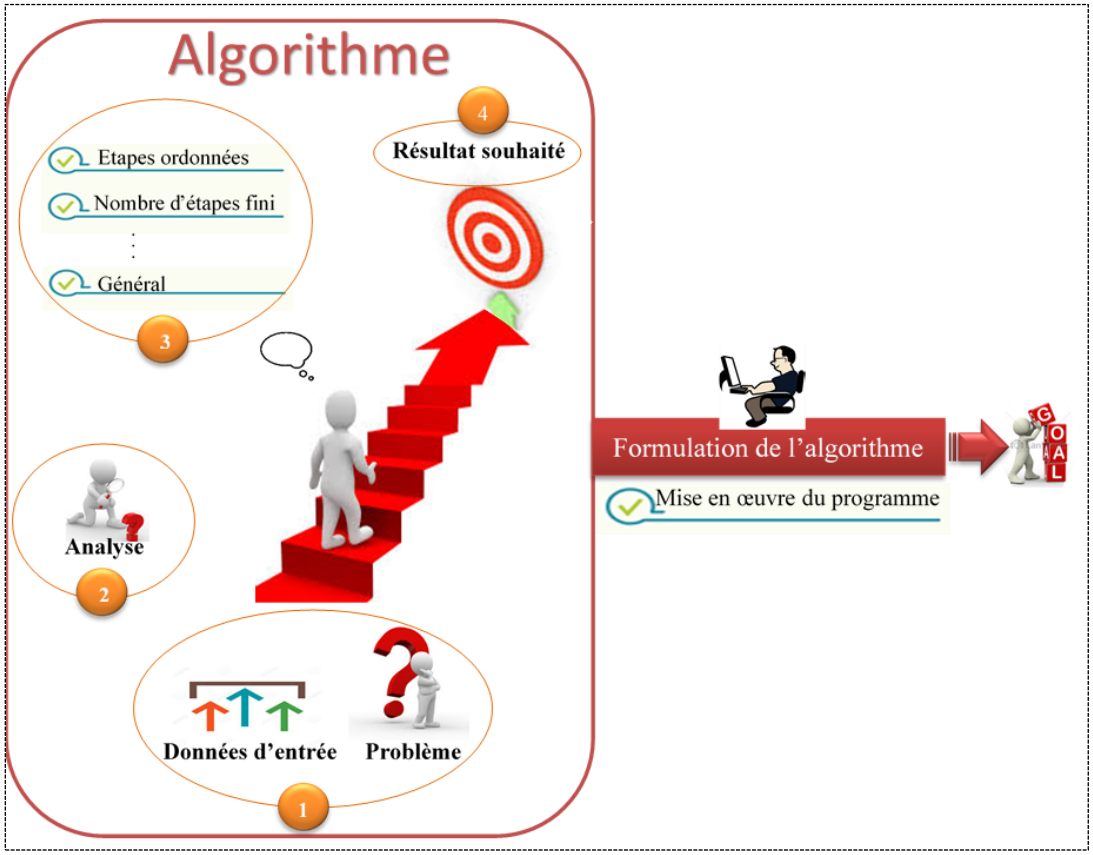


Figure 1.3 : Du problème au programme informatique.

Face à un problème donné, à partir de données en entrée, on analyse et on élabore un procédé qui nous conduit au résultat souhaité tout en satisfaisant les propriétés sus-énumérées. Ce procédé ou **algorithme** va permettre de mettre en œuvre un programme informatique qui sera soumis à l'ordinateur.

4. Série d'exercices -Introduction à l'informatique-

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

1. Qu'est-ce l'informatique ? Qu'est-ce qu'un ordinateur ? Faire une comparaison entre l'être humain et l'ordinateur.

2. Qu'est-ce qu'une information ? Quelle est la différence entre donnée et information ?

3. Quelles sont les différences entre langages machine, assembleur et évolué ?

4. Qu'est qu'un système informatique ?

5. Quels sont les composants d'un ordinateur (architecture d'un PC) ?

6. A quoi sert une carte mère ?

7. Dans les équipements informatiques, les données sont représentées par un signal électrique de quelle forme est-il ?

8. Identifier les périphériques suivants, et dire s'ils sont des périphériques d'entrée, de sorties, ou d'entrée /sortie. : Clavier, Ecran, Imprimante, Lecteur DVD, Scanner, Webcam, Graveur DVD, Souris, Lecteur disquette, Lecteur CDROM, Microphone, Casque, les enceintes, Graveur CDROM, Connecteur USB.

9. Désigner dans le tableau suivant le type d'opération et le périphérique utilisé (Entrée, Sortie, stockage). :

Opération	**Type de l'opération**	**Périphérique**
Je tape un texte		
Je scanne un graphe		
J'imprime des résultats		
Je sauvegarde des données		
J'écoute de la musique à partir de mon PC		

10. Que contient la mémoire principale ?

11. Quelle est la différence entre la RAM et la ROM.

12. Quelle est la différence entre la mémoire centrale et la mémoire auxiliaire ?

13. Les logiciels peuvent se classifier de plusieurs manières. Expliquer et décrire la classification des logiciels par rapport à leur importance.

14. Donner un schéma fonctionnel simplifié d'un ordinateur.

15. Quelles caractéristiques permettent d'apprécier les performances d'un ordinateur ? (Citer seulement celles relatives au processeur, à la mémoire et aux unités de stockage).

16. Je suis dans la rue, et je veux rentrer chez moi (dans un immeuble ou il faut prendre l'ascenseur). Proposer un algorithme.

17. On cherche à résoudre dans R l'équation $ax^2+bx+c=0$, en d'autres termes pour les données réelles a, b et c, donner la ou les solutions de l'équation dans R. Proposer un algorithme.

Chapitre 2 Algorithme séquentiel simple

Ce chapitre est divisé en huit sections. La première section est une introduction aux notions de Langage de programmation et Langage Algorithmique (LA). La deuxième section présente les trois parties de mise en forme d'un algorithme par le LA. La troisième est consacrée aux données, à leurs types et leurs natures. La quatrième section aborde les opérations de base sur les données. La cinquième section se focalise sur les instructions de base, à savoir, l'affectation et les instructions d'Entrée-Sortie. La sixième présente la construction d'un algorithme et sa représentation par un organigramme. La septième section montre leur traduction en langage C. La huitième section propose une série d'exercices sur les concepts traités dans ce chapitre.

1. Notion de langage et de langage algorithmique

Un **Langage de programmation** étant destiné à une machine, il doit donc être compris par cette dernière. Pour ce faire il doit posséder, un vocabulaire, une grammaire (respecter des règles syntaxiques) et une sémantique (signification).

Il doit aussi permettre de formuler des algorithmes et de produire des programmes informatiques qui appliquent ces algorithmes.

La Figure 2.1 donne une vue conceptuelle sur un langage de programmation

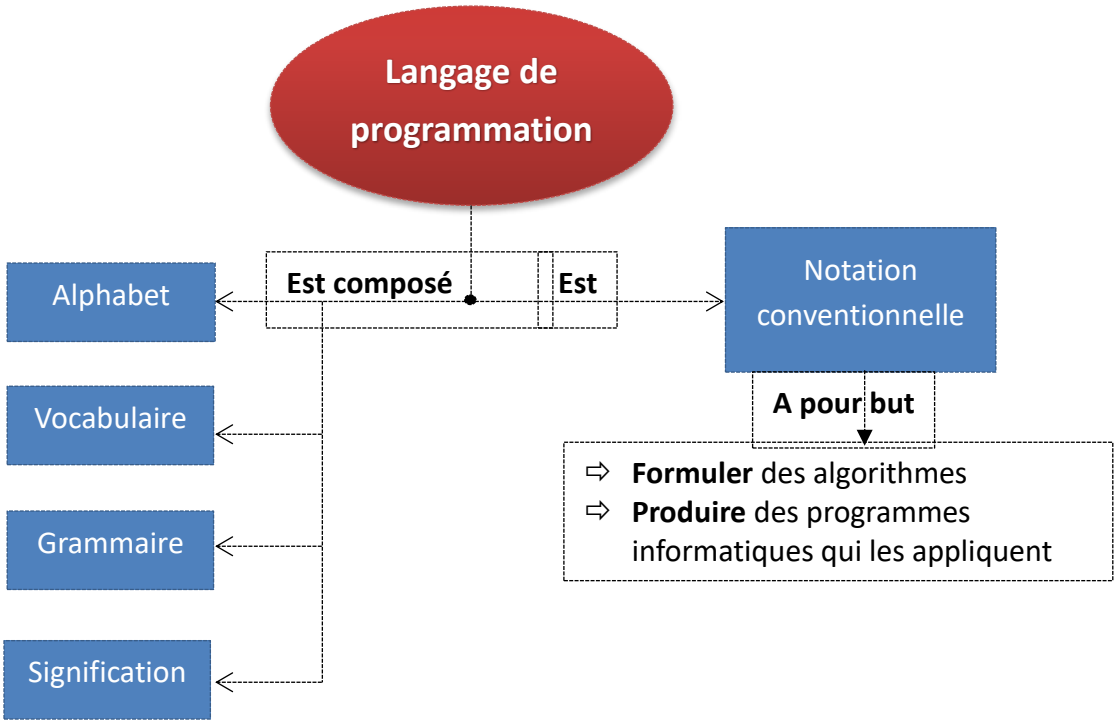


Figure 2.1 Vue conceptuelle d'un langage de programmation

La figure 2.1 montre bien que l'algorithme est au cœur même du langage de programmation. On a vu dans les exemples du *chapitre II section 3 Algorithme* qu'on pouvait décrire un algorithme à travers des pseudo-algorithmes.

Cependant, exprimer en langage naturel un algorithme présente :

- Des difficultés pour exprimer certaines idées, telles que répéter plusieurs fois les mêmes actions.
- Des ambiguïtés, un mot possédant plusieurs significations.
- Plusieurs façons d'exprimer la même solution avec un risque d'interprétations différentes.

Pour pallier les insuffisances du langage naturel dans la description d'un algorithme (pseudo-algorithme), nous adoptons un formalisme constitué d'un ensemble de conventions présenté sous forme d'un texte structuré qu'on appellera : **Langage Algorithmique** (LA). Le LA se veut un langage de programmation qui fait abstraction aux contraintes techniques liées aux spécificités des langages de programmation professionnels.

La section suivante montre la mise en forme d'un algorithme et décrit ses différentes parties.

2. Parties d'un algorithme

Comme le montre la figure 1.5, la mise en forme d'un algorithme par le LA se compose de trois parties : (1) Entête, (2) Partie Déclaration et (3) Corps.

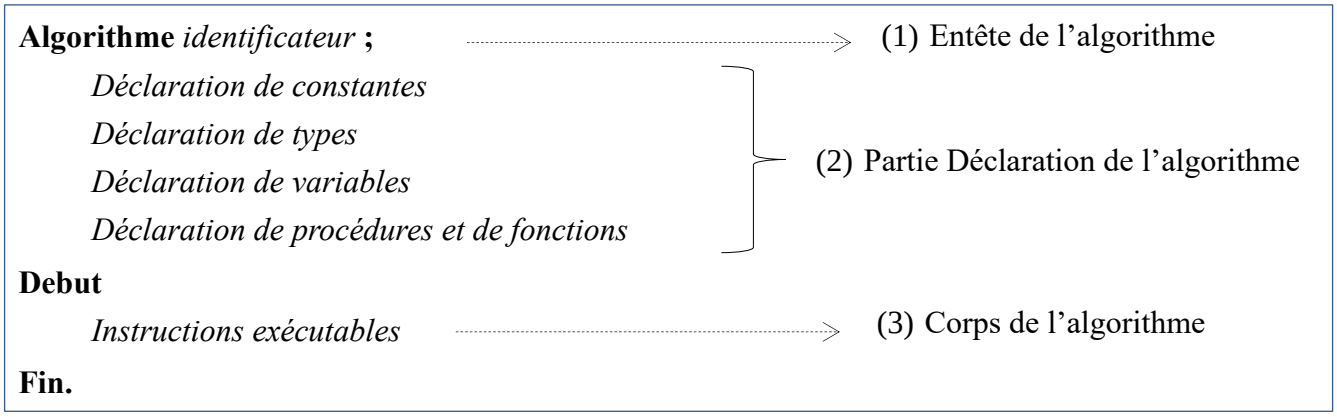


Figure 2.5 : Mise en forme d'un algorithme par le LA.

a. Entête d'un algorithme :

Le mot **Algorithme** est le premier mot de n'importe quel algorithme formulé par le LA. Il est suivi du nom de l'algorithme appelé identificateur et se termine par un point-virgule **(;)**.

En règle générale un identificateur est utilisé pour nommer : (1) un algorithme, (2) une variable, (3) une constante, (4) une procédure, (5) une fonction ou (6) un type.

Un identificateur doit commencer par une **lettre non accentuée** ou le symbole **underscore** (_) et ne peut être suivi que de **chiffres** et/ou de lettres non accentuées et/ou le symbole **underscore**.

b. Partie déclaration

Dans cette partie, sont déclarés toutes les **données** manipulées par l'algorithme (constantes ou variables), les **types** définis par l'algorithme, les **fonctions** et les **procédures** auxquelles l'algorithme fait appel.

c. Corps de l'algorithme

Cette partie de l'algorithme appelé Corps de l'algorithme contient les **instructions** qui conduisent à la solution désirée. Elle est délimitée par les mots clés **Debut** et **Fin**.

Exemple.

La figure 1.6 est un exemple d'écriture d'un algorithme en LA permettant de saisir le rayon d'un cercle et d'afficher le résultat du calcul de sa surface.

```
Algorithme surface_cercle;
const pi=3.14 ;
var r: reel; (* r est le rayon du cerle *)
Debut
        Ecrire('Quel est le rayon du cercle :');
        lire(r);
        Ecrire('la surface du cercle est :',pi*r*r) ;
Fin.
```

Figure 2.6 : Exemple d'écriture en LA de l'algorithme qui calcule la surface d'un cercle.

Remarques

- On convient que dans le LA on adopte une syntaxe d'écriture qui ne tient pas en compte des majuscules et des minuscules ; on dit que le LA **n'est pas sensible à la casse**. Pour l'exemple ci-dessus **r** ou **R** désignent la même variable rayon du cercle.

- Un **mot-clé** est un mot qui est **réservé**, il ne peut pas être utilisé en tant qu'identificateur. Pour l'exemple ci-dessus **Algorithme, const, var, reel, Debut, Fin** sont des mots-clés ou des mots réservés.

- (* *r est le rayon du cercle* *) est un commentaire. Les commentaires servent à documenter le LA et ne sont pas pris en considération par ce dernier. Dans le LA, un commentaire doit être écrit entre les symboles **(*** et ***)**.

3. Les données

a. Aspect de données

Une donnée utilisée dans un LA doit posséder : (1) un **nom**, (2) un **type** et (3) une **valeur**.

1. Le **nom** est appelé **identificateur**
2. Le **type** peut être : **entier**, **reel**, **caractere**, **logique** (**booleen**) ou **chaine de caractere**.
3. La **valeur** peut être respectivement pour ces types : **3, 3.5, 'a'**, **Vrai, Faux, 'informatique'**.

Une donnée possède une **nature**, elle peut être (1) **constante** ou (2) **variable** :

1. Dans le cas d'une constante, le type, le nom et la valeur sont fixes.
2. Dans le cas d'une variable, le type et le nom sont fixes tandis que la valeur est variable.

Remarques

- Avant d'utiliser une donnée dans un LA, il faut préciser son nom, son type, sa valeur et sa nature. C'est ce qu'on appelle : Déclaration de la donnée.

- Dans la suite du cours, nous convenons d'utiliser en LA des symboles spéciaux, à savoir : < >, { } et [ ] dans l'écriture des schémas syntaxiques des déclarations et des instructions. Le tableau 2.1 donne la signification de ces symboles.

Symbole	Signification
< >	Ce qui se trouve à l'intérieur de ces deux symboles est remplacé selon le choix du programmeur en respectant ses définitions.
{ }	Ce qui se trouve à l'intérieur de ces deux symboles peut être répété 0,1,2, ,n fois.
[ ]	Ce qui se trouve à l'intérieur de ces deux symboles est facultatif (peut être omis).

Tableau 2.1 : Signification des symboles spéciaux en LA.

b. Déclaration de variables et de constantes

Chaque donnée utilisée dans le LA doit être déclarée. Cette déclaration se traduit par une **réservation / allocation** d'un espace mémoire. La taille de cet espace mémoire varie selon le type de la donnée déclarée.

- En LA, il existe une possibilité de donner un nom (identificateur) à une constante. On convient d'utiliser le schéma syntaxique suivant pour déclarer une ou plusieurs constantes en LA tel que donné par la figure 2.7.

const < identificateur > = < valeur constante >; {< identificateur > = < valeur constante >}

Figure 2.7 : Schéma syntaxique de la déclaration d'une ou plusieurs constantes.

Exemple

const pi=3.14; test=faux; discipline='informatique';

const nbre_etud=540;

- On convient d'utiliser le schéma syntaxique suivant pour déclarer une ou plusieurs variables en LA tel que donné par la figure 2.8.

var <identificateur> {**,** <identificateur>}**:** <type>**;** { <identificateur> { **,**<identificateur> }**:** <type>**;** }

Figure 2.8 : Schéma syntaxique de la déclaration d'une ou plusieurs variables.

Exemple

var a, b, c : entier; q:logique; x,y :réel ; signe=caractere ;

4. Opérations de base

Nous distinguons en LA trois types d'opérateurs, (1) les opérateurs arithmétiques, (2) les opérateurs de relation ou de comparaison et (3) les opérateurs logiques.

Ces opérateurs vont mettre en relation des opérandes de type entier, réel, logique ou caractère pour former deux types d'expression, à savoir : les expressions arithmétiques et les expressions logiques.

a. Les expressions arithmétiques

On dispose de six opérateurs arithmétiques en LA pour réaliser les opérations arithmétiques de base, à savoir : l'addition, la soustraction, la multiplication, la division, la division entière et le modulo comme illustré par le tableau 2.2.

Opérateur arithmétique disponible en LA	Symbole utilisé en LA
Addition	+
Soustraction	-
Produit	*
Division	/
Division entière	**div**
Modulo (reste de la division entière)	**mod**

Tableau 2.2 : Symboles utilisés pour les opérateurs arithmétiques en LA

Dans les expressions arithmétiques il existe une compatibilité de type telle que donnée par le tableau 2.3.

Type du 1er opérande	Type du 2ème opérande	+	-	/	*	div	Mod
Entier	Entier	Entier	Entier	Réel	Entier	Entier	Entier
Réel	Réel	Réel	Réel	Réel	Réel	N'est pas défini	
Entier	Réel	Réel	Réel	Réel	Réel	N'est pas défini	
Réel	Entier	Réel	Réel	Réel	Réel	N'est pas défini	

Tableau 2.3 : Compatibilité de types dans les expressions arithmétiques

Remarque

Au moins un espace est obligatoire entre les opérateurs div et mod et leurs opérandes. Par exemple, étant données deux variables entières a et b, on écrit : a div b a mod b et non pas adivb amodb.

b. Les expressions logiques

Les expressions logiques sont formées d'opérateurs de relation (comparaison) : >, <, >=, <=, =, <> et/ou d'opérateurs logiques : not (non), and (et), or (ou inclusif).

Les opérandes reliés par des opérateurs de relation doivent être de même type. Le résultat de l'expression logique ainsi obtenue est de type logique.

Exemple

Etant données les déclarations suivantes en LA : var a,b: entier; q: logique;

L'expression **a=b** est une expression logique

L'expression **(a+1)=b** est une expression logique.

L'expression **(a=b) and ((a+1)=b)** est une expression logique

c. Priorité des opérateurs arithmétiques, logiques et de comparaison

D'une manière générale la priorité des opérateurs dans une expression est comme le montre la figure 1.9.

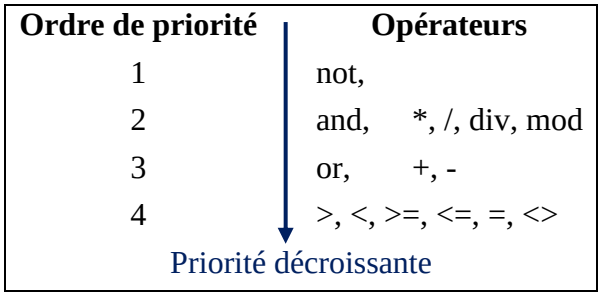


Figure 2.9 : Priorité des opérateurs dans une expression.

Pour modifier l'ordre d'exécution des opérateurs il est possible d'utiliser les parenthèses.

d. Autres fonctions standards

Les trois types déjà vus, à savoir : **entier**, **logique** et **caractere** décrivent les données appartenant à des **ensembles ordonnés.** Pour un élément, on peut dire quel élément le **précède** et quel élément le **suit**. On peut donc introduire les opérations de relation sur les données de ces ensembles.

Le tableau 2.4 liste les fonctions standards concernant ces données.

Signification	Fonction LA	Type argument	Type résultat
Prédécesseur de l'élément	**pred(x)**	**E, C, L**	**E, C, L**
Successeur de l'élément	**succ(x)**	**E, C, L**	**E, C, L**
N° d'ordre dans la suite	**ord(x)**	**C,L**	**E**
Inverse à ord	**chr(x)**	**E**	**C**

E: Entier
C: Caractere
L: Logique

Tableau 2.4 : Fonctions standards -données appartenant à des ensembles ordonnés-

5. Instructions de base

a. Instruction d'affectation

Le schéma syntaxique de l'instruction d'affectation est comme celui donné par la figure 1.10.

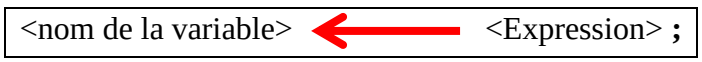


Figure 2.10 : Schéma de l'instruction d'affectation.

<Expression> est calculée et sa valeur est affectée à la variable qui se trouve à gauche.

Dans une instruction d'affectation, il existe une compatibilité de type telle que donnée par le tableau 2.4.

Type de l'expression	Type de la variable
Entier	Entier
	Réel
Réel	Réel
Logique	Logique
Caractère	Caractère

Tableau 2.4 : Compatibilité entre types dans une Instruction d'affectation.

Exemple

```
algorithme affectation_exp_logique;
 var a,b: entier;  q: logique;
    debut
       q← a=b;
       q← (a+1) =b;
    fin.
```

b. Instruction d'entrée et de sortie

Pour faire parvenir des données de la MC vers l'écran, on parle généralement d'affichage, on utilise une instruction de **sortie**. En LA cette instruction est traduite par l'instruction **Ecrire.** Sa syntaxe d'écriture est donnée la figure 2.11.

Ecrire(liste de données constantes et/ou variables séparées par des virgules**);**

Figure 2.11 : Syntaxe d'écriture de l'instruction de sortie Ecrire en LA.

De même pour faire parvenir des données du clavier vers la MC, on parle généralement de saisie, on utilise une instruction d'**entrée**. En LA cette instruction est traduite par l'instruction **Lire.** Sa syntaxe d'écriture est donnée par la figure 2.12.

Lire(liste de données variables séparées par des virgules**);**

Figure 2.12 : Syntaxe d'écriture de l'instruction d'entrée Lire en LA.

6. Exemple de construction d'un algorithme simple

L'exemple traitée dans cette section consiste à écrire en LA un algorithme qui à partir de la valeur du rayon d'un cercle saisie par l'utilisateur, calcule et affiche le périmètre et la surface de ce cercle.

```
Algorithme surface_perimetre_cercle;
const pi=3.14 ;
var r: reel; (* r est le rayon du cerle *)
    s,p :reel ; (* s et p sont respectivement la surface et le périmètre du cercle *)
Debut
        Ecrire('Donner le rayon du cercle :'); (* Instruction d'affichage sur écran *)
        Lire(r); (* Instruction invitant l'utilisateur à saisir la valeur de r *)
        s←pi*r*r ; (* Instruction d'affectation du calcul de la surface du cercle à la variable s*)
        p←2*pi*r ; (* Instruction d'affectation du calcul du périmètre du cercle à la variable p*)
        Ecrire('La surface du cercle est :',s, 'Le périmètre du cercle est :',p ) ; (* Instruction d'affichage sur
                écran de s et de p précédées respectivement des messages leur corresponaent *)
Fin.
```

Figure 2.13 : Exemple d'écriture en LA de l'algorithme qui calcule la surface et le périmètre d'un cercle.

7. Traduction en langage C

Les concepts abordés dans ce chapitre ont touché principalement à la mise en forme d'un algorithme par le LA, les données et leurs types, les opérations de base sur les données, les trois instructions de base, à savoir : l'affectation, la lecture et l'écriture. Dans cette section, nous donnons la traduction de ces concepts en langage C tel qu'illustré dans le tableau 2.5.

	LA	**Langage C**

Mise en forme d'un algorithme	**Algorithme** *identificateur* **;**
	Déclarations
	Debut
	Instructions exécutables
	Fin.
		`main()`
		`{`
		`<déclarations>`
		`<instructions>`
		`return 0;`
		`}`

Identificateurs	Concernant les identificateurs, les mêmes règles déjà vues en LA s'appliquent au langage C mais ce dernier est sensible à la casse. Cela veut dire que le langage C fait une distinction entre les lettres majuscules et les lettres minuscules. Il ne traite pas de la même façon les données ou les instructions selon qu'elles sont écrites en majuscules ou en minuscules.	
Les types	**Entier** **reel** **caractere** **logique**	**short** ou **int** ou **long** **float** ou **double** ou **long double** **char** *(entier qui tient sur un octet)* N'existe pas.
Déclaration d'une constante	**const** <identificateur>=<valeur>; **const pi=3.14 ;**	**const** <type> <identificateur>=<valeur>; **const float** pi=3.14 ;
Déclaration d'une variable	**var** <identificateur>=<valeur>; **var** a **: entier ;** s **: reel;** c**: caractere;**	<type> <identificateur> ; **int** a ; **float** s ; **char** c ;

Opérateurs arithmétiques	+ - * / **Div** **mod**	**+** **-** ***** **/** **/** **%**
Opérateurs de comparaison	= <> > < >= <=	**==** **!=** **>** **<** **>=** **<=**
Opérateurs logiques	**not** **and** **or**	**!** **&&** **||**
Instruction d'affectation	<Nomvariable> ←<Expression> **;** a ← 5 ;	<Nomvariable>**=**<Expression> **;** a **=** 5 **;**
Ecriture	**Ecrire(**<Expr1>**,** <Expr2>**, ...)**	**printf(**"<format>",<Expr1>,<Expr2>,... **)** où : "<format>" *: format de représentation des variables et expressions.* <Expr1>, ... *: variables et expressions*
Lecture	**Lire(**<Nomvariable > , <Nomvariable >, …**)**	**scanf(**"<format>"**,<@Var1>,<@Var2>, ...)** où : "<format>"*: format de lecture des données.* <@Var1>,...*: adresses des variables où seront affectées les données.*
Commentaires	(* Texte du commentaire ***)**	**//** Commentaire dans une même ligne **/*** Commentaire sur plusieurs lignes … ….. ****/***
Remarque	Les instructions d'écriture et de lecture **scanf** et **printf** font partie de la librairie standard **stdio.h**. Afin d'utiliser ces deux instructions, il faut inclure en haut du programme C la directive **#include <stdio.h>**.	
Exemple d'un programme en C	**#include <stdio.h>** **/*** Ce programme calcule et affiche la somme de 2 nombres entiers introduits au clavier. ****/*** **main()** **{** **int** NOMBRE1, NOMBRE2, SOMME; **printf(**"Entrez un nombre entier :"**);** **scanf(**"%i", **&**NOMBRE1**);** **printf(**"Entrez un nombre entier :"**);** **scanf(**"%i", **&**NOMBRE2**);** SOMME **=** NOMBRE1 **+** NOMBRE2**;** **printf(**"La somme est: **%i \n**", SOMME**);** **return** 0**;** **}**	

Tableau 2.5 : Traduction du LA vers le langage C.

Les tableaux 2.6 et 2.7 donnent respectivement les spécificateurs de format pour l'instruction **printf** et **scanf**.

Format	Type	Affichage
%d ou %i	**int**	entier relatif
%u	**int**	entier naturel (unsigned)
%o	**int**	entier exprimé en octal
%x	**int**	entier exprimé en hexadécimal
%c	**int**	caractère
%f	**double**	reel en notation décimale
%e	**double**	reel en notation scientifique
%s	**char***	chaîne de caractères

Tableau 2.6 : Spécificateurs de format pour printf.

Formate	Type		Lecture d'un(e)
%d ou %i	**int***		entier relatif
%u	**int***		entier naturel (unsigned)
%o	**int***		entier exprimé en octal
%b	**int***		entier exprimé en hexadécimal
%c	**char***		caractère
%s	**char***		chaîne de caractères
%f ou %e	**float***		reel en notation décimale ou exponentielle (scientifique)

Tableau 2.7 : Spécificateurs de format pour scanf.

8. Série d'exercices -Algorithme séquentiel simple et traduction en langage C-

La solution de cette série d'exercice se trouve dans la partie Annexe de ce polycopié de cours.

Exercice 1. Parmi les propositions suivantes, lesquelles ne correspondent pas à des identificateurs valides ?

Hauteur, épaisseur, POIDS, niveau_d'eau, _Longueur, longueur_plus_2, 2_fois_longueur

Exercice 2. Ecrire un algorithme/ un programme en langage C qui affiche :

- le texte " le nombre est 12",
- le nombre 12 et son successeur,
- le texte "Le résultat du calcul de 323 moins 117 est X", où X est remplacé par le résultat de l'opération,

- le texte "Trois heures quinze minutes contiennent M minutes, ou S secondes", où M est remplacé par le nombre de minutes et S par le nombre de secondes.

Exercice 3. Soient les déclarations suivantes en lange C :

int n = 10, p = 4; long q = 2; float x = 1.75;

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

a) n + q	**b)** n + x	**c)** n % p +q	**d)** n < p	**e)** n >= p	**f)** n > q
g) q + 3 * (n > p)	**h)** q && n	**i)** (q-2) && (n-10)	**j)** x * (q==2)	**k)** x *(q=5)	

Exercice 4. Ecrire un algorithme/un programme en langage C qui affiche le texte "Veuillez entrer un nombre : ", attend que l'utilisateur entre un nombre entier au clavier et affiche ensuite le message : "Votre nombre est : ", suivi de la valeur du nombre.

Chapitre 3 Les structures conditionnelles

Ce chapitre est divisé cinq sections. La première et deuxième sections présentent l'instruction alternative et l'instruction conditionnelle composée. La troisième section montre la possibilité de choisir un traitement parmi plusieurs traitements possibles selon des cas précis. La quatrième section montre leur traduction en langage C. La cinquième section propose une série d'exercices sur les concepts traités dans ce chapitre.

1. Instruction alternative

Une instruction **alternative** permet d'influencer, suivant les données, sur la séquence d'instructions à exécuter. La figure 3.1 présente le format général d'une instruction alternative.

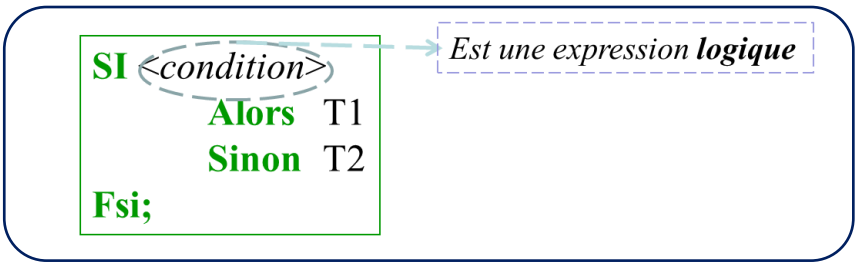


Figure 3.1 : Format général d'une instruction alternative

Remarques

- La partie **Sinon** peut être **omise** (enlevée), on parlera dans ce cas d'une **instruction conditionnelle**.
- T1 et T2 peuvent être n'importe quelles instructions.

Exemple

La figure 3.2 montre l'algorithme écrit en LA utilisant l'instruction alternative qui affiche le résultat de la comparaison de deux nombres entiers nbr1 et nbr2 saisis au clavier.

```
Algorithme comparaison;
var nbr1, nbr2: entier;
Debut
 ecrire('Entre un nombre:'); lire(nbr1);
 ecrire('Entrer un second nombre:'); lire(nbr2);
 Si nbr1>nbr2 alors ecrire(nbr1,' est supérieur à  ', nbr2);
                sinon si nbr1<nbr2 alors  ecrire(nbr1,' est inférieur à  ', nbr2);
                                          sinon ecrire(nbr1,' est égal à  ', nbr2);
                      fsi;
 fsi;
Fin.
```

Figure 3.2 : Exemple d'un algorithme écrit en LA utilisant une instruction alternative.

2. Instruction conditionnelle composée

Dans cette section nous allons détailler le cas où les traitements T1 et T2 de la figure 3.1 de la section précédente sont à leurs tours des instructions conditionnelles. On parle dans ce cas d'instructions conditionnelles imbriquées.

La figure 3.3 présente le format général d'une instruction conditionnelle composée.

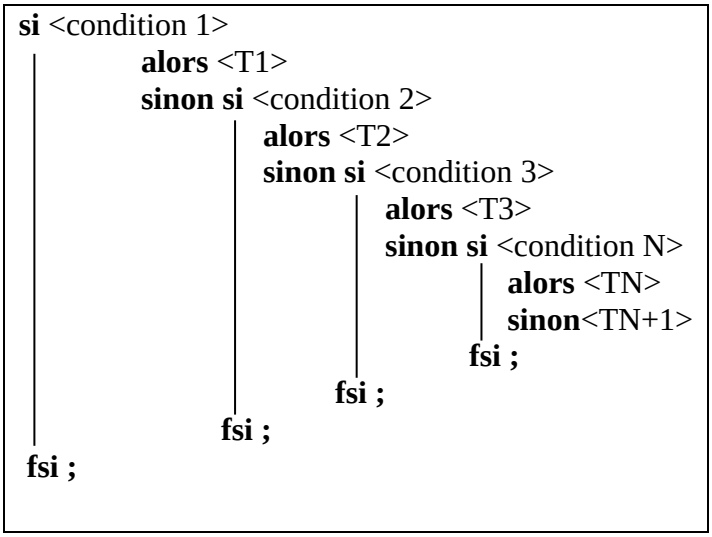


Figure 3.3 : Format général d'une instruction conditionnelle composée

3. Instruction à choix multiples

Une instruction à **choix multiples** permet de choisir, suivant les données, la séquence d'instructions à exécuter parmi plusieurs. La figure 3.4 présente le format général d'une instruction à choix multiples.

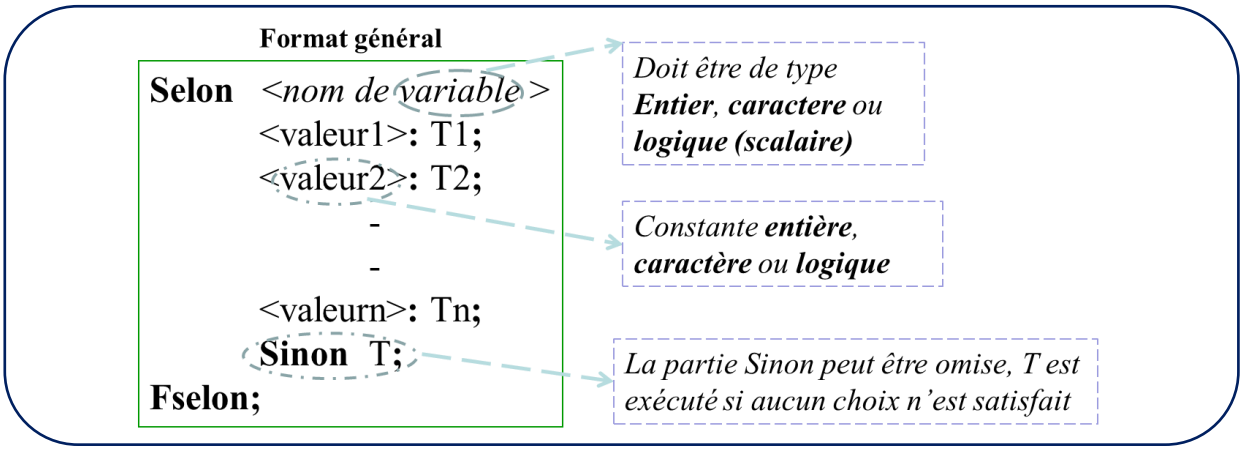


Figure 3.4 : Format général d'une instruction de choix multiples

Remarque

- Les parties T1, T2, …, Tn et T peuvent être n'importe quelles instructions.

Exemple

Supposons que les spécialités Informatique, Mathématiques, Physique et Chimie possèdent respectivement les codes de spécialité 1, 2, 3 et 4. L'algorithme écrit en LA décrit par la figure 3.4 utilise l'instruction à choix multiples pour afficher le nom de la spécialité qui correspond au code de la spécialité saisi au clavier.

```
Algorithme specialite;
var code_spec: entier;
Debut
 ecrire(‘Entrer  un code  compris entre 1 et  4):’); lire(code_spec);
 Selon  code_spec
          1: ecrire(‘Informatique’);
          2: ecrire(‘Mathématiques’);
          3: ecrire(‘Physique’);
          4: ecrire(‘Chimie’);
          sinon ecrire(‘ code  erroné ! ’);
 fselon;
Fin.
```

Figure 3.4 : Exemple d’un algorithme écrit en LA utilisant l’instruction de choix multiple.

4. Traduction en Langage C

Les concepts abordés dans ce chapitre ont touché principalement les instructions alternatives et de choix multiple. Dans cette section, nous donnons la traduction de ces concepts en langage C tel qu’illustré dans le tableau 3.1.

	LA	**Langage C**

Instruction conditionnelle	**si** <condition>
	  alors
	  <T1>
	  sinon
	  <T2>
	fsi ;
	si (x>y)
	  alors
	  max←x ;
	  sinon
	  max←y ;
	fsi ;
		`if ( <condition> )`
		`  <T1>`
		`else`
		`  <T2>`
		
		`if (x > y)`
		`  max = x;`
		`else`
		`  max = y;`

Instructions conditionnelles imbriquées	**si** <cond1>
	alors <T1>
	sinon si <cond2>
	alors <T2>
	sinon si <cond3>
	alors <T3>
	sinon si <condN>
	alors <TN>
	sinon<TN+1>
	fsi ;
	fsi ;
	fsi ;
	fsi ;
		`if (<expr logique1>)`
		`  <bloc1>`
		`else if (<expr2>)`
		`  <bloc2>`
		`else if (<expr3>)`
		`  <bloc3>`
		`else if (<exprN>)`
		`  <blocN>`
		`else <blocN+1>`

Instruction à choix multiples	**Selon** <*nom de variable* > <valeur1>**:** T1**;** <valeur2>**:** T2**;** **…** <valeurn>**:** Tn**;** **Sinon** T**;** **Fselon;**	**switch(**nom variable**)** **{** **case** valeur1**:** T1**;** **break;** **case** valeur2**:** T2**;** **break;** … **case** valeur2**:** T2**;** **break;** **default** :T**;** **}**

Tableau 3.1 : Traduction du LA vers le langage C -Instructions alternatives.

5. Série d'exercices -Instructions conditionnelles, composées et de choix multiple-

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

Exercice 1 :

Ecrire en LA un algorithme/ un programme en langage C qui à partir de deux nombres fournis par l'utilisateur affiche le signe de leur produit : 'négatif' ou 'positif' (ignorer le cas où le produit est nul).

Exercice 2

Ecrire en LA un algorithme/ un programme en langage C qui permet de trier par ordre décroissant trois nombres entiers en échangeant leurs valeurs.

Exercice 3

Ecrire en LA un algorithme/ un programme en langage C qui demande à un utilisateur de lui fournir deux nombres réels x et y ainsi que l'opération (à l'aide d'un caractère égal à + - * /) qu'il veut effecteur entre ces deux nombres. Le LA/programme C calculera et affichera le résultat de l'opération.

Chapitre 4 Les Boucles

Ce chapitre est divisé en six sections. La première section introduit la répétition des instructions dans LA. La deuxième, troisième et quatrième section expliquent le respectivement le fonctionnement des boucles Tant que, Répéter et Pour. La cinquième section montre leur traduction en langage C. La sixième section propose une série d'exercices sur les concepts traités dans ce chapitre.

1. Introduction

Pour répéter plusieurs fois une suite d'instructions, par exemple l'affichage des notes d'une classe comptant 100 étudiants, on devrait répéter 100 fois l'instruction d'affichage des notes respectives des étudiants.

En général, afin d'éviter la répétition d'une suite d'instruction, on utilise les **Boucles**. Ces dernières sont également appelées instruction répétitives ou itératives.

On en distingue trois, (1) la boucle **Tant que**, (2) la boucle **Repeter** et (3) la boucle **Pour**.

2. Boucle Tant que

Le format général d'une boucle **Tant que** en LA est donné par la figure 4.1.

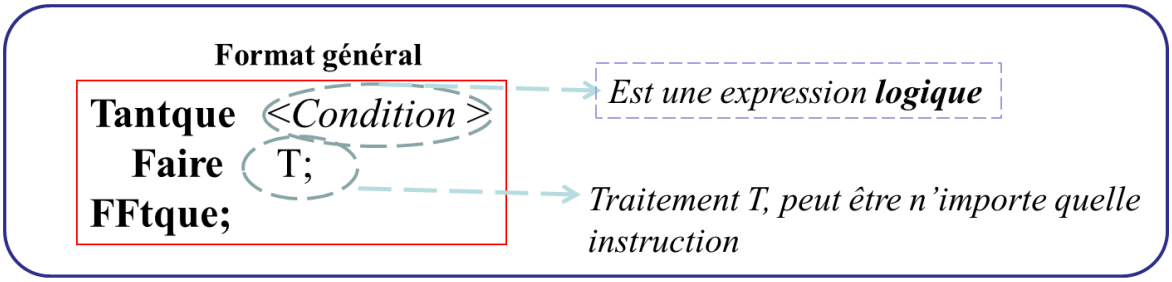


Figure 4.1 : Format général de la boucle Tant que

Principe de fonctionnement

- Tant que l'expression logique <*Condition*> est vraie, on exécute T.

Remarques

- Le **changement de condition** au niveau du traitement T est obligatoire, en d'autre termes, il faut que la condition devienne fausse à un moment donné, **sinon** on aura une **boucle infinie**.

- Si on est sûr que le traitement T est exécuté au moins une fois, il est préférable d'utiliser la boucle **Repeter**.

3. Boucle Répéter

Le format général d'une boucle **Répéter** en LA est donné par la figure 4.2.

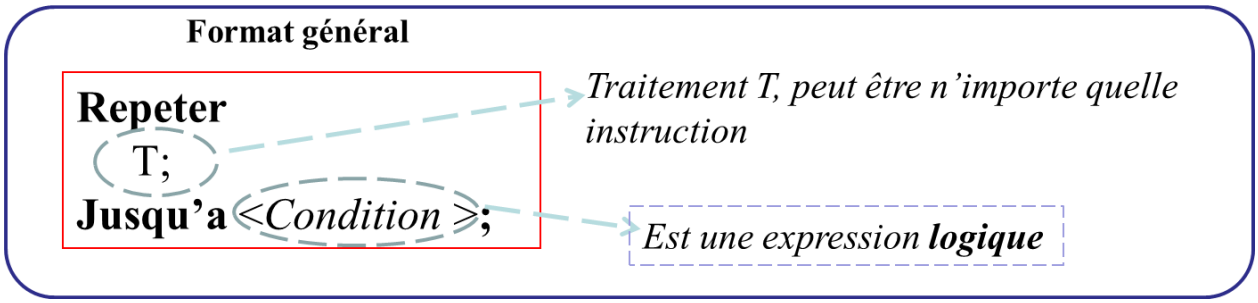


Figure 4.2 : Format général de la boucle Repeter

Principe de fonctionnement

- Le traitement T sera exécuté au moins une fois, on sortira de la boucle **Repeter** une fois l'expression logique <Condition> sera égale à vraie.

Remarques

- Le **changement de condition** au niveau du traitement T est obligatoire (il faut que la condition devienne vraie à un moment donné), **sinon** on aura une **boucle infinie**.

- Si le nombre de répétition du traitement T est connu, on utilise la boucle **Pour**.

4. Boucle Pour

Le format général d'une boucle Pour en LA est donné par la figure 4.3.

Format général

Pour <*identificateur*>⟵ <*valeur initiale*> **jusqu'a** <*valeur finale*> [**par pas de** <*valeur*>]
Faire T;
Ffpour,

Figure 4.3 : Format général de la boucle Pour

Principe de fonctionnement

- Cette forme permet de parcourir un ensemble de valeurs entières avec un pas donné.
- Le traitement T sera exécuté pour <*identificateur*> (une variable compteur, généralement entière), variant de <*valeur initiale*> jusqu'à <*valeur finale*>.

Remarques

- Il ne faut pas que le traitement T modifie la valeur de la variable compteur étant donné que cette dernière est incrémentée / décrémentée d'une valeur égale au pas pour chaque itération de la boucle Pour.
- Le pas n'est pas précisé quand il est égal à 1.

Exemple

Cet exemple se veut un exemple récapitulatif, utilisant à la fois les boucles Tant que, Repeter et Pour dans l'écriture en LA d'un même traitement.

La figure 4.4 montre l'utilisation des instructions répétitives Tant que, Repeter et Pour dans l'algorithme écrit en LA permettant de calculer et d'afficher le résultat de la somme de 1 jusqu'à n (n est un entier saisi au clavier). *Au cas où n=5 l'exécution du LA calculera et affichera la somme 1+2+3+4+5.*

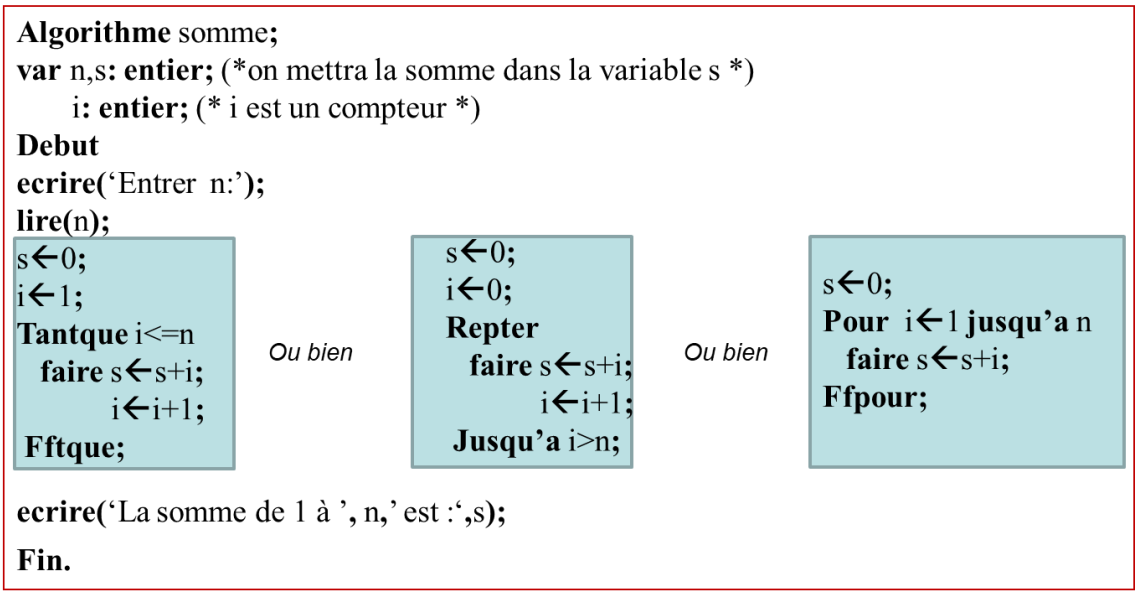


Figure 4.4 : Exemple d'un LA qui calcule la somme de 1 jusqu'à n utilisant Tant que, Repeter et Pour.

5. Traduction en langage C

Les concepts abordés dans ce chapitre ont touché principalement les instructions de Boucle : Tant que, Répéter et Pour. Dans cette section, nous donnons la traduction de ces concepts en langage C tel qu'illustré dans le tableau 4.1.

	LA	**Langage C**

Instruction de boucle Tant que	**Tantque** <*Condition* >
	Faire T;
	FFtque;
	
	(* Afficher les nombres de 0 à 9*)
	var I : entier ;
	debut
	I←0 ;
	Tant que I<10
	Faire ecrire(I, ' '**) ;**
	I←I+1 ;
	FFTque ;
		while (<condition>)
		T ;
		
		/* Afficher les nombres de 0 à 9 */
		int I = 0;
		while (I<10)
		{
		printf("%i ", I);
		I=I+1;
		}

Instruction de boucle repeter	**Repeter** T; **Jusqu'a** <*Condition* >**;**	**do** T; **while (<condition>);**
Instruction de boucle pour	**Pour** <*var*> ← <*val1*> **a** <*val2*> **par pas de** *val* **Faire** T; **Ffpour,**	**for (**<*var*=*val1*>; <*val1*<=*val2*>; *var*=*var*+*val*>**)** T ;

Tableau 4.1 : Traduction du LA vers le langage C -Instructions de boucle.

6. Série d'exercices -Instructions de Boucle-

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

Exercice 1

Écrire un algorithme en LA/Programme C qui détermine si un nombre est premier ou non. Il affichera si le nombre est premier ou non, en indiquant dans ce dernier cas par quel nombre il est divisible.

Exercice 2

Les populations respectives de deux villages sont A= 210 personnes et B = 280 personnes. Chaque année les populations de A et de B augmentent respectivement de 8% et de 5%. Écrire un algorithme en LA/Programme C qui calcule le nombre d'années nécessaires pour que A dépasse B. Généraliser pour des populations A et B quelconques.

Exercice 3

Écrire un algorithme en LA/Programme C qui détermine (1) le nombre de chiffres dans un entier positif et (2) le maximum de ces chiffres

Chapitre 5 Les structures de données statiques

Ce chapitre est divisé en cinq sections. La première section présente la notion de type, à savoir le type scalaire, le type énuméré et le type intervalle. La deuxième et troisième sections abordent respectivement les tableaux à une et deux dimensions, communément appelés vecteurs et matrices. La quatrième section présente le type chaine de caractères. La cinquième section montre leur traduction en langage C. La cinquième section propose une série d'exercices sur les concepts traités dans ce chapitre.

1. Notion de type

Un type définit l'ensemble des valeurs que peut prendre une donnée.

Nous définissons deux catégories de types en LA :

- Types **standards** (entier, logique, caractere, reel)
- Types **définis par le programmeur**, qui **font l'objet d'une définition dans la partie déclaration** du LA, comme le montre la figure 5.1.

Algorithme <*identificateur*>**;**
<*déclaration de constantes*>
<*définition de types*>
<*déclaration de variables*>
<*déclaration de procédures et de fonctions*>
Debut
<*Instructions exécutables*>
Fin.

Figure 5.1 : Définition de types dans la partie déclaration du LA

a. Type scalaire

Les types **scalaires** constituent des ensembles de **valeurs** finies et linéairement ordonnées.

Un type est dit scalaire s'il est :

- ✓ **scalaire standard** (entier, logique, caractere)
- ✓ **énuméré**
- ✓ **intervalle**

La figure 5.2 montre l'organisation des types scalaires

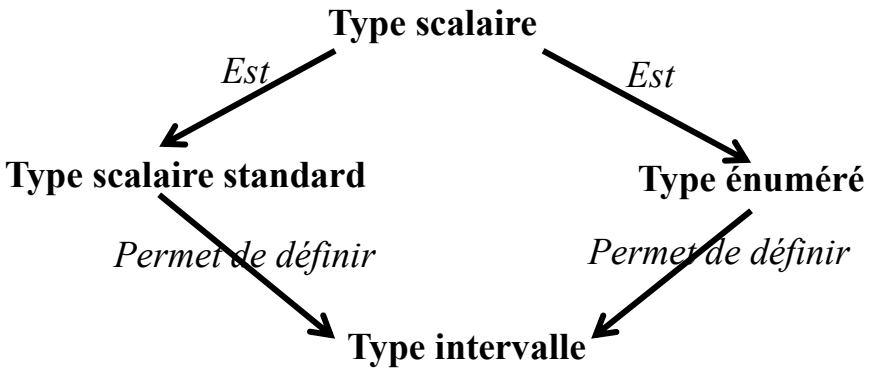


Figure 5.2 : Organisation des types scalaires

b. Type énuméré

Définir un type énuméré consiste à lister dans l'ordre toutes les valeurs possibles représentées par des identificateurs et qui seront les constantes de ce type.

La figure 5.3 donne le schéma syntaxique de la définition d'un type énuméré.

type *<identificateur de type>* **=(***<identificateur>* {**,***<identificateur>*}**);**

Figure 5.3 : Schéma syntaxique de la définition d'un type scalaire

Exemple

type logique = **(**faux, vrai**);**
couleur = **(**rouge, vert, noir, gris**);**
genre = **(**masculin, feminin**);**
} **définition du type scalaire**

var l**:** logique**;**
c**:** couleur**;**
g**:** genre**;**
} **déclaration de variables de type scalaire**

Les opérations de relation **>, <, >=, <=, =, <>** sont valables pour les données de n'importe quel type scalaire. Pour l'exemple précédent, on peut écrire l'opération : *masculin* < *feminin* dont le résultat est la valeur logique : Vrai.

Les fonctions standards déjà vues **pred**, **succ** et **ord** s'appliquent aux arguments (*valeurs représentées par des identificateurs*) de type scalaire.

Pour l'exemple précédent :

- s ← **pred**(feminin) donc s reçoit la valeur masculin
- c ← **succ**(rouge) donc c reçoit la valeur vert
- La fonction **ord** de la première valeur du type scalaire = 0 donc **ord**(faux) vaut 0.

c. Type intervalle

Un type **intervalle** est défini comme un intervalle d'un type scalaire déjà défini. Il est nécessairement un sous-type d'un type scalaire (standard ou énuméré) déjà défini. La figure 5.4 montre le schéma syntaxique de la définition d'un type intervalle.

type *<identificateur de type>* **=** *<constante1>* **..** *<constante2>***;**

Figure 5.4 : Schéma syntaxique de la définition d'un type intervalle.

Remarque

Notons que *constante1* et *constante2* sont des constantes de type scalaire énuméré ou type scalaire standard telles que *constante1* < *constante2*.

Exemple

type mois = **(**janvier, fevrier, mars, avril, mai, juin, juillet, aout, septembre, octobre, novembre, decembre**);**
ete = juin **..** aout**;**
lettre = 'a' **..** 'z'**;**
chiffre = 0 **..** 9**;**

2. Les tableaux à une dimension (Vecteurs)

Situation du problème

Supposons qu'on désire charger en mémoire centrale toutes les notes des étudiants pour pouvoir les traiter. Peu importe le traitement qu'on souhaite effectuer sur ces notes.

Si on a par exemple 100 étudiants, on devrait déclarer 100 variables ayant toutes des noms différents conformément à la règle de la non-duplication des identificateurs des variables dans le LA. Une telle situation représente une tâche quasi impossible pour le programmeur.

⇨ Une des solutions possibles serait d'utiliser les Tableaux.

a. Présentation des vecteurs

Un tableau à une dimension ou communément appelé **vecteur** se compose d'un **nombre fixe d'éléments** de **même type (***type de base***)**.

Le nombre d'éléments du vecteur est appelé la **taille** du vecteur.

Le type des éléments du vecteur peut être quelconque (entier, réel, logique ou caractère).

Chaque élément du vecteur est repéré par un **indice** qui indique sa position dans le vecteur.

Exemple

La figure 5.5 montre un exemple d'un vecteur représentant les notes obtenues par 6 étudiants. La dimension du tableau est égale à 1, le tableau est donc un vecteur. Sa taille est égale à 6 et le type de base de ses éléments est : réel. 1, 2, …,6 sont les indices du vecteur.

14.75	11.00	8.00	14.75	10.00	7.25
1	2	3	4	5	6

Figure 5.5 : Exemple d'un vecteur représentant les notes obtenues par 6 étudiants.

b. Déclaration d'un vecteur

La figure 5.6 montre le schéma syntaxique de la déclaration d'un vecteur.

type <*identificateur de type*> **= tableau [**T1**] de** T2**;**

Figure 5.6 : Schéma syntaxique de la déclaration d'un tableau

Où:

- **T1** est un type **intervalle**
- **T2** est **n'importe quel type**

Exemple

Déclaration d'un vecteur contenant les notes de 6 étudiants.

```
type  indice = 1..6;
      Tnote = tableau [indice] de reel;

var   note: Tnote;
```

La déclaration du vecteur **note** se traduit par une réservation en MC de 6 variables de type réel. Le nom de chacune de ses variables est composé du nom de la variable vecteur suivi entre crochets de son indice correspondant dans le vecteur.

La figure 5.7 montre l'état de la MC à la suite de cette déclaration.

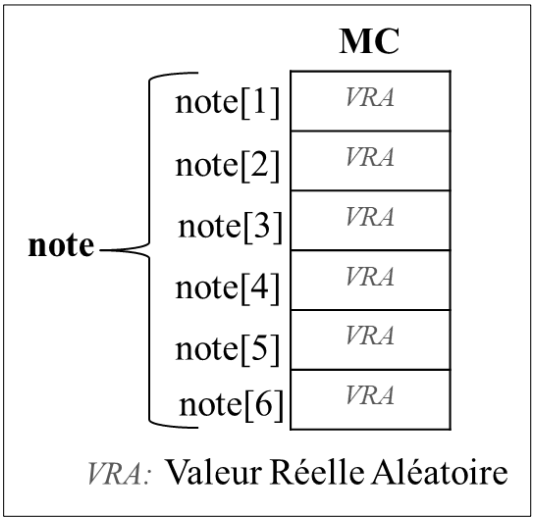


*Figure 5.7 : Etat de la MC suite à la déclaration d'un vecteur de réels **note** de taille égale à 6.*

Remarque

Les indices d'un tableau ne sont pas forcément des entiers, comme le montre l'exemple suivant :

```
Type  valeur = 97..122;
      Tascii_lettre_min = tableau ['a' .. 'z'] de valeur;
Var   ascii_lettre_min : Tascii_lettre_min ;
```

ascii_lettre_min	97	98	99	.....	121	122
	'a'	'b'	'c'		'y'	'z'

ascii_lettre_min [‘a’] contient la valeur 97, **ascii_lettre_min [‘b’]** contient la valeur 98, …, **ascii_lettre_min [‘z’]** contient la valeur 122.

c. Opérations sur les vecteurs

Parmi les opérations élémentaires qu’on peut faire sur un vecteur on peut citer la recherche du maximum, du minimum ou bien la moyenne des éléments d’un vecteur.

Ceci se traduit respectivement par des exemples concrets tels que la recherche de la plus faible note, de la meilleure note ou bien de la moyenne des notes obtenues par les étudiants lors d’un examen de fin de semestre.

Exemples

Ecrire en LA l’algorithme permettant rechercher la plus faible note obtenue dans un examen comptant 100 étudiants.

```
Algorithme Plus_faible_note;
  const n=100;
  type  indice = 1..n;
        Tnote = tableau [indice] de reel;
  var   note: Tnote;  i: entier;   note_min:reel;

 Debut

    (*Saisie des notes d’examen des étudiants*)
  Pour i ←1 jusqu’à n
   faire ecrire(‘Entrer la note de l’’étudiant ‘,i,’ :’);
         lire(note[i]);
   ffpour;

    (*Recherche de la plus faible note*)
  note_min← note[1];
  Pour i ←2 jusqu’à n
   faire si note_min > note[i]
           alors note_min← note[i];
         fsi
   ffpour;

  Ecrire (‘La plus faible note des ’,n,’ étudiants est :’, note_min);
Fin.
```

Nous reprenons l’exemple précédent mais en recherchant la position de la plus faible note dans le vecteur.

```
Algorithme Plus_faible_note;
   const n=100;
   type  indice = 1..n;
         Tnote = tableau [indice] de reel;
    var   note: Tnote;  i,pos_min : entier;

Debut

      (*Saisie des notes d'examen des étudiants*)
    Pour i ←1 jusqu'à n
     faire ecrire('Entrer la note de l''étudiant ',i,' :');
           lire(note[i]);
     ffpour;

      (*Recherche de l'indice de la plus faible note*)
    pos_min← 1;
   Pour i ←2 jusqu'à n
    faire si note[pos_min] > note[i]
              alors pose_min← i;
             fsi
    ffpour;

   Ecrire ('Parmi les ', n, ' étudiants, l'' étudiant ', pos_min, ' a eu la plus faible note  égale
              à  ', note[pos_min]);
Fin.
```

d. Quelques algorithmes de base dans un vecteur

Algorithme de recherche

La recherche d'un élément précis dans un vecteur qui peut se traduire concrètement par la recherche d'un étudiant particulier à partir de son numéro d'inscription.

Etape 1 : On saisit les éléments du vecteur

Etape 2 : On saisit l'élément à rechercher

Etape 3 : On compare l'élément à rechercher avec l'élément courant du vecteur

Etape 4 : Si l'élément recherché est égal à l'élément courant du vecteur, on relève son indice et on va à l'étape 6.

Etape 5 : Si on n'a pas encore parcouru tous les éléments du vecteur on revient à l'étape 3.

Etape 6 : On affiche l'indice de l'élément recherché dans le vecteur dans le cas où on l'a relevé, ou bien le message *"élément recherché n'existe pas !"* dans le cas contraire

Algorithme de tri

Un algorithme de tri dans un vecteur est un algorithme qui permet d'organiser les éléments du vecteur selon un ordre déterminé.

Les éléments à trier font donc partie d'un ensemble muni d'une relation d'ordre.

Parmi les algorithmes de tri les plus utilisés on citera : Tri par sélection , Tri par bulle (propagation) et Tri par permutation (insertion)

3. Les tableaux à deux dimensions

Situation problème

Supposons qu'on désire charger en mémoire centrale les notes des deux contrôles et la note de l'examen d'une classe comptant 100 étudiants.

Plusieurs scénarios s'offrent à nous, On peut penser par exemple à :

- 100 vecteurs ayant chacun 3 éléments réels. (A ne pas retenir),
- ou encore à 3 vecteurs comportant chacun 200 éléments réels.

Pour ce faire, reprenons le schéma syntaxique de la déclaration d'un tableau (voir figure x.x) et intéressons-nous au cas où T2 est de type vecteur.

On aura ainsi un vecteur dont les éléments sont à leur tour des vecteurs. Autrement dit un vecteur de 3 éléments dont chaque élément est un vecteur comportant 100 éléments réels, ou l'inverse un vecteur de 200 éléments dont chaque élément est un vecteur comportant 3 éléments réels.

Une telle structure de données est appelée **Matrice**.

a. Présentation d'une matrice

Un tableau à deux dimensions ou communément appelé **matrice** se compose d'un **nombre fixe d'éléments** de **même type (***type de base***)**.

Le nombre d'éléments de la matrice est appelé la **taille** de la matrice.

La déclaration d'une matrice reste la même que celle d'un tableau :

Exemple

Déclaration pour le cas de la situation problème.

```
type  ind_col = 1..3;
      Tnotes= tableau [ind_col] de reel;
      ind_Ligne = 1..200;
      Tstruct_note = tableau [ind_Ligne] de Tnotes;

var   note: Tstruct_note;
```

La figure 5.8 montre l'état de la MC à la suite de cette déclaration.

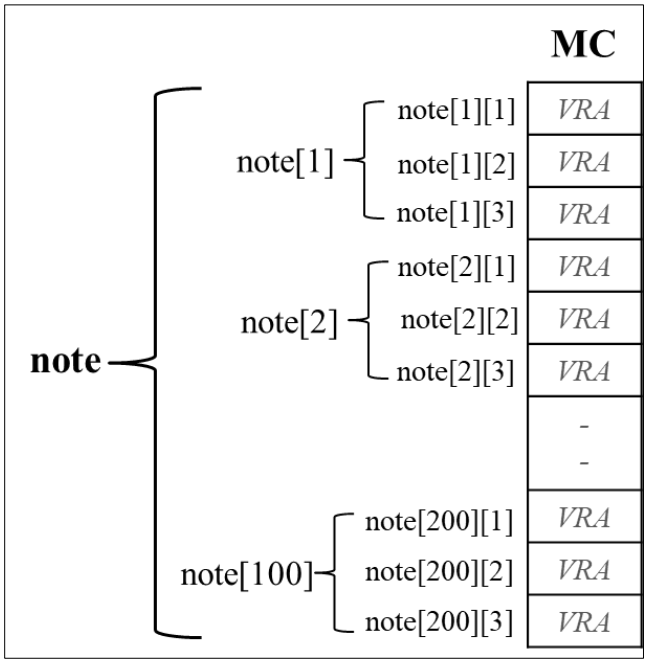


*Figure 5.8 : Etat de la MC suite à la déclaration d'une matrice de réels **note** de taille égale à 100 x 3.*

Remarque

On peut également déclarer une matrice d'une manière plus simple :

type ind_Ligne = 1..200**;** ind_col = 1..3**;**

Tstruct_note **= tableau [**ind_ligne,ind_col**] de** reel**;**

var note**:** Tstruct_note**;**

La figure 5.9 montre une telle déclaration

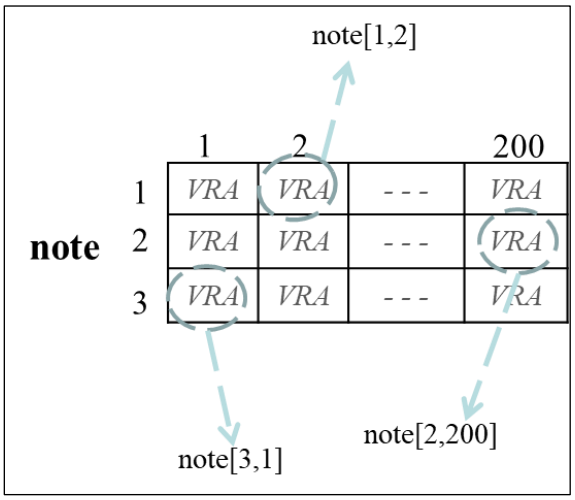


Figure 5.9 : Autre manière de représenter une matrice

Ou encore :

```
const     ligne=200; col=3;
type      Tstruct_note = tableau [1..ligne, 1..col] de Tnotes;

var   note: Tstruct_note;
```

b. Opérations sur les matrices

Il existe plusieurs opérations mathématiques qu'on peut faire sur une matrice, on peut citer l'addition, le produit de deux matrices, le déterminant, l'inverse d'une matrice etc... Et aussi d'autres opérations comme la recherche du maximum et du minimum, le tri par ligne, par colonne, etc.

Exemple

Ecrire en LA l'algorithme qui saisit les notes des deux contrôles et la note de l'examen d'une classe comptant 100 étudiants.

```
Algorithme Saisie_notes;
   const ind_ligne=100; ind_col=3;
   type Tnote = tableau [1..ind_ligne,1..ind_col] de reel;
    var   note: Tnote;
              i,j: entier;
  Debut
    Pour i←1 jusqu'à ind_ligne
     faire  ecrire('Entrer les notes de l''étudiant ',i,' :');
            Pour j←1 jusqu'à ind_col
              faire ecrire('Note ',j,': ');
                    lire(note[i,j]);
              ffpour;
     ffpour;
  Fin.
```

4. Les Chaines de caractères

a. Présentation

Le LA permet d'avoir des chaines de caractères de longueurs variables qui varient entre 0 (chaine vide) et une longueur maximale précisé par le programmeur qui généralement ne dépasse pas 256. La figure 5.10 montre le schéma syntaxique de la déclaration d'une chaine de caractères.

type <*identificateur de type*> **= chaine de caractere [cst];**

Figure 5.10 : Schéma syntaxique de la déclaration d'une chaine de caractères

Où:

- **cst** est une constante entière comprise entre 0 et 256
- Si **cst** n'est pas mentionnée, la longueur de la chaine de caractères est par défaut égale à 256.

Exemple

Déclaration de deux variables chaines de caractères censées contenir respectivement le nom et le prénom d'une personne.

```
type    Tnom = chaine de caractere [20];
        Tadresse= chaine de caractere ;

var   nom: Tnom; adresse: Tadresse;
```

Par analogie aux vecteurs une chaine de caractères est un vecteur dont les éléments sont des caractères. On peut ainsi accéder vers un élément de la chaine en utilisant l'indice.

Exemple

Déclaration d'une chaine de caractères en utilisant un vecteur.

```
type  long= 1..20;
      Tnom = tableau [1..long] de caractere;
var   nom, pseudo: Tnom;

nom[1] ←'S'; (* affectation du caractère S à la variable chaine de caractère nom*)
```

Les constantes de type **chaine de caractère** sont les suites de caractères entre apostrophes ''.

b. Opérations sur les chaines de caractères

Parmi les opérations qu'on peut faire sur les chaines de caractères, nous pouvons citer :

1. **Affectation** : nom ←'Mohamed'; pseudo←nom;
2. **Concaténation** : elle est donnée par le signe plus +
   - pseudo ← 'alias_'+ pseudo; ➔ dans pseudo il y aura : 'alias_Mohamed'
3. **Opérateurs de comparaison**: Les chaines sont comparées caractère par caractère de gauche à droite.
   - Deux chaines sont égales si elles ont la même longueur et le même contenu
   - Si une chaine x appartient à une autre chaine y alors x<y
   - 'ab'<'abb' 'aa'<'ab' '12'<'2'

5. Traduction en langage C

Les concepts abordés dans ce chapitre ont touché principalement la notion de type, les tableaux à une et deux dimensions, communément appelés vecteurs et matrices et le type chaine de caractères. Dans cette section, nous donnons la traduction de ces concepts en langage C tel qu'illustré dans le tableau 5.1.

	LA	Langage C
Définition de type	**type** <*ident. de type*> = <*type*> **;**	**typedef** <*type*> <*ident. de type*> ;
Type énuméré	**type** <*ident. de type*> **=(**<*ident.*> **,**<*ident.*>,… <*ident.*>**);** **var** <*ident.*> : <*ident. de type*> **type size** = **(**BIG, MIDLLE, SMALL**);** **var** s**: size;**	**typedef enum {**<*ident.*>,<*ident.*>,… <*ident.*>**}** <*ident. de type*>; <*ident.*> <*ident. de type*>; **typedef enum {**BIG, MIDLLE, SMALL**}** size; **size** s;
Type tableau à une dimension (Vecteur)	**var** <ident> **: tableau [**1**..**taille**]** de <**type**> **;** **var** t **:tableau[**1**..**3**] de entier ;**	<type> <ident>**[**taille**];** **int** t**[**3**]** ; **/***Initialisation du vecteur lors de la déclaration ***/** **int** t**[**3**] = {**-4,6,12**};** **/*** ceci veut dire que **t[**0**]** vaut −4 , **t[**1**]** vaut **6 et t[**2**]** vaut **12 */**
Type tableau à deux dimensions (Matrice)	**var** <ident> **: tableau [**1**..**nbl**,** **1..**nbc**]** de <**type**> **;** **var** t **:tableau[**1**..**3**,** 1**..**2**] de entier ;**	<type> <ident>**[**nbl**] [**nbc**];** **int** t**[**3**][**2**]** ;
Type chaine de caractères.	**var** chaine **:tableau[**1**..**long**] de caractere ;**	**char** chaine[long]; **/***Initialisation d'une chaine de caractères ***/** **char** chaine**[]** = "Bonjour"; // ou bien **char** chaine**[] = {** 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' **};**

Tableau 5.1 : Traduction du LA vers le langage C -Type énuméré, Tableaux et chaines de caractères.

6. Série d'exercices – Tableaux et Chaines de caractères -

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

Exercice 1

Écrire un algorithme en LA/Programme C qui permet de saisir la taille d'un tableau à une dimension et d'afficher les éléments de ce vecteur.

Exercice 2

Écrire un algorithme en LA/Programme C qui permet de mettre à zéro tous les éléments d'une matrice, de l'afficher, de reprendre la saisie des éléments de la matrices et de faire la somme de ses éléments.

Chapitre 6 Les types personnalisés – Les enregistrements.

Ce chapitre est consacré au type enregistrement. Pour ce faire, il est divisé en trois sections. La première section est présente le type enregistrement, à savoir, sa définition et sa déclaration. La deuxième section montre la possibilité de définir un tableau dont les éléments sont des enregistrements, on parlera dans ce cas de tableau d'enregistrements. La troisième section présente l'accès aux champs d'une variable enregistrement. La quatrième section montre leur traduction en langage C.

1. Le type enregistrement

a. Définitions

Un enregistrement est une structure composée d'un nombre fini d'éléments (appelés **Champs**) qui peuvent être de types différents. Il permet de regrouper dans un même type l'ensemble des informations caractéristiques d'un un objet déterminé. C'est un type personnalisé dans le sens où il est défini par le programmeur selon ses propres besoins.

Remarque

A la différence d'un tableau dont tous les éléments sont de même type, l'enregistrement offre la possibilité de contenir dans une même structure de données des éléments de type différents. Cette possibilité permet de représenter en machine des entités ou objets du monde réel (Personnes, Animaux, livres, etc…).

b. Déclaration

Avant de déclarer (réserver en MC) une variable de type enregistrement, nous devons (i) définir son type enregistrement ensuite (ii) procéder à sa déclaration .

i. Définition du type Enregistrement

La figure 6.1 montre comment définir le type enregistrement.

```
Type  <nom_enregistrement> =   Enregistrement
                                    Champ 1 : Type 1 ;
                                    - -          - -
                                    Champ n : Type n ;
                               Fin Enregistrement ;
```

Figure 6.1 : Définition du type enregistrement.

Où :

Type 1, … Type n peuvent être de types différents (éventuellement de type enregistrement).

ii. Déclaration de la variable du type Enregistrement

La figure 6.2 montre comment déclarer le type enregistrement.

```
var  <nom_var enregistrement> : <nom_enregistrement> ;
```

Figure 6.2 : Déclaration du type enregistrement.

Exemples

1. **Déclaration simple.** Dans ce cas de figure les Type 1,… , Type n sont des types simples ou chaines caractères.

```
Type Tetudiant  =        Enregistrement
                         num_inscr : chaine de caractere [8] ;
                         nom, prenom : chaine de caractere [15] ;
                         date_nais : chaine de caractere [10] ;
                         adresse : chaine de caractere [40] ;
                         resultat : logique ;
                    Fin Enregistrement ;
Var Vetud : Tetudiant ;
```

2. **Déclaration faisant appel à des types Enregistrement d'Enregistrements**

```
Type Tdate  = Enregistrement
                    j: 1..31 ;
                    m: 1..12 ;
                    a : integer ;
               Fin Enregistrement ;
      Tadresse  = Enregistrement
                    num: entier ;
                    Rue, ville : chaine de caractere [12] ;
                  Fin Enregistrement ;
      Tetudiant  = Enregistrement
                    num_inscr : chaine de caractere [8] ;
                    nom, prenom : chaine de caractere [15] ;
                    date_nais : Tdate ;
                    adresse : Tadresse ;
                    resultat : logique ;
                  Fin Enregistrement ;
Var Vetud : Tetudiant ;
```

2. Déclaration d'un vecteur d'enregistrements

Supposons que l'on souhaite déclarer 100 étudiants. On procédera alors comme suit :

```
Const n= 100;
Type Tdate  = Enregistrement
                    j: 1..31 ;
                    m: 1..12 ;
                    a : integer ;
               Fin Enregistrement ;
      Tadresse  = Enregistrement
                    num: entier ;
                    Rue, ville : chaine de caractere [12] ;
```

```
                Fin Enregistrement ;
    Tetudiant  = Enregistrement
                        num_inscr : chaine de caractere [8] ;
                        nom, prenom : chaine de caractere [15] ;
                        date_nais : Tdate ;
                        adresse : Tadresse ;
                        resultat : logique ;
                Fin Enregistrement ;
    Ttab_etud : Tableau [1..n] de Tetudiant;

Var  Vtab_etud :  Ttab_etud ;
```

3. Accès aux champs d'une variable de type enregistrement

Les champs d'une variable de type enregistrement s'utilisent comme n'importe quelle variable mais ils sont précédés du nom de la variable enregistrement et d'un point.

On peut également utiliser l'instruction : **avec** comme suit :

```
avec  <var enregistrement> {, <var enregistrement>}
    faire
            <instructions> ;
    FFavec ;
```

Exemples

1. En utilisant la variable enregistrement et le point

```
lire(Vetud.nom) ;   lire(Vetud.dn.j)
```

2. En utilisant l'instruction **avec**

```
avec  emp, dn
    faire
        lire(nom) ;
        lire(j) ;
    fin ;
```

Remarque

La seule instruction possible qu'on peut faire sur un enregistrement entier (tout l'enregistrement) est l'instruction d'affectation d'un enregistrement vers un autre enregistrement de même type.

4. Traduction en langage C

Les concepts abordés dans ce chapitre ont touché principalement au type enregistrement et aux tableaux d'enregistrements. Dans cette section, nous donnons la traduction de ces concepts en langage C tel qu'illustré dans le tableau 6.1.

	LA	Langage C
Définition de type enregistrement	**type** <*ident.TEnrg.*>**=Enregistrement** <*champ1*>**:**<*type*>**;** … <*champN*>**:**<*type*>**;** **FinEnregistrement;** **var :**<*ident.VarEnrg.*>**:**<*ident.TEnrg.*> **type tComplexe = Enregistrement** re : reel **;** im : reel **;** **Fin Enregistrement;** **var :** complexe **: tComplexe ;**	**typedef struct {** <*type*> <*champ1*>; … <*type*> <*champ1*>; **}** <*ident.TEnrg.*>; <*ident.TEnrg.*> <*ident.VarEnrg.*>; **typedef struct {** float re; float im; **} tComplexe;** **tComplexe** complexe ;

Tableau 6.1 : Traduction du LA vers le langage C -Type enregistrement.

5. Série d'exercices – Enregistrements et tableaux d'enregistrements -

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

Exercice 1

Ecrire un algorithme en LA/programme en langage C permettant de saisir des données concernant deux personnes, de calculer et d'afficher leur différence d'âge, sachant qu'une personne est caractérisée par son nom, son prénom et son âge.

Exercice 2

Un étudiant est défini par les informations suivantes : son identificateur (entier), son nom (chaine de 30 car), le numéro de groupe auquel il appartient (1...4) et sa une note comprise entre 0 et 20.

Ecrire un algorithme en LA/programme en langage C qui permet de lire les informations relatives à 10 étudiants d'une section, puis rechercher et afficher les noms et les prénoms de tous les étudiants du groupe 2 ayant une note < 10.

PARTIE II Algorithmique et Structure De Données 2 (ASDD2)

Cette deuxième partie a pour objectif de permettre à l'étudiant d'acquérir les notions fondamentales de la programmation. Les Connaissances préalables recommandées sont l'algorithmique et les structures de données. Ces connaissances ont été abordées dans la partie 1 de ce manuscrit.

Pour atteindre cet objectif cette partie se décline en trois chapitres :

CHAPITRE 1 : Les sous-programmes : Fonctions et Procédures
CHAPITRE 2 : Les Fichiers
CHAPITRE 3 : Les Listes chainées

Chapitre 7 Les sous-programmes : Fonctions et Procédures

Ce chapitre est divisé en six sections. La première section est une introduction à la notion de sous-programmes. La deuxième section donne les définitions se rapportant à cette notion, notamment celles de procédures et de fonctions. La troisième section appréhende la notion de variables locales et variables globales. Il y sera abordé également les blocs, leurs structures en Langage Algorithmique (LA) et en Langage C ainsi que la portée des variables déclarées au niveau de ces blocs. La quatrième section définit les paramètres utilisés dans les sous-programmes ainsi que les règles de passage des paramètres par valeur et par adresse. La section cinq aborde l'utilisation de la récursivité dans les sous-programmes. Finalement, la section six propose une série d'exercices se rapportant à ce chapitre.

1. Introduction

Afin d'introduire la notion de sous-programmes, nous partons des deux exemples suivants :

Exemple 1. Soient a, b et c des entiers naturels, donner l'algorithme qui calcule : (a! + b!) / c!

Exemple 2. Soient A et B deux entiers naturels, donner l'algorithme qui permet d'obtenir un nombre C égal à la concaténation des deux nombres A et B. Par exemple, si A=13 et B=904 le résultat de la concaténation serait le nombre C=13904.

a. Cas de l'exemple 1

En ce qui concerne l'Exemple 1, tel qu'illustré par l'algorithme de la Figure 7.1, on répète pratiquement 3 fois la même séquence d'instructions du calcul de la factorielle à des endroits différents du corps de l'algorithme.

```
algorithme expression_factorielle;
var a,b,c,i,f: entier;
      exp: reel ;
debut
     lire (a,b,c) ;
     (*calcul de a!*)
     f←1 ;
     pour i←2 jusqu'à a
        faire
           f←f*i ;
        ffpour ;
     exp←f ;

     (*calcul de b !*)
     f←1 ;
     pour i←2 jusqu'à b
        faire
          f←f*i ;
        ffpour ;
     exp←exp+f ;

     (*calcul de c!*)
     f←1 ;
     pour i←2 jusqu'à c
        faire
          f←f*i ;
        ffpour ;
     exp←exp /f;
fin.
```

Figure 7.1 : Algorithme de (a! + b!) / c!

La question qui se pose est : Comment procéder pour éviter la répétition d'une même séquence d'instructions ?

Pour répondre à cette question nous procédons en deux étapes :

Etape 1. Nous définissons une entité de données et d'instructions, qu'on appellera **module** permettant de calculer n'importe quelle factorielle d'un nombre entier n donné et de récupérer le résultat du calcul.

Etape 2. Nous faisons par la suite appel à ce module pour le calcul de a!, de b! et de c!.

b. Cas de l'exemple 2

En ce qui concerne l'Exemple 2, nous nous trouvons en face d'un problème assez complexe.

La question qui se pose est : Comment réduire (diminuer) cette complexité ?

Pour répondre à cette question nous procédons en deux étapes :

Etape 1. Nous définissons trois modules, à savoir : Module 1, Module 2 et Module 3 où chaque module sera responsable des fonctionnalités suivantes :

- Module 1 : Compter le nombre de chiffres du nombre B et récupérer le résultat du calcul.
- Module 2 : Calculer 10 à la puissance le nombre de chiffres de B et récupérer le résultat du calcul.
- Module 3 : Calculer C= A x (10 à la puissance le nombre de chiffres de B) + B et récupérer le résultat du calcul.

Etape 2. Nous faisons par la suite appel à ces trois modules pour calculer le nombre C égal à la concaténation des deux nombres A et B.

c. Conclusions

Des deux exemples précédents, nous pouvons dire que ce procédé :

1. Permet :
   a) D'éviter de réécrire plusieurs fois dans le corps d'un algorithme une même séquence d'instructions.
   b) De réduire la complexité d'un problème complexe.

2. Se réalise en deux étapes :
   a) Définition du/des module(s)
   b) Appel du/des module(s)

2. Définitions

Un s/s programme est un module (entité de données et d'instructions) qui permet la répétition d'une même séquence d'instructions :

- Avec des **données différentes** (a, b et c sont des données différentes pour le calcul de la factorielle) et à des endroits différents d'un programme.
- Sans avoir à réécrire cette séquence, il suffit juste de lui faire **appel**.
- Cette technique d'utilisation des sous programmes appelée **modularisation** facilite la programmation.

Il existe deux types de sous programmes : (1) les Procédures et (2) les Fonctions.

c. A retenir

- **Définir** le sous-programme.
- Chaque fois que l'exécution du sous-programme est voulue, il suffit de l'**appeler** par son nom accompagné des données sur lesquelles il va agir.

d. Fonctions

Une fonction est un s/s programme qui retourne un **résultat unique**.

En Langage algorithmique (LA), on adoptera la syntaxe illustrée par la Figure 7.2 pour définir une fonction.

iii. Définir une fonction

La Figure 1.2 montre les différentes parties nécessaires à la définition d'une fonction en LA.

<Entête de la fonction>	**fonction** <nom de la fonction> **(**liste de paramètres formels**) :**<type de la fonction> **;** ➢ La liste des paramètres formels peut être omise.
<Partie déclaration>	<Déclaration des constantes, des types, des variables, des fonctions et des procédures>. ➢ Selon la solution en LA, cette partie déclaration peut être omise.
<Corps de la fonction>	**Début** <instructions> ; **retourner** <valeur résultat > **;** **fin ;**

Figure 7.2 : Les différents parties nécessaires à la définition d'une fonction en LA.

iv. Faire Appel à une fonction

On fait appel à une fonction par le biais de son **nom suivi entre parenthèses** de la **liste des paramètres effectifs**.

<nom de la fonction> **(**liste de paramètres effectifs**) ;**

v. A retenir

- L'appel d'une fonction (<nom de la fonction> (liste de paramètres effectifs)) dans une instruction permet d'obtenir la valeur retournée par cette même fonction.
- Cette valeur retournée a pour type <type de la fonction>, à savoir, un type réel, entier, caractère ou logique… et obéit donc aux règles sémantiques et syntaxiques auxquelles devrait se conformer ce type.

Exemple

Les Figures 7.3 et 7.4 montrent deux exemples d'utilisation d'une fonction en LA et en Langage C dans le calcul de l'expression (a! + b!) / c!.

```
algorithme expression_factorielle;
var a,b,c: entier;
      exp: reel ;
fonction fact(n :entier) :entier ;
 var i,f :entier;
 debut
     f←1 ;
     pour i←2 jusqu'à n
        faire
          f←f*i ;
        ffpour ;
     retourner f ;
 fin;

debut
lire (a,b,c) ;
exp←(fact(a)+fact(b)) /fact(c);
fin.
```

(* Définition de la fonction qui calcule n! , pour n entier, et retourne dans la variable f le résultat du calcul *)

(* Trois Appels de la fonction par son nom fact accompagnée lors de chaque appel respectivement du paramètre a ou b ou c *)

Figure 7.3 : Utilisation d'une fonction dans le calcul de (a! + b!) / c! -Solution en LA-

```c
#include <stdio.h>
int fact(int n)
{
 int i, f=1;
 for (i=2; i<=n; i++)
    f=f*i;
 return f;
}
int main()
{
 int a, b,c,d;
 float s;
 printf("\nDonner a:");scanf("%d",&a);
 printf("\nDonner b:");scanf("%d",&b);
 printf("\nDonner c:");scanf("%d",&c);
 s=(float) (fact(a)+fact(b))/(fact(c);
 printf("S= %.2f",s);
 return 0;
}
```

Figure 7.4 : Utilisation d'une fonction dans le calcul de (a! + b!) / c! -Solution en Langage C-

e. Procédure

Une procédure est un s/s programme qui ne renvoi aucun résultat,

Une procédure permet de modifier des données, ou produire des effets physiques (lecture, écriture).

En LA, on adoptera la syntaxe illustrée par la Figure 7.5 pour définir une procédure.

i. Définir une procédure

La Figure 1.5 montre les différentes parties nécessaires à la définition d'une procédure en LA.

<Entête de la procédure>	**procedure** <nom de la procedure> **(**liste de paramètres formels**);** ➢ La liste des paramètres formels peut être omise
<Partie déclaration>	<Déclaration des constantes, des types, des variables, des fonctions et des procédures>. • Selon la solution en LA, cette partie déclaration peut être omise
<Corps de la procédure>	**debut** <instructions> ; **fin ;**

Figure 7.5 : Les différents parties nécessaires à la définition d'une procédure en LA.

ii. Faire Appel à la procédure

On fait appel à une procédure par le biais de son **nom suivi entre parenthèses** de la **liste des paramètres effectifs**.

<nom de la procédure> **(**liste de paramètres effectifs**) ;**

iii. A retenir

- L'appel d'une procédure ne retourner aucune valeur
- Une procédure contrairement à une fonction n'as pas de type.

f. Remarques

- L'exécution d'un algorithme commence par la **première instruction du corps de l'algorithme**.
- Après l'exécution d'une fonction ou d'une procédure, le **retour se réalise à l'instruction qui suit l'appel**.

3. Les variables locales et les variables globales

Avant d'appréhender les notions de variables locales et globales, nous définissons la notion de bloc en LA et en Langage C.

a. Structure de Bloc en LA

En LA, l'algorithme est organisé comme un ensemble de blocs imbriqués. La Figure 7.6 montre les différentes parties constituant une structure de bloc.

- <En tête > (algorithme ou fonction/procedure)
- <Déclarations> (Déclarations de constantes, de type, de variables, de procédures et/ou de fonctions)
- <Instructions exécutables>

Figure 7.6 : Les différents parties constituant une structure de bloc en LA.

Exemple

La Figure 7.7 montre un exemple de deux algorithmes en LA organisés en un ensemble de blocs imbriqués ainsi que leurs représentations arborescentes.

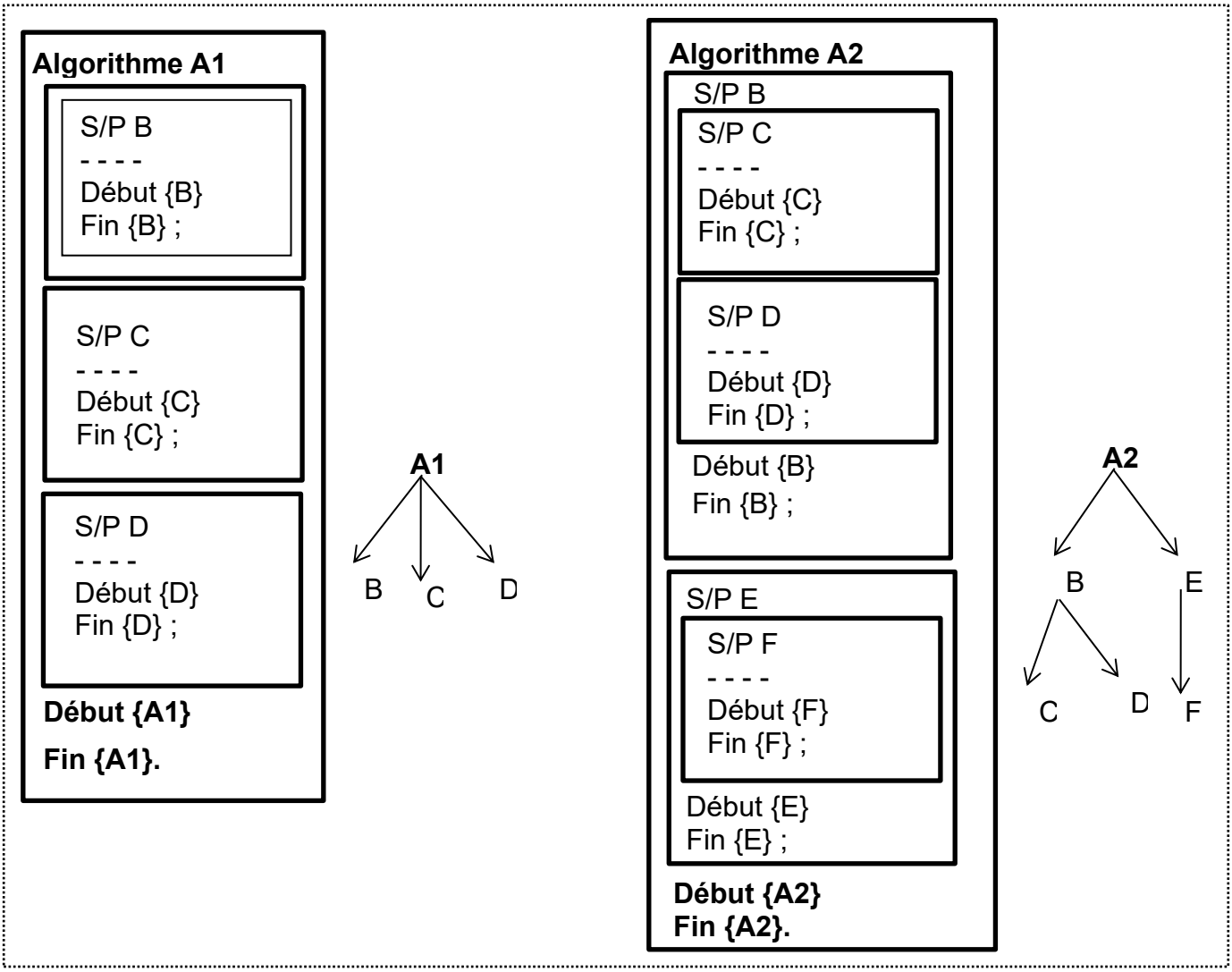


Figure 7.7 : Exemples d'algorithmes en LA organisés en blocs imbriqués.

b. Structure de Bloc en Langage C

En Langage C, les fonctions sont définies à l'aide de blocs d'instructions. Un bloc d'instructions est encadré d'accolades et est composé de deux parties comme présentées par la Figure 7.8.

```
{
    <déclarations locales>
    <instructions>
}
```

Figure 7.8 : Blocs d'instructions en C

En langage C, on peut avoir un bloc d'instructions d'une commande **if**, **while** ou **for** qui peut contenir des déclarations locales de variables et même de fonctions.

Exemple

Comme le montre la Figure 7.9, la variable I est déclarée à l'intérieur d'un bloc conditionnel. Si la condition (N>0) n'est pas remplie, I n'est pas défini. A la fin du bloc conditionnel, I disparaît.

```
if (N>0)
   {
    int I;
    for (I=0; I<N; I++)
       ...
   }
```

Figure 7.9 : Exemple d'un bloc conditionnel en C.

c. Variables locales et variables globales

La Figure 7.10 donne l'ensemble des règles à retenir sur la définition et l'utilisation des variables locales et des variables globales.

1. Toute variable, avant son utilisation dans un sous-programme, doit être déclarée dans ce sous-programme ou dans un bloc englobant de n'importe quel niveau.
2. Les variables déclarées dans un bloc d'instructions sont uniquement visibles à l'intérieur de ce bloc. On dit que ce sont des **variables locales** à ce bloc.
3. Une variable non déclarée dans un sous-programme peut y être utilisée si elle est déclarée dans un bloc englobant de n'importe quel niveau. Ces variables sont appelées des **variables globales** pour les blocs englobés.
4. Si une variable est déclarée dans les blocs de niveau différents, la déclaration locale est prioritaire par rapport à la déclaration globale.

Figure 7.10 : Règles à retenir sur la définition et l'utilisation des variables locales et des variables globales.

Exemples

1) Reprenons l'algorithme A2 de la figure 1.5 et illustrons l'application de ces règles à travers le Tableau 7.1.

	Les variables déclarées dans	Sont accessibles dans
Blocs	A2	A2, B, C, D, E, F
	B	B, C, D
	C	C
	D	D
	E	E, F
	F	F

Tableau 7.1 : Exemple 1 de la portée des variables dans une structure de blocs imbriqués.

2) Soit Figure 7.11 montrant une représentation arborescente constituée des blocs imbriqués A, B, C et D ainsi que les variables x, y et z déclarées au niveau de chaque bloc :

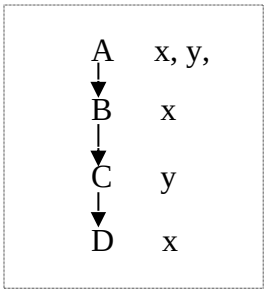


Figure 7.11 : Exemple d'une représentation arborescente d'une structure de blocs imbriqués.

Le Tableau 7.2 montre également l'application de ces règles.

Variable	Est accessible dans
x déclarée dans A	A
y déclarée dans A	A, B
z déclarée dans A	A, B, C, D
x déclarée dans B	B, C
y déclarée dans C	C, D
z déclarée dans D	D

Tableau 7.2 : Exemple 2 de la portée des variables dans une structure de blocs imbriqués.

4. Le passage des paramètres

Avant d'appréhender le passage des paramètres, nous définissons ce que sont les paramètres.

a. Les paramètres

Les paramètres fournissent un mécanisme de remplacement qui permet de répéter un sous-programme avec des données différentes.

Sachant que l'utilisation d'une procédure/fonction passe par la définition de la procédure/fonction et l'appel de la procédure/fonction. On distingue deux sortes de paramètres :

(1) Les **paramètres formels**[2]. Ce sont les paramètres de définition d'une procédure/fonction.
(2) Les **paramètres effectifs** ou **réels**[3]. Ce sont les paramètres d'appel de procédure/fonction.

Notons cependant les trois remarques suivantes concernant ces paramètres :

1- Les valeurs que représentent les paramètres formels ne sont déterminées qu'au moment de l'appel.

2- Il existe une correspondance positionnelle et une égalité en nombre entre les paramètres effectifs et les paramètres formels.

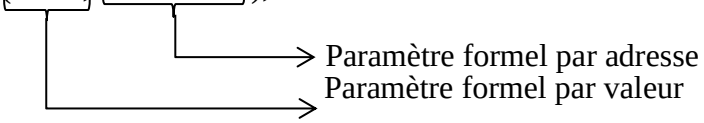
3- Le type de chaque paramètre doit être précisé dans la définition du sous-programme.

b. Les règles de passage de paramètres à l'appel d'un sous-programme

Les paramètres formels sont de deux (02) types :

(1) Les paramètres formels par **valeurs**.
(2) Les paramètres formels par **adresse** ou **référence** précédés par le mot clé **var** en LA et par le caractère ***** en langage C.

Exemple. procedure somme(a:reel; **var** b:entier);



c. Transfert de paramètres par valeur

Dans ce cas de transfert, le passage des paramètres se fait conformément aux quatre règles suivantes :

1. Au moment de l'appel du sous-programme, le paramètre effectif correspondant est calculé.
2. La valeur de ce paramètre est passée au sous-programme et devient la valeur initiale du paramètre formel correspondant qui joue le rôle d'une variable locale.
3. Le transfert par valeur ne peut passer l'information que de l'appelant vers l'appelé.
4. Le sous-programme peut modifier le paramètre formel qui est la copie du paramètre effectif mais il ne peut pas modifier la valeur du paramètre original.

1- Transfert par adresse (référence)

Dans ce cas de transfert le passage des paramètres se fait conformément aux trois règles suivantes :

1. Au moment de l'appel l'adresse du paramètre effectif correspondant est calculée.
2. Le paramètre formel correspondant sera associé à cette adresse.
3. L'appelant et l'appelé travaillent avec la même adresse.

[2] Appelés parfois uniquement : **paramètres**
[3] Appelés parfois : **arguments**

Exemple

Supposons que l'on souhaite calculer la note de travaux dirigés (TD) d'un étudiant selon la formule suivante :

Note TD = note présence + note assiduité +note contrôle 1 + note contrôle 2

Cependant, on doit faire un contrôle de saisie des quatre notes, à savoir :

0<=note présence<=2, 0<=note assiduité<=4, 0<=note contrôle 1<=7, 0<=note contrôle 2<=7

Le Tableau 7.3 contient deux solutions pour cet exemple, une solution triviale et une solution utilisant un sous-programme avec des paramètres par adresse.

Solution triviale

```
algorithme note_td ;
 var    nt,np,na,nc1,nc2: reel;
debut
 repeter lire (np) ;
 jusqu'à (np>=0 et np<=2);

 repeter lire (na) ;
 jusqu'à (na>=0 et na<=4);

 repeter lire (nc1) ;
 jusqu'à    (nc1>=0    et nc1<=7);

 repeter lire (nc2) ;
 jusqu'à    (nc2>=0    et nc2<=7);

 nt←np+na+nc1+nc2;
fin.
```

Solution utilisant une procédure avec des paramètres par adresse

Les suites d'instructions encadrées sont pratiquement les mêmes. Pour éviter cette répétition on utilise une procédure.

```
algorithme note_td ;
 var nt,np,na,nc1,nc2: reel;
 Procedure controle(var n :reel;lim:reel);
   Debut
        repeter lire (n) ;
        jusqu'à (np>=0 et np<=lim);
   fin;
debut
  controle(np,2);
  controle(na,4);
  controle(nc1,7);
  controle(nc2,7);
  nt←np+na+nc1+nc2;
fin.
```

Tableau 7.3 : Exemple d'utilisation d'une procédure avec des paramètres par adresse.

Remarques concernant cet exemple

- Dans cet exemple, nous avons utilisé une procédure au lieu d'une fonction car le contrôle d'une saisie d'une variable ne renvoie pas de résultat.

- Dans la procédure **`controle`**, le paramètre formel **n** est un paramètre formel par adresse (référence) car l'instruction **lire(n)** doit pouvoir accéder aux variables globales np, na, nc1et nc2 déclarés dans l'algorithme **note_td**.

  ➔ Ainsi, le paramètre formel **n**, après appel, contient l'adresse de la variable globale (np, na, nc1 ou nc2) et non pas sa valeur.

5. La récursivité

Dans cette section, nous commençons par donner une définition d'un sous-programme récursif. Nous illustrons par la suite cette notion en donnant deux exemples. Finalement nous citerons les critères d'un sous-programme récursif correct.

a. Définition

Un sous-programme est dit récursif lorsqu'il fait appel à lui-même.

- La récursivité permet de résoudre certains problèmes de manière très rapide comparativement à leur résolution itérative.
- Une solution itérative consommerait plus de temps et de structures de données intermédiaires qu'une solution récursive.

b. Exemples

i. Calculer la factorielle d'un entier naturel n selon la formule : n! = n*(n-1)*(n-2)*…2*1

- Algorithme utilisant une fonction itérative comme le montre la Figure 7.11.

```
Fonction fact_iterative(n:entier):entier;
  var i, f:entier;
   Debut
     f←1;
     Pour i← 2 jusqu'à n
        Faire f←f*i;
        FFpour;
      retourner f;
    Fin;
```

Figure 7.11 : Algorithme de calcul de n ! utilisant une fonction itérative

- Algorithme utilisant une fonction récursive comme le montre la Figure 7.12.

Pour ce faire, on constate que (si n=0 alors n!=1) et que (si n>0 alors n!=n * (n-1)!)

```
Fonction fact_recursive(n:entier):entier;
var f :entier ;
  Debut
     Si n<=1
            alors (*cas particulier ou cas d'arrêt*)
                    f←1 ;
             sinon (*cas général*)
                     f→ n*fact(n-1);
     Fsi;
     retourner f;
  Fin;
```

Figure 7.12 : Algorithme de calcul de n ! utilisant une fonction récursive

Faisons appel à la fonction récursive `fact_recursive` en lui passant le paramètre effective n=4 (`fact_recursive` (4)). Ce qui donne le déroulement suivant, tel qu'illustré par la Figure 7.13.

Exécution de l'appel **`fact(4)`**

n=4, on exécute la partie `sinon` de l'instruction `si` : `f←4 *` Appel de **`fact(3)`**

Exécution de l'appel **`fact(3)`**

n=3, on exécute la partie `sinon` de l'instruction `si` : `f←4 * 3 *` Appel de **`fact(2)`**

Exécution de l'appel **`fact(2)`**

n=2, on exécute la partie `sinon` de l'instruction `si` : `f←4 * 3 * 2 *` Appel de **`fact(1)`**

Exécution de l'appel **`fact(1)`**

n=1, on exécute la partie `alors` de l'instruction `si` : `f←4 * 3 * 2 * 1 ;`

On sort de l'instruction `si` et on exécute `retourner` **`24`**

Figure 7.13 : Déroulement de l'appel de `fact_recursive` (4).

ii. Calculer le $n^{ième}$ terme de la suite de Fibonacci définie comme suit :

$$\begin{cases} F_1=F_2=1 \text{ (ce sont les } 1^{er} \text{ et } 2^{ème} \text{ termes de la suite de Fibonacci)} \\ F_{n+1}=F_n+F_{n-1} \end{cases}$$

- Algorithme utilisant une fonction itérative comme le montre la Figure 7.14.

```
fonction fibo_iterative(n:entier):entier;
 var fib,un1,un2,i:entier;
 Debut
   Si (n=0) ou (n=1)
       Alors fib←1;
       Sinon un1←1;
             un2←1;
             Pour i←2 jusqu'a n
                faire fib←un1+un2;
                      un1←un2;
                      un2←fib ;
                ffpour ;
   FSi ;
    retourner fib ;
 fin;
```

Figure 7.14 : Algorithme de calcul du $n^{ième}$ terme de la suite de Fibonacci -cas itératif-

- Algorithme utilisant une fonction récursive comme le montre la Figure 7.15.

```
fonction fibo_recursive(n:entier):entier;
 var fib,un1,un2,i:entier;
 Debut
  Si n<=2 Alors fib←1; (*cas particulier ou cas d'arrêt*)
          Sinon fib← fibo(n-1)+ fibo(n-2); (*cas général*)
  FSi ;
 retourner fib;
Fin;
```

Figure 7.15 : Algorithme de calcul du $n^{ième}$ terme de la suite de Fibonacci -cas récursif-

c. Critères d'un sous-programme récursif correct.

Un sous-programme récursif correct doit remplir essentiellement les critères suivants :

Critère 2. Un sous-programme récursif doit nécessairement se terminer, il doit se dérouler en un nombre d'étapes fini.

Critère 1. Dans un sous-programme récursif, il doit exister un ou plusieurs cas d'arrêt.

Critère 2. Les appels récursifs utilisent toujours des paramètres (arguments) inférieurs à ceux donnés en entrée.

Critère 3. Il faut s'assurer qu'après simplifications successives on arrive au cas particulier.

6. Série d'exercices – Les sous-programmes : les fonctions et les procédures-

La solution de cette série d'exercice est donnée dans la partie Annexe de ce présent polycopié de cours.

Exercice 01

En utilisant des procédures/fonctions, écrire un algorithme qui permet de calculer $e^y + e^z$, sachant que :

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + x^4/4! + ... + x^n/n! \quad , \forall x \quad \text{(Euler)}$$

Exercice 02

En utilisant des procédures/fonctions, écrire un algorithme qui permute le nombre réel maximum avec le nombre réel minimum d'un vecteur contenant n (n>0) nombres réels.

Exercice 03

Soit A un tableau d'entiers. En utilisant des procédures/fonctions récursifs écrire l'algorithme qui permet de :

a) Trouver le maximum de A.
b) Trouver le minimum de A.
c) Calculer la somme des éléments de A.
d) Calculer le produit des éléments de A.
e) Calculer la moyenne des éléments de A.

Chapitre 8 Les fichiers

Ce chapitre est divisé en cinq sections. La première section est une introduction à la notion de fichiers. La deuxième section donne les définitions se rapportant à cette notion et précise le cadre d'étude. La troisième section aborde les types de fichiers, à savoir les modes de structuration des données dans un fichier. La section quatre est consacrée à la manipulation des fichiers en LA et en Langage C. Finalement, la section cinq propose une série d'exercices se rapportant à ce chapitre.

1. Introduction

Nous introduisons la notion de **Fichier**, à partir des deux exemples suivants :

Exemple 1. On désire obtenir une liste contenant les noms, prénoms et notes de 100 étudiants ordonnée selon leurs notes. Pour ce faire, on écrit un programme permettant de réaliser ce traitement.

Exemple 2. On veut écrire un programme qui nous permet de saisir du texte contenant des prises de notes sur un cours. On veut par la suite consulter ces prises de notes ou bien les mettre à jour. Pour ce faire on écrit un programme permettant de réaliser ce traitement.

a. Cas de l'exemple 1

Une fois que le programme est écrit, on l'**exécute.** Cette exécution se traduit par :

- La saisie par l'utilisateur à partir du clavier de 100 noms, 100 prénoms et 100 notes.
- Le programme les classe automatiquement par rapport aux notes saisies et affiche la liste des 100 étudiants ordonnée selon leurs notes.

⇨ MAIS une fois que l'**exécution du programme est terminée toutes ces données disparaissent**.

- Si on veut **lister encore une fois le résultat** du programme, il va falloir **réexécuter le programme**.
- Si on veut **rectifier une saisie erronée,** il va falloir **réexécuter le programme.**

b. Cas de l'exemple 2

Une que fois le programme est écrit, on l'exécute. Cette exécution se traduit par :

- La saisie par l'utilisateur à partir du clavier du texte à rédiger.

⇨ MAIS une fois que l'**exécution du programme est terminée le texte saisi est perdu.**

c. Constatation

Des deux exemples précédents nous pouvons dire que toutes les données utilisées dans nos algorithmes/programmes :

1. Sont créées par nos algorithmes/programmes
2. Existent pendant l'exécution de l'algorithme/programme et (3) une fois que l'exécution de l'algorithme/programme est terminée ces données disparaissent.

Deux questions se posent :

1. Comment faire exister ou mémoriser des données hors de l'algorithme/programme qui les crée, c'est-à-dire d'une manière permanente ?
2. Comment manipuler ces données indépendamment de l'exécution de l'algorithme/programme ?

La réponse à ces deux questions se trouve dans la section suivante.

2. Définitions

Il existe en algorithme/programme une structure de données appelée **fichiers** qui répond aux deux questions précédentes. Ces fichiers nécessitent un moyen de stockage permanent (disque dur par exemple).

On retient la définition suivante pour les fichiers : Un fichier est un ensemble d'informations stockées sur un support physique (disque ou autre mémoire externe). C'est donc un ensemble de données que l'on peut conserver de façon permanente, même quand la machine est éteinte.

a. A retenir

- L'ensemble des informations (données) qui se trouvent en mémoire externe (disque dur, clé USB...) sont organisées sous forme d'un ou plusieurs fichiers.

- Tout fichier enregistré sur un support externe est repéré par son **nom**, appelé **nom externe** ou **nom physique**

<nom du fichier>.<suffixe/extension>

nom externe ou nom physique du fichier

Exemple : 'liste_etud_S3.xls' 'cours_asdd2.txt' 'td1_exo1.c' 'compte_rendu.txt'

- Le suffixe ou l'extension du nom du fichier indique son type ou son format en d'autres termes comment que sont organisées les informations contenues dans ce fichier.

- Ainsi pour l'exemple précédent, l'extension 'xls' du nom de fichier 'liste_etud_S3.xls' nous indique que c'est un fichier Excel. De même le suffixe 'txt' du nom de fichier 'cours_asdd2.txt' nous indique que c'est un fichier texte.

b. Cadre d'étude

Nous tenons à préciser que dans ce chapitre de cours, notre cadre d'étude sur les fichiers est limité uniquement sur les fichiers en langage algorithmique et en langage C en général.

3. Types de fichiers

Comme nous l'avons précisé dans les sections précédentes, les fichiers sont une structure de données qui va nous permettre de stocker ou de manipuler des données sur un support permanent tel que le disque dur.

⇨ Notons que ces données sont mémorisées sous la forme d'une **suite d'octets**.

De ce fait, on peut choisir entre deux modes de structuration des données dans un fichier :

1. **binaire** ou
2. **texte**.

➔ Lorsque l'accès à chaque élément du fichier se fait en séquence, c'est à dire dans l'ordre d'apparition des éléments dans le fichier, on dit que le fichier est à **accès séquentiel**. Pour lire l'élément i, on doit lire tous les éléments numérotés 1 à n – 1.

➔ Lorsque chaque élément du fichier peut être lu directement, quelle que soit sa place dans le fichier, on dit que le fichier est à **accès direct**.

Le Tableau 8.1 ci-dessous résume l'organisation et les caractéristiques essentielles des fichiers en mode binaire et en mode texte.

Fichier en mode binaire	**Fichier en mode texte**
➢ Chaque donnée est stockée selon les règles de codage imposées par son type (entier, reel, caractere…) ➢ La taille du fichier est alors optimale et les données sont facilement lues/écrites en peu d'instructions. ➢ On ne peut pas éditer le fichier avec un éditeur de texte tel que Notepad, dans le but de modifier ou de vérifier son contenu. ➢ Les données ne peuvent être lues/écrites que par le biais d'un programme. Ce qui est peut-être considéré comme une contrainte ➢ Le fichier peut être à accès direct ou à accès séquentiel. ➢ Ce type de fichier est utilisé pour : o Faire des sauvegardes de données entre deux appels d'un programme. ➢ Il se prête le mieux pour : o La gestion des bases de données.	➢ Chaque donnée est stockée sous la forme d'une succession de codes ASCII (ensemble de caractères) ➢ Les données sont organisées en ligne. Chaque ligne est terminée par une marque de fin de ligne **eol** (end of line) ➢ Les données occupent plus de place et sont difficiles à manipuler par un programme. ➢ Les données du fichier peuvent être manipulées par l'utilisateur à l'aide d'un éditeur de texte tel que Notepad. C'est son plus grand avantage. ➢ Les données du fichier peuvent également être manipulées par un programme, dans ce cas l'accès aux données du fichier est séquentiel.
➢ Un fichier est toujours terminé par une marque de fin de fichier **eof** (**e**nd **o**f **f**ile). Qu'il soit en mode binaire ou en mode texte,	

Tableau 8.1 : Organisation et caractéristiques essentielles des fichiers en mode binaire et en mode texte.

4. Manipulation des fichiers

Pour manipuler les fichiers on introduit la notion de primitives de gestion des fichiers.

a. Primitives de gestion des fichiers

Pour manipuler un fichier, quel que soit son mode, binaire ou texte, nous faisons des appels à des **primitives de gestion** qui sont des **sous-programmes** ayant pour paramètre la variable fichier ou le nom logique du fichier[4].

Ces **primitives de gestion** concernent :

1. La **gestion de fichiers** (Ouverture et Fermeture du fichier)
2. L'**utilisation des fichiers** (Lecture et Ecriture dans le fichier)

Elles suivent généralement le même processus illustré par la Figure 2.1 :

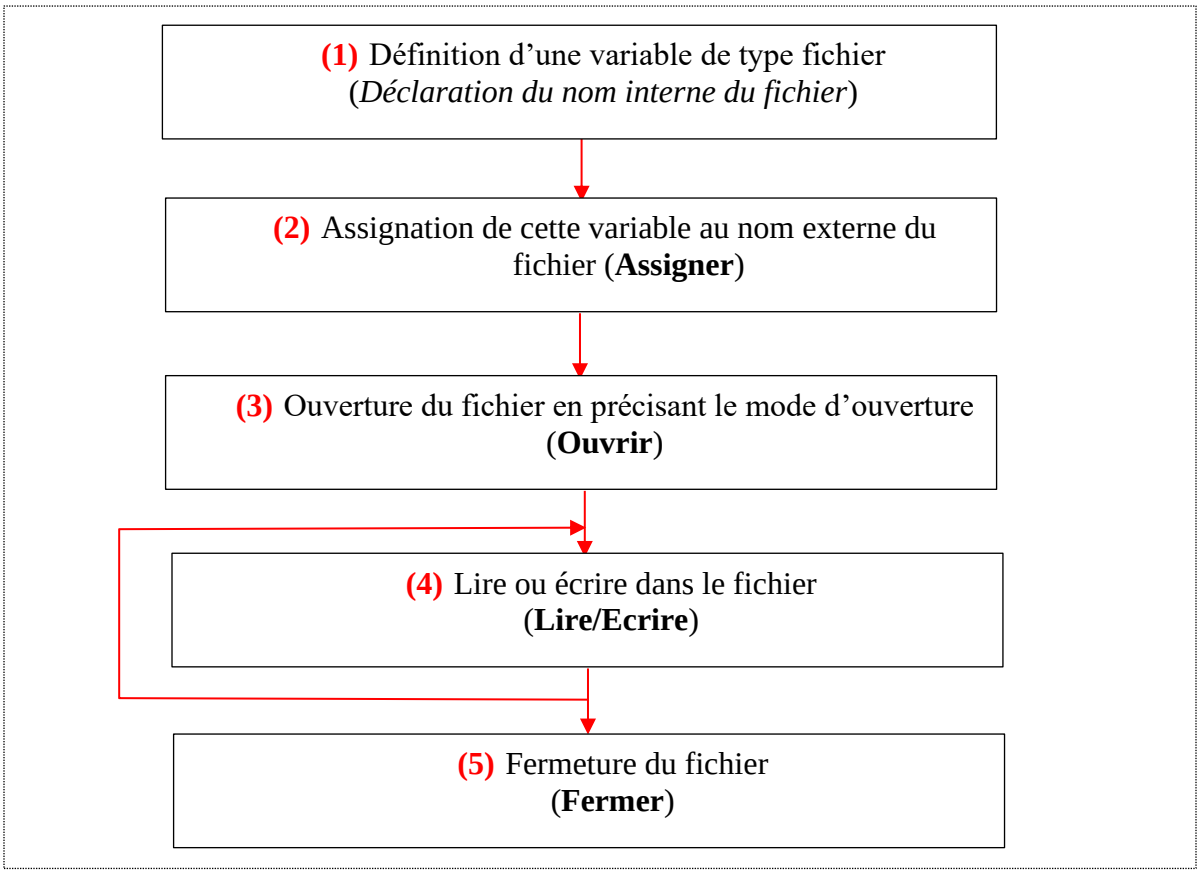


Figure 8.1 : Processus suivi dans la manipulation des fichiers.

Nous détaillons dans ce qui suit ces primitives de gestion de fichier.

b. Déclaration d'un fichier

Dans ce qui suit, nous donnerons quelques primitives de gestion de fichier en mode texte et en mode binaire en LA et en Langage C.

[4] Appelé dans la suite de ce chapitre : `<nom_interne_fichier>`

i. En Langage Algorithmique

Un fichier **binaire** est déclaré comme suit :

```
type <nom_fichier> =  Fichier  de  <type de base> ;
var  <nom_interne_fichier> : <nom_fichier> ;
```

Un fichier **texte** est déclaré comme suit :

```
type <nom_fichier> =  text ;
var  <nom_interne_fichier> : <nom_fichier> ;
```

Où `<nom_interne_fichier>` :

3. Désigne un identificateur
4. Est associé au fichier de **nom externe** se trouvant sur le support physique
5. Permet de réaliser toutes les opérations sur ce fichier (on le verra par la suite)

Exemples

Déclaration d'un fichier binaire

```
type  employe =  enregistrement
                  Matricule : chaine de caracteres[5] ;
                  Nom, prenom : chaine de caracteres[20] ;
                  Adresse : chaine de caracteres[30] ;
                  Salaire : reel ;
                 fin enregistrement ;
     TfichierB = fichier de employe ;
var  fichier : Tfichier ;
```

Déclaration d'un fichier texte

```
type  TfichierT =  text ;
Var  fichier : TfichierT ;
```

ii. En Langage C

```
FILE* <nom_interne_fichier> ;
```

Dans le cas du langage C, remarquons la présence de `*`, qui indique que `<nom_interne_fichier>` est une variable pointeur, et le mot clé `FILE` qui indique que cette variable pointeur est associée (liée) au fichier de **nom externe** se trouvant sur le support physique.

c. A retenir

- Un lien doit toujours être établi entre le fichier logique repéré par `<nom_interne_fichier>` et le fichier réel se trouvant sur le support physique repéré par le **nom externe** du fichier

d. Opérations sur les fichiers

Il existe plusieurs fonctions et procédures permettant de faire des opérations sur les fichiers, nous commençons par voir ces opérations en LA (en mode texte et en mode binaire), ensuite en langage C (en mode texte et en mode binaire)

i. En Langage Algorithmique

Le Tableau 8.2 liste les opérations qu'on peut faire sur un fichier en mode texte, ainsi que les significations qui leur correspondent.

Opération	Signification
assigner(<nom_interne_fichier>,<chemin du fichier sur le support>);	Etablit le lien entre la variable <nom_interne_fichier> et le fichier qui se trouve sur le support physique.
ouvrir(<nom_interne_fichier>, <mode d'ouverture>**);**	Ouvre un fichier en mode texte. Selon le <mode d'ouverture> = '**C**' : Ecriture seule, si le fichier n'existe pas alors il est créé sinon il est vidé de son contenu. '**L**' : Lecture seule, si le fichier n'existe pas alors une erreur se produit, sinon la tête de lecture est positionnée sur le premier caractère du fichier. '**A**' : Ajout fin de fichier, la tête d'écriture est positionnée à la fin du fichier.
fin_ligne(<nom_interne_fichier>**);**	Fonction qui retourne un booléen=vrai si la tête de lecture est sur une marque de fin de ligne (ou sur la marque de fin de fichier), faux sinon.
fin_fichier(<nom_interne_fichier>**)** ;	Fonction booléenne qui retourne la valeur vraie si la fin du fichier a été atteinte, fausse sinon.
fermer(<nom_interne_fichier>**)** ;	Ferme le fichier quel que soit le mode d'ouverture utilisé.
ecrire(<nom_interne_fichier>, <expression>**)** ;	Ecrit un texte dans le fichier. <expression> est une expression de type caractère, logique, entier, réel ou chaîne de caractères. Le fichier doit être préalablement ouvert avec <mode d'ouverture>='**C**' ou '**L**'.
ecrire_nl(<nom_interne_fichier>, <expression>**)** ;	Ecrit un texte dans le fichier et rajoute une marque de fin de ligne au fichier.
lire_nl(<nom_interne_fichier>, <variable_chaine_caractere>**)** ;	Equivalent à **Lire(**<nom_interne_fichier>, <variable>**)** où <variable> est de ' **type chaine de caracteres**'. Mais la marque de fin de la ligne courante est ignorée et la tête de lecture est déplacée jusqu'au caractère suivant la prochaine fin de ligne.

lire(<nom_interne_fichier>, <variable>**) ;**	Le fichier doit être ouvert en lecture, la tête de lecture ne doit pas être sur la marque de fin de fichier, et selon <variable> est : 'type **caractere**' : Lecture du caractère situé sous la tête de lecture, et stockage dans <variable>. La tête de lecture est avancée d'un caractère. 'type **entier** ou **reel**' : Lecture de l'entier ou réel situé dans le fichier à partir de la position donnée par la tête de lecture, et stockage dans <variable>. Les Espaces, tabulations, fin de lignes sont ignorés avant la lecture de l'entier ou réel. S'il n'y a pas d'entier ou réel trouvé, alors erreur à l'exécution. Après l'exécution de la primitive, la tête de lecture est positionnée juste après l'entier ou le réel. 'type **chaine de caracteres**' : Lecture de la chaine de caractères dans le fichier à partir de la position de la tête de lecture, jusqu'à la prochaine marque de fin de ligne. Les caractères lus sont stockés dans <variable>. La tête de lecture est ensuite positionnée sur la marque de fin de ligne.

Tableau 8.2 : Opérations sur un fichier en mode texte (LA).

Le Tableau 8.3 liste les opérations qu'on peut faire sur un fichier en mode binaire, ainsi que les significations qui leur correspondent

Opération	Signification
assigner(<nom_interne_fichier>,<chemin du fichier sur le support>);	Idem que le fichier en mode texte.
creer(<nom_interne_fichier>**);**	Ouvre un fichier en mode binaire, en Ecriture et en Lecture, si le fichier n'existe pas alors il est créé sinon il est vidé de son contenu.
ouvrir(<nom_interne_fichier>**);**	Ouvre un fichier en mode binaire, en Lecture et en Ecriture, si le fichier n'existe pas alors une erreur se produit, sinon la tête de lecture est positionnée sur le premier caractère du fichier.
fin_fichier(<nom_interne_fichier>**) ;**	Idem que le fichier en mode texte.
fermer(<nom_interne_fichier>**) ;**	Ferme le fichier quel que soit le mode d'ouverture utilisé.
ecrire(<nom_interne_fichier>, <expression>**) ;**	Ecrit un texte dans le fichier. <expression> est une expression de type caractère, logique, entier, réel ou chaîne de caractères. Le fichier doit être préalablement ouvert avec <mode d'ouverture>='**C**' ou '**L**'.

Opération	Signification
fermer(<nom_interne_fichier>**) ;**	Idem que le fichier en mode texte.
ecrire(<nom_interne_fichier>, <expression>**) ;**	Ecrit la valeur de <expression> dans le fichier. Si (la tête de lecture/écriture est positionnée sur la marque de fin de fichier) Alors La valeur de <expression> est écrite à la place de la marque de fin. La marque de fin est placée après ce nouvel élément. La tête de lecture est positionnée sur la marque de fin. Sinon La valeur de <expression> est écrite à la place de la valeur située sous la tête de lecture/écriture. La tête de lecture est avancée sur l'élément suivant. FSi ;
lire(<nom_interne_fichier>, <expression>**) ;**	Une copie de l'élément situé sous la tête de lecture est stockée dans <expression>. La tête de lecture est avancée jusqu'à l'élément suivant. Pour lire dans un fichier, la tête de lecture ne doit pas être sur la marque de fin de fichier.
taille_fichier(<nom_interne_fichier>**);**	Fonction qui retourne le nombre d'éléments du fichier.
pos_fichier(<nom_interne_fichier>**);**	Fonction qui retourne la position de l'élément du fichier qui se trouve sous la tête de lecture écriture. Les éléments du fichier sont numérotés de **0** jusqu'à **taille_fichier(**<nom_interne_fichier>**)-1.**
chercher_pos(<nom_interne_fichier>, <position>**) ;**	<position> est un entier compris entre **0** et le nombre d'éléments du fichier. Si 0<=pos<**taille_fichier(**<nom_interne_fichier>**)** Alors La tête de lecture/écriture est positionnée sur l'élément numéro <position>. Si <position>=**taille_fichier(**<nom_interne_fichier>**)** Alors La tête de lecture/écriture est positionnée sur la marque de fin de fichier. Si non1) et non 2) Alors Erreur à l'exécution.

Tableau 8.3 : Opérations sur un fichier en mode binaire (LA).

ii. En Langage C

Le Tableau 8.4 liste les opérations qu'on peut faire sur un fichier en mode texte, ainsi que les significations qui leur correspondent.

Opération	Signification
fopen(<Fichier physique>, <mode_ouverture>**)** ;	Selon <mode_ouverture> = "**r**" (Lecture seule, le fichier doit exister) "**w**" (Ecriture seule) "**a**" (Ajout fin de fichier) "**r+**" (Lecture/Ecriture, le fichier doit exister) "**w+**" (Lecture/Ecriture, supprime le contenu) "**a+**" (Ajout Lecture/Ecriture, fin de fichier)
fclose(<fichier>**)** ;	Fermer un fichier ouvert
feof(<fichier>**)** ;	Tester la fin d'un fichier
fgetc(<fichier>**)** ;	Lire un caractère
fgets(<chaine>,<taille_chaine>,<fichier>**)**	Lit une ligne
fscanf(<fichier>,<format>,...**)** ;	Lit du texte formaté (%d, %c, ...)
fputc(<caractere>, <fichier>**)** ;	Ecrit un caractère
fputs(<chaine>, <fichier>**)** ;	Ecrit une ligne de texte
fprinf(<fichier>, <format>,...**)** ;	Ecrit du texte formaté
ftell(<fichier>**)** ;	Retourne la position courante du curseur
fseek(<fichier>,<deplacement>,<origine>**)**	Déplace le curseur de <deplacement> à partir de <origine> Selon <origine> = SEEK_SET (Début de fichier) SEEK_CUR (Position courante) SEEK_END (Fin de fichier)
rewind(<fichier>**)**	Réinitialise la position du curseur.
rename(<ancien_nom_physique>,< nouveau_nom_physique>**)** ;	Le fichier doit être fermé.
remove(<fichier_physique >**)**	Le fichier doit être fermé.

Tableau 8.4 : Opérations sur un fichier en mode text (Langage C).

En ce qui concerne les fichiers en mode binaire en langage C, il faut rajouter le caractère '**b**' dans le <mode_ouverture> du fichier pour dire que c'est un fichier binaire.

[ECRITURE]

```
fwrite(<adr_donnee>, <taille_donne>,<nombre_donnee>,<fichier>) ;
```

[LECTURE]

```
fread(<adr_donnee>, <taille_donne>,<nombre_donnee>,<fichier>)
```

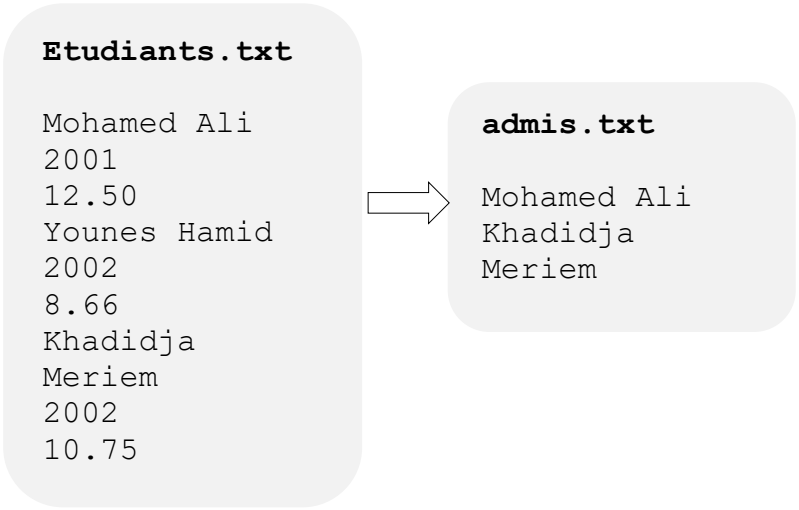
Exemple.

Les deux exemples suivants concernent la manipulation de fichiers. Dans le premier exemple, il s'agit d'écrire un algorithme qui manipule des fichiers de type 'text'. Dans le second exemple, il s'agit d'écrire d'un algorithme qui sauvegarde des informations dans un fichier binaire (structuré).

Exemple 1. On désire sauvegarder dans un fichier texte ayant pour nom externe **'admis.txt'** les noms des étudiants admis, à partir d'un autre fichier texte qui existe déjà et qui a pour nom externe **'etudiants.txt'.**

Le fichier **'etudiants.txt'** contient, pour chaque étudiant sur des lignes séparées :

- Le nom
- L'année de naissance
- La moyenne



Exemple 2. On veut sauvegarder et afficher la liste des étudiants, où chaque étudiant est caractérisé par (1) son numéro d'inscription, (2) son nom, (3) son année de naissance et (4) sa moyenne. Notre algorithme doit nous permettre de :

1) Saisir et sauvegarder un étudiant dans un fichier ayant pour nom **'etudiants.dat'**
2) Afficher un étudiant.

On écrira cet algorithme en utilisant des procédures de saisie, d'ajout et d'affichage. La Figure 8.2 donne en LA la solution à l'exemple1.

```
algorithme resultat_admis;
 type tFichT = text;
 var finput, foutput: tFichT; (*Déclaration des variables fichiers logiques de type text *)
     nom: chaine de caractere[20];
     annais: entier;
     moy: reel;

 debut

 (*Etablir le lien entre la variable fichier logique finput et le fichier de nom externe 'etudiants.txt'*)
   assign(finput, 'etudiants.txt');
 (*Ouverture de 'etudiants.txt' en lecture seule*)
   ouvrir(finput, 'L');

 (*Etablir le lien entre la variable fichier logique foutput et le fichier de nom externe 'admis.txt'*)
   assign(foutput, 'admis.txt');
 (*Ouverture du fichier de nom 'admis.txt' en mode création, s'il existe déjà, il sera vidé de son contenu *)
   ouvrir(foutput, 'C');

 (*Boucle de remplissage du fichier de nom 'admis.txt' à partir du fichier de nom 'etudiants.txt' *)
  tant que non fin_fichier(finput)
         faire lire_nl(finput,nom);
               lire_nl(finput,annais);
               lire_nl(finput,moy);
               si moy>= 10.00
                  alors ecrire_nl(foutput,nom);
               fsi ;
         fftque;

(*Fermeture des fichiers de nom 'etudiants.txt' et 'admis.txt' *)
 fermer(finput);
 fermer(foutput);

fin.
```

Figure 8.2 : Exemple d'un algorithme qui manipule un fichier de type text.

La Figure 8.3 donne en LA la solution à l'exemple 2.

```
algorithme gerer_etudiants;
 type tetud = enregistrement
                ni : entier ;
                nom: chaine de caractere[20];
                annais: entier;
               moy: reel;
             fin enregistrement ;
      tFichierB = fichier de tetud ;

 var fich_etud: tFichierB; etud :tetud ;

 procedure saisie(var e : tetud) ;
   debut
    ecrire('Numero inscription: ');  lire(e.ni);
    ecrire('Nom : '); lire (e.nom);
    ecrire('Annee de naissance : '); lire(e.annais);
    ecrire('Moyenne : '); lire(e.moy);
   fin;

 (* Ajouter l'étudiant e dans le fichier f *)
 procedure ajouter(var f: tFichierB; e: tetud);
  debut
   chercher_pos(f, taille_fichier(f));
   ecrire(f, e);
  fin;

 (*Afficher le contenu du fichier f *)
 procedure afficher(var f: tFichierB);
  var e: tetud;
  debut
   chercher_pos(f, 0);
   tant que non fin_fichier(f)
           faire lire(f, e);
                 ecrire(e.ni, ' ', e.nom, ' ', e.annais, ' ',e.moy);
           fftque;
  fin;


 debut (*Algorithme*)
 (*Etablir le lien entre la variable fichier logique fich_etud  et le fichier de nom externe 'etudiants.dat' *)
   assign(fich_etud, 'etudiants.dat');
 (*Ouverture de 'etudiants.dat', si le fichier n'existe pas on l'ouvre avec creer(fich_etud)*)
```

```
  ouvrir(fich_etud);
 (*Ajout d'un étudiant, Utiliser une boucle pour ajouter plusieurs étudiants dans le fichier *)
  saisie(etud);
  ajouter(fich_etud, etud);
 (*L'affichage est fait pour tous les étudiants se trouvant dans le fichier *)
  afficher(fich_etud);
 (*Fermeture du fichier de nom 'etudiants.dat' *)
  fermer(fich_etud);
fin (*Algorithme*).
```

Figure 8.3 : Exemple d'un algorithme qui manipule un fichier binaire structuré.

Chapitre 9 Les listes chainées

Ce chapitre est divisé en huit sections. La première section est une introduction aux variables statiques et aux variables permettant une gestion dynamique de la mémoire centrale. La deuxième section aborde la notion d'un nouveau type de variable, à savoir, le type pointeur. La troisième section est consacrée à la gestion dynamique de la mémoire grâce aux variables pointées. La quatrième section aborde une nouvelle structure de données, à savoir, les listes linéaires chainées (llc). La cinquième section se focalise sur les opérations essentielles de manipulation des llc. La sixième section aborde les listes linéaires doublement chainées. La section sept traite des listes chainées particulières, à savoir, les piles et les files. Finalement, la section huit propose une série d'exercices se rapportant à ce chapitre et dont la solution est donnée ….

1. Introduction

Rappelons tout d'abord que la mise en forme d'un algorithme par le LA se présente selon la Figure 9.1.

Algorithme <identificateur>**;**	Partie **Entête** de l'algorithme
<Déclaration de constantes> <Déclaration de types> <Déclaration de variables> <Déclaration de procédures et de fonctions>	Partie **Déclaration** de l'algorithme
Début <Instructions exécutables> **Fin**.	Partie **Corps** de l'algorithme

Figure 9.1: La mise en forme d'un algorithme par le LA.

Les valeurs entières, caractères, logiques, réelles, … manipulées dans ce LA sont stockées dans des cases mémoires appelées variables.

Ces variables portent un nom (identificateur de variable) et sont déclarées dans la partie **Déclaration** de l'algorithme.

Rappelons ensuite que la réalisation d'un algorithme passe par deux étapes, à savoir :

Etape 1. La conception et l'écriture de l'algorithme (design time).

Etape 2. L'exécution de l'algorithme (runtime).

Rappelons finalement que le cycle de vie de l'exécution d'un algorithme se présente selon la Figure 9.2.

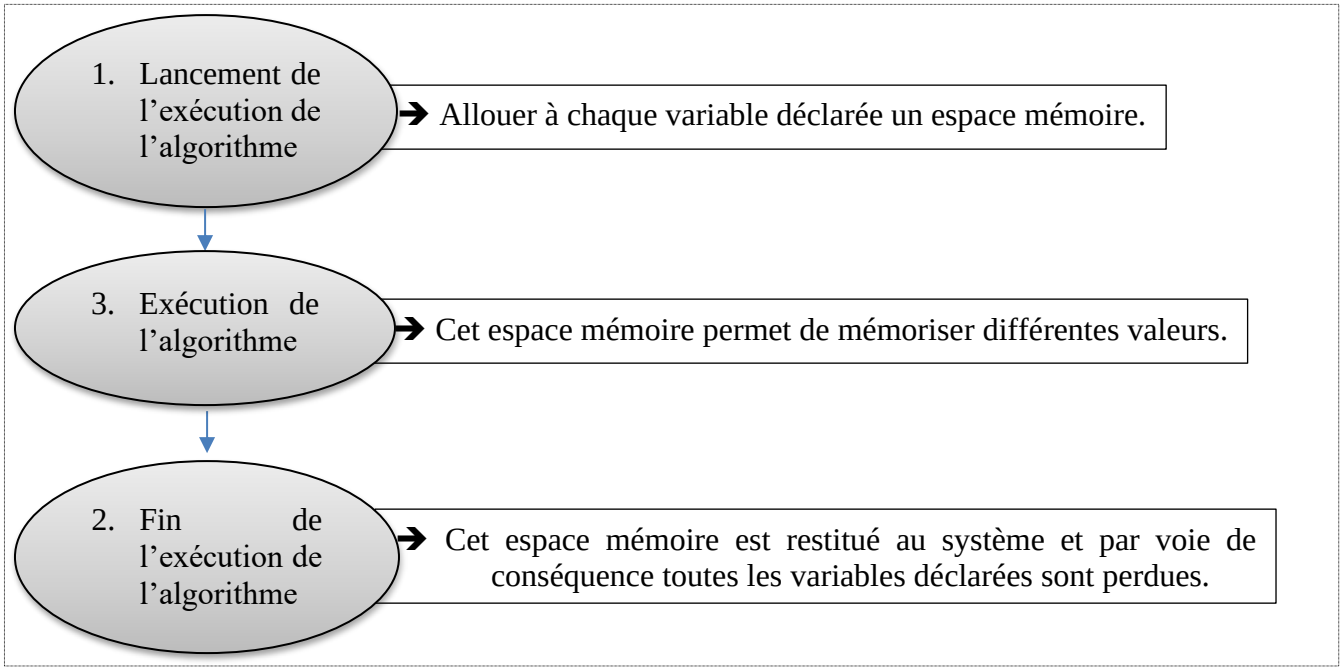


Figure 9.2: Cycle de vie de l'exécution d'un algorithme.

De ce cycle de vie, nous pouvons tirer les trois conséquences suivantes :

Conséquence 1. L'allocation de l'espace mémoire aux variables se fait une fois pour toute, pendant le Lancement de l'exécution de l'algorithme.

Conséquence 2. Lors de l'exécution de l'algorithme, uniquement les variables déclarées dans la partie **Déclaration** de l'algorithme, peuvent être manipulées par les instructions qui se trouvent dans la partie **Corps** de l'algorithme.

Conséquence 3. Toutes les variables que devrait manipuler l'algorithme doivent être connues lors de la conception de l'algorithme (design-time). Ce qui oblige le concepteur de l'algorithme à fixer ses besoins en variables pendant le design-time.

Cette dernière conséquence est en fait une contrainte entrainant une mauvaise gestion de l'allocation de l'espace mémoire. En effet :

- Les variables déclarées dans un algorithme existent en mémoire durant tout le temps d'exécution de l'algorithme. Si, par exemple, au cours du runtime certaines variables ne sont plus utilisées par l'algorithme, l'espace mémoire qui leur est associé demeure occupé jusqu'à la fin de l'exécution de l'algorithme.
- Le concepteur de l'algorithme doit déclarer ses variables en se basant sur le cas le plus défavorable. Par exemple, un algorithme prévu pour contenir un maximum de 500 valeurs doit déclarer une variable tableau de 500 cases mémoires, même si certaines exécutions ne nécessitent que quelques dizaines de valeurs.

⇨ Ce type de variables est dit **Statique**.

Deux questions se posent :

1. Comment créer et utiliser des variables au fur et à mesure des besoins de l'algorithme ?
2. Comment Détruire les variables qui ne sont plus utiles dans la suite de l'exécution de l'algorithme et récupérer l'espace mémoire qui leur était alloué ?

La réponse à ces deux questions passe par l'introduction d'un nouveau type de variables appelé : **Pointeur de** ou simplement **Pointeur.**

⇨ Ce type de variable permet une gestion **Dynamique** de l'espace mémoire.

2. Les pointeurs

Une variable pointeur est une variable dont le contenu (la valeur) est une adresse d'une cellule mémoire.

<var. pointeur> | @celluleMémoire |

Une variable pointeur permet de :

⇨ Mémoriser une adresse mémoire d'une autre variable, on dit que la variable pointeur pointe sur cette variable. Cette dernière est appelée **variable pointée**.

⇨ Accéder à la variable pointée pour pouvoir la traiter.

a. Déclaration d'une variable pointeur

Un nouveau type « **pointeur de** » permet la déclaration d'une variable pointeur :

```
var <nom de la variable pointeur> : pointeur de <type de base> ;
```

<type de base> est le type de base de la **variable pointée**. Il peut être réel, entier, caractère, logique, chaine de caractère, enregistrement, etc. On l'appellera **sous-type** de la variable pointeur.

Exemple

```
type Tdate=     Enregistrement
                j :1..12;
                m :1..31;
                a : entier;
                     FinEnregistrement;

var  pdate:     pointeur de Tdate;
     pe :            pointeur de entier ;
```

Cette déclaration alloue deux espaces mémoire nommés pdate et pe

Tels que :

- @Tdate est une adresse aléatoire d'une case mémoire de type Tdate.
- @**entier** est une adresse aléatoire d'une case mémoire de type **entier**.

Notons que les types Tdate et **entier** sont respectivement les **sous-types** des variables pointeurs pdate et pe.

b. La valeur Nil

Une variable pointeur peut ne pointer sur aucune cellule mémoire, on dit qu'elle ne pointe sur rien. Pour ce faire elle doit contenir une valeur spéciale **Nil**.

La valeur **Nil** peut être affectée dans toute variable pointeur quel que soit son sous-type.

Exemple

```
Pdate←Nil;
Pe←Nil;
```

Bien que Pdate et Pe aient des sous-types différents, elles peuvent toutes les deux recevoir la valeur NIL.

Après exécution des deux instructions précédentes on aura en mémoire centrale :

On peut également utiliser la représentation graphique suivante :

c. Les comparaisons entre pointeurs

- ⇨ Deux variables pointeurs p1 et p2 peuvent être comparées si et seulement si elles possèdent le même sous-type.
- ⇨ Dans ce cas les deux seules comparaisons permises sont l'égalité (=) et l'inégalité (<>).
- ⇨ La valeur **Nil** peut être comparée à toutes les valeurs pointeurs quels que soient leurs sous-types.

Pourquoi comparer des pointeurs ?

On compare deux variables pointeurs pour savoir si elles pointent sur la même variable pointée.

3. Gestion dynamique de la mémoire

La gestion dynamique de la mémoire est possible grâce à l'utilisation des variables pointées.

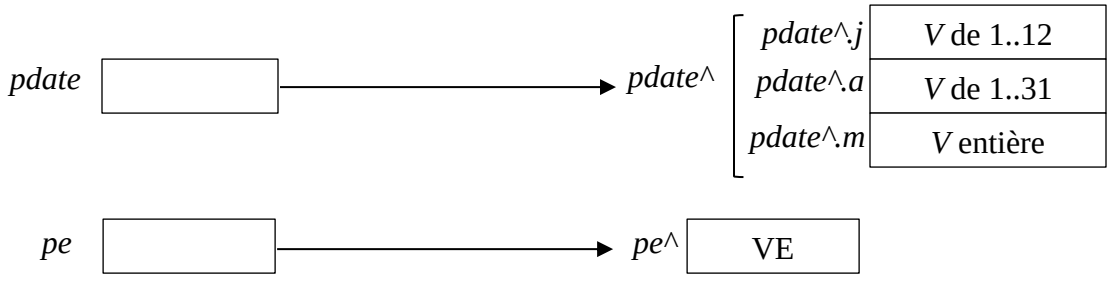
a. Création d'une variable pointée

Pour créer une variable pointée, on utilise l'instruction **Nouveau** dont la syntaxe est la suivante :

```
Nouveau (<id. de la variable pointeur>) ;
```

Exemple.
```
Nouveau (pdate) ;
Nouveau (pe) ;
```

Après l'exécution des deux instructions précédentes on aura en mémoire centrale :



Tels que, V est une valeur aléatoire et VE est une valeur aléatoire de type **entier.** Les variables pdate^ et pe^ sont des variables pointées dont on expliquera l'utilisation dans la section qui suit.

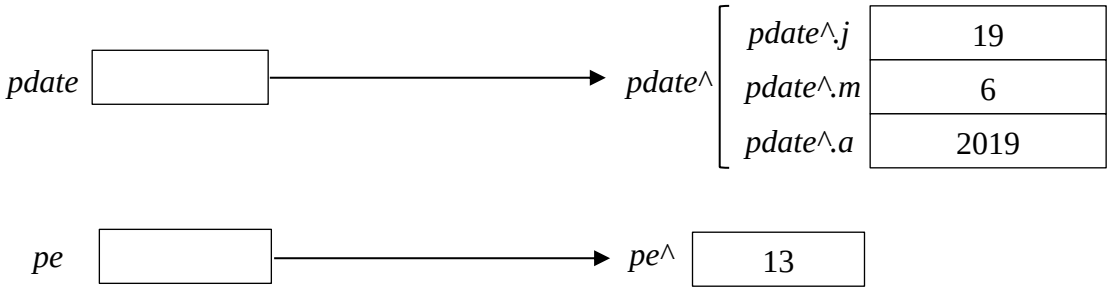
b. Utilisation d'une variable pointée

Pour utiliser le contenu d'une variable pointée, il faut indiquer le nom de la variable pointeur qui en contient l'adresse et coller à ce nom le signe ^.

Exemples

```
Lire(pe^) ;
pdate^.j←19 ;
pdate^.m←6 ;
pdate^.a←2019 ;
```

Après exécution des instructions précédentes on aura en mémoire centrale :



c. Suppression d'une variable pointée

Lorsqu'une variable pointée n'a plus d'utilité, il est possible de la supprimer et rendre disponible l'espace qu'elle occupait. Pour se faire on utilise l'instruction **Liberer** dont la syntaxe est la suivante :

```
Liberer (<id. de la variable pointeur>) ;
```

Exemple.

```
Liberer (pdate) ;
Liberer (pe) ;
```

Après l'exécution des deux instructions précédentes on aura en mémoire centrale :

d. Affectation d'un pointeur à un autre pointeur

Le contenu d'une variable pointeur (une adresse mémoire) peut être recopié dans une autre variable pointeur à l'aide de l'instruction d'affectation (←), à condition que les deux variables pointeurs soient du même sous-type.

Exemple 1

```
algorithme LA1 ;
var pe, pes : pointeur de entier ;
Debut
 Nouveau (pe) ;
 pe^←13;
 pes←pe ;
Fin.
```

L'exécution de LA1 est schématisée par la Figure 9.3.

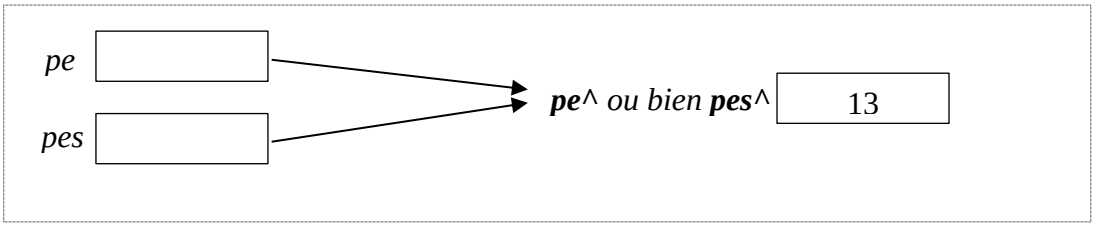


Figure 9.3 : Schéma d'exécution de l'algorithme LA1.

Exemple 2

```
algorithme LA2 ;
var N : entier ;
     p,q :pointeur de entier ;
Debut
N ←5 ; Nouveau(p) ; p^ ← N+1 ; Nouveau (q) ; q^ ← p^ +1 ; Ecrire
(p^) ; Liberer(p) ; p←q ;
Ecrire (p^,' ;',q^ ) ; Liberer(q) ;
Fin.
```

L'exécution de LA2 est schématisée par le Tableau 9.1.

Déclarations et Instructions	Cases mémoires					Ecran
	N	p	q	p^	q^	
Var N : entier ;	V.A.E	N'existe pas	N'existe pas	N'existe pas	N'existe pas	Rien
p,q :**pointeur de** entier ;	V.A.E	@.A.E	@.A.E	N'existe pas	N'existe pas	Rien
N ←5 ;	5	@.A.E	@.A.E	N'existe pas	N'existe pas	Rien
Nouveau(p) ;	5	@ p^	@.A.E	V.A.E	N'existe pas	Rien
p^ ← N+1 ;	5	@ p^	@.A.E	6	N'existe pas	Rien
Nouveau (q) ;	5	@ p^	@ q^	6	V.A.E	Rien
q^ ← p^ +1 ;	5	@ p^	@ q^	6	7	Rien
Ecrire (p^) ;	5	@ p^	@ q^	6	7	6
Liberer(p) ;	5	@.A.E	@ q^	Est perdu	7	6
p←q ;	5	@ p^ =@ q^	@ p^ =@ q^	7	7	6
Ecrire (p^,' ;',q^) ;	5	@ p^ =@ q^	@ p^ =@ q^	7	7	7 ;7
Liberer(q) ;	5	@.A.E	@.A.E	Est perdue	Est perdue	7 ;7

V.A.E : Valeur aléatoire entière.

@.A.E : Adresse aléatoire d'un entier.

Tableau 9.1 : Trace de de l'exécution de l'algorithme LA2.

La traduction de LA2 en Langage C ainsi que les explications y afférentes sont données dans Figure 9.4.

Traduction de LA2 en C	Explications

`# include <stdio.h>`
`int main()`
`{`
`int n;`
`int *p; int *q;`
`n=5;`
`p=malloc(sizeof(int));`
`*p = n +1;`
`q=malloc(sizeof(int));`
`*q=*p + 1;`
`printf("%d \n",*p);`
`free(p);`
`p = q;`
`printf("%d;%d\n",*p,*q);`
`free(q);`
`return 0;`
`}`
	• **`int *p`** ; (**`p`** est une variable qui contient l'adresse d'une variable entière, **`p`** est un **pointeur**) ⇔`var p : pointeur de entier ;`
	• De même pour **`int *q`**; ⇔ `var q : pointeur de entier` ;
	• **`p= malloc(sizeof(int));`** permet d'allouer la taille d'un entier pointé par **`p`**. (Pour allouer **`n`** octets en mémoire centrale en langage C, on écrit l'instruction : **`malloc(n);`**)
	• De même pour **`q=malloc(sizeof(int));`** permet d'allouer la taille d'un entier qui sera pointée par **`q`**.
	• **`*p = n +1;`** ⇔ `p^ ← N+1 ;` (La valeur de la variable pointée par le pointeur **`p`** est **`*p`**)
	• **`free(p)`**; ⇔ **`Liberer`**`(p) ;`

Figure 9.4 : Traduction en Langage C de l'algorithme LA2 accompagnée des explications y afférentes.

4. Les Listes linéaires chainées

Avant de donner une définition des listes linéaires chainées (llc), nous commençons par une introduction.

a. Introduction

Revenons à la -conséquence 3 de l'introduction du chapitre 3 Listes linéaires chainées- où nous avons énoncé que toutes les variables que devrait manipuler l'algorithme doivent être connues lors de la conception de l'algorithme obligeant le concepteur de l'algorithme à fixer ses besoins en variables pendant l'écriture de son algorithme.

Si par exemple, un algorithme est conçu pour contenir au maximum 150 valeurs entières, alors on doit déclarer une variable tableau de 150 cases mémoires.

Soit **T** ce tableau

	1	2	3	4	5	6	7	8		149	150
T	10	12	0	-5	11	10	13	18	…	56	20

Que faire ? Si :
Certaines exécutions ne nécessitent que quelques dizaines de valeurs ? ou bien
Le nombre de valeurs dépasse 150 ?

- Une chose est sûre c'est qu'on ne peut pas intervenir sur la taille du tableau pendant l'exécution du programme.

- Nous devons donc recourir aux variables dynamiques.

Les valeurs contenues dans **T** peuvent être vues comme un ensemble **E** de couples (**position**, valeur) :

E={(**1**,10), (**2**,12), (**3**,0) … (**149**,56), (**150**,20)}

Le couple (**position**, valeur) peut être définie par la structure de données enregistrement suivante :

En LA	En langage C

`type tCouple = Enregistrement`
`  valeur  : entier ;`
`  suivant : pointeur de tCouple ;`
`Fin enregistrement ;`
	`struct tCouple {`
	`  int valeur;`
	`  tCouple* suivant ;`
	`}`

Et l'ensemble **E** par une liste de ces couples :

En LA	En langage C
`var pt_Couples : pointeur de tCouple ;`	`tCouple*  pt_Couples`

La Figure 9.5 montre à quoi ressemblerait une structure de données représentant l'ensemble **E** suite à la déclaration de la variable `pt_Couples` et d'autres opérations sur les listes linéaires chainées qu'on verra dans la section -opérations sur les llc-).

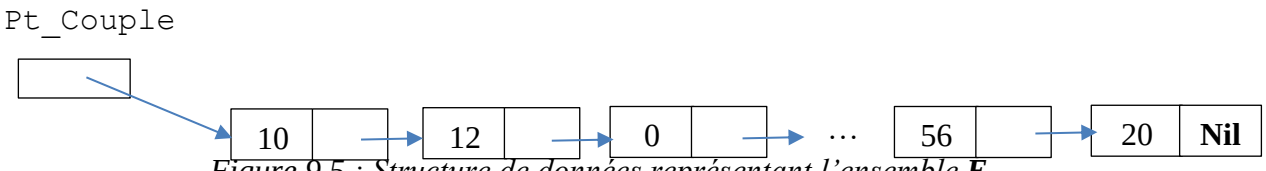


*Figure 9.5 : Structure de données représentant l'ensemble **E**.*

Cette structure de données appelée **liste linéaire chainée** (llc) nous permet de mémoriser cette liste de couples à l'aide des pointeurs. Ces couples sont appelés des **cellules** ou des **maillons**.

b. Définition

Une liste linéaire chainée (llc) est constituée d'un ensemble d'éléments appelés **cellules** ou **maillons** dont la structure de données est généralement un enregistrement ayant des champs qui contiennent des valeurs et au moins un champ qui contient l'adresse de la cellule suivante. La Figure 9.6 schématise une représentation conceptuelle d'une llc.

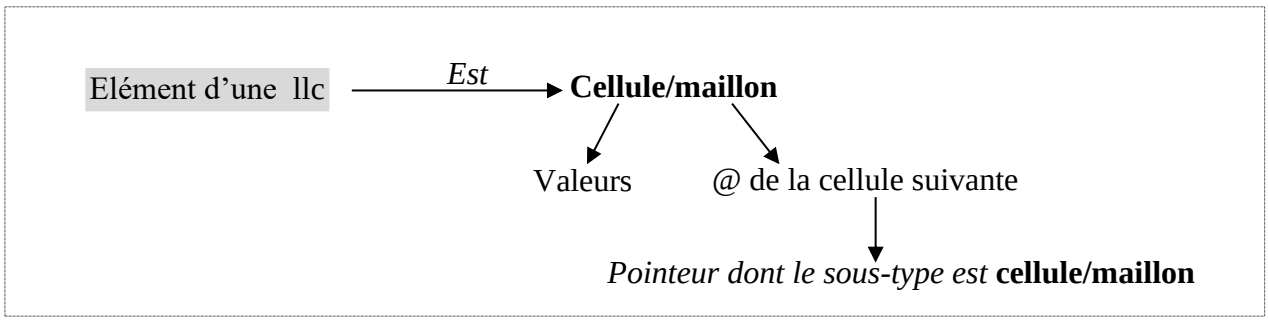


Figure 9.6 : Représentation conceptuelle d'une llc.

Exemple

Soit la définition et la déclaration suivantes :

```
type tCellule = Enregistrement
                  v : entier ;
                  suiv : pointeur de tCellule ;
                Fin enregistrement ;
var pt_llc : pointeur de tCellule ;
```

Après création de cette liste et ajout de 4 cellules (on verra l'ajout des cellules dans la section -opérations sur les llc-), on aura la llc comme le montre la Figure 9.7.

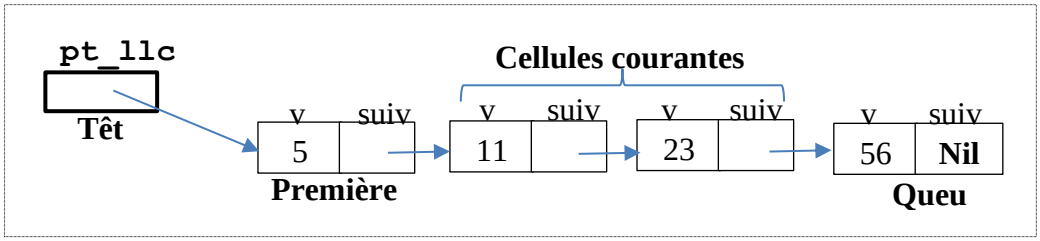


Figure 9.7 : llc contenant 4 cellules.

Le pointeur **`pt_llc`** est la tête de la llc. Il contient l'adresse de la première cellule. Le Tableau 9.2 montre comment, à partir de **`pt_llc`** on peut accéder à toutes les cellules de la llc.

Cellules	Valeurs des cellules	Adresses des cellules
1ère cellule	**pt_llc**^.v (=5)	**pt_llc**
2ième cellule	**pt_llc**^.suiv^.v (=11)	**pt_llc**^.suiv
3ième cellule	**pt_llc**^.suiv^.suiv^.v (=23)	**pt_llc**^.suiv^.suiv
4ième cellule	**pt_llc**^.suiv^.suiv^.suiv^.v (=56)	**pt_llc**^.suiv^.suiv^.suiv
pt_llc^.suiv^.suiv^.suiv est l'adresse de la 5ième cellule		

Tableau 9.2 : Accès à toutes les cellules de la llc à partir de la tête de liste.

Nous remarquons que le fait d'accéder à toutes les cellules de la llc à partir de la tête de la liste n'est plus faisable surtout le nombre de cellules de la llc est important.

Pour pallier cette insuffisance et pouvoir parcourir toutes les cellules de la llc d'une manière automatique quel que soit le nombre de ses cellules, nous déclarons une nouvelle variable `pt_parcours` de type `tCellule` que nous initialisons à `pt_llc` (tête de liste), (2) nous utilisons une boucle de parcours pour faire passer `pt_parcours` d'une cellule vers la suivante. La Figure 9.8 montre cette une boucle de parcours.

```
pt_parcours←pt_llc; //nous mettons dans pt_parcours l'adresse de la première cellule.
Tant que pt_parcours <> Nil
Faire   (* Instructions *)
        pt_parcours ← pt_parcours^.suiv ;
FFTantque ;
```

Figure 9.8 : Boucle de parcours d'une llc.

c. Remarques.

- Une llc doit posséder une variable appelée **Tête** (`pt_llc` pour notre exemple) qui mémorise l'adresse de la **Première cellule** de la liste.
- Si la valeur de **Tête** est égale à **Nil** alors la llc est **vide**.
- Si la valeur de **Tête** est différente de **Nil** alors la valeur du pointeur de la dernière cellule appelé **Queue** doit avoir la valeur **Nil**. La Queue est le dernier élément de la llc.
- La perte de **Tête** entraine la perte de la llc.
- Chaque cellule de la llc doit être reliée à la cellule suivante par un pointeur (le champ `suiv` pour notre exemple).

5. Opérations sur les listes linéaires chainées

Dans cette section nous abordons les deux opérations les plus importantes sur une liste chainée, à savoir :

a. L'insertion ou l'ajout d'une cellule dans une llc.
b. La suppression d'une cellule d'une llc.

Notons que les insertions successives de cellules permettent de **créer** une llc, qui au départ ne contenait qu'une tête de liste (llc **vide**).

a. Insertion d'une cellule dans une llc

L'insertion d'une cellule dans une llc est à considérer en :

1. Tête de liste
2. Milieu[5] de liste
3. Fin de liste

Au fait, les opérations d'insertion en milieu de liste et en fin de liste reviennent au même, comme nous le verrons par la suite.

i. Insertion en tête de liste

Cette opération consiste à insérer une cellule de sorte qu'elle devienne la première cellule.

Soit `pt_ncellule` le pointeur qui contient l'adresse de la nouvelle cellule à insérer.

Les Figure 9.9 et Figure 9.10 schématisent cette opération, avant l'insertion et après l'insertion de la nouvelle cellule en tête de liste.

Avant insertion de la nouvelle cellule

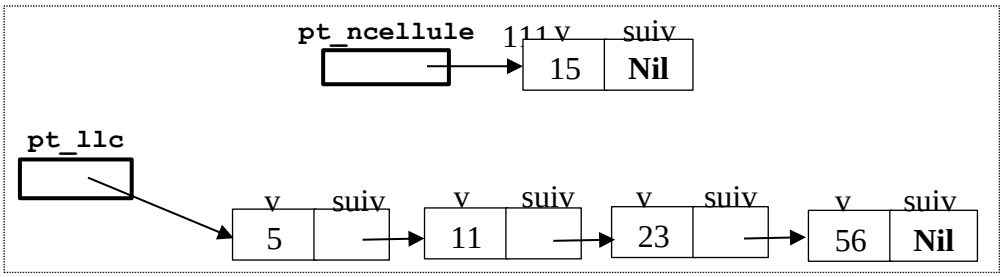


Figure 9.9 : Etat de la llc avant insertion de la nouvelle cellule en tête de liste.

Après insertion de la nouvelle cellule

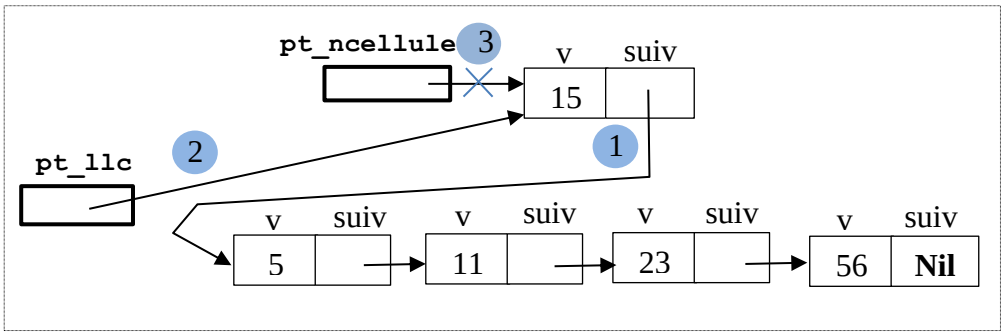


Figure 9.10 : Etat de la llc après insertion de la nouvelle cellule en tête de liste.

[5] Le milieu veut dire une position qui n'est ni la première ni la dernière de la liste

En se basant sur les Figures 9.9 et 9.10, nous pouvons écrire la procédure suivante, d'insertion d'une cellule en tête de liste, comme le montre la Figure 9.11 :

```
procedure  insertion_teteListe (var pt_ncellule : pointeur de
tCellule  ) ;
debut
     pt_ncellule^.suiv ← pt_llc ; // c'est 1
     pt_llc← pt_ncellule ; // c'est 2
     pt_ncellule ←Nil ; // c'est 3
fin ;
```

Figure 9.11 : Procédure d'insertion d'une nouvelle cellule en tête de liste dans une llc.

ii. Insertion en milieu de liste (fin de liste)

Cette opération consiste à insérer une cellule en milieu de la liste juste après une cellule que nous appellerons : **pt_ccellule** (qui représente le pointeur de courante cellule).

Soient **pt_ncellule** le pointeur qui contient l'adresse de la nouvelle cellule à insérer et **pt_ccellule** le pointeur qui contient l'adresse de la cellule après laquelle on va insérer **pt_ncellule**.

Les Figure 9.12 et Figure 9.13 schématisent cette opération, avant l'insertion et après l'insertion de la nouvelle cellule en milieu de liste.

Avant l'insertion

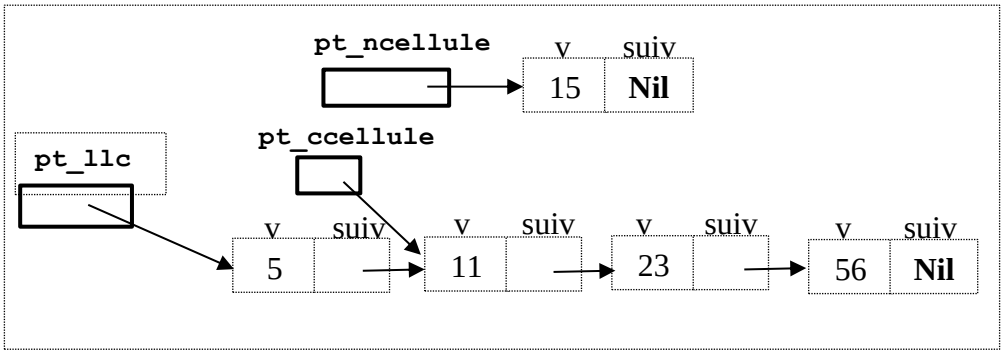


Figure 9.12 : Etat de la llc avant insertion de la nouvelle cellule en milieu de liste.

Après l'insertion

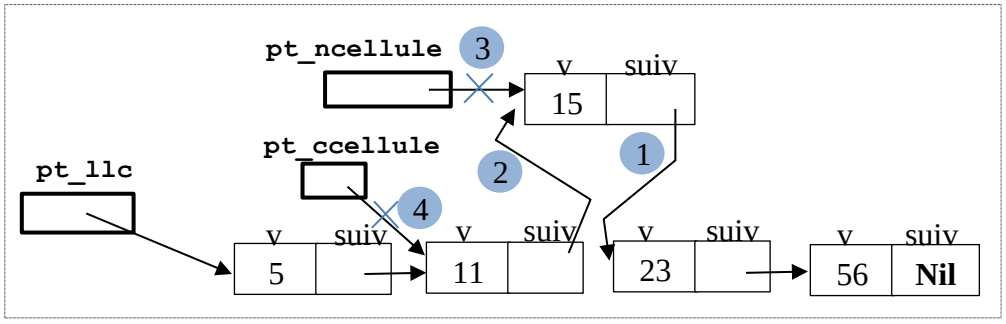


Figure 9.13 : Etat de la llc après insertion de la nouvelle cellule en milieu de liste.

En se basant sur les Figures 3.12 et 3.13 nous pouvons écrire la procédure suivante, d'insertion d'une cellule en milieu de liste, comme le montre la Figure 9.14.

```
Procedure insertion_milListe (var pt_ncellule, pt_ccellule: pointeur de tCellule);

debut
      pt_ncellule^.suiv ← pt_ccellule^.suiv ; // c'est 1
      pt_ccellule^.suiv ← pt_ncellule ; // c'est 2
      pt_ncellule ←Nil  ; // c'est 3
      pt_ccellule ←Nil ; c'est 4
fin ;
```

Figure 9.14 : Procédure d'insertion d'une nouvelle cellule en milieu de liste.

Dans le cas où on veut insérer la nouvelle cellule en fin de liste, le pointeur **pt_ccellule** aura l'adresse de la dernière cellule et on aura : **pt_ccellule^.suiv = Nil.** La procédure '**insertion_milListe**' sera équivalente à la suite d'instructions suivantes :

```
pt_ncellule^.suiv ← Nil
pt_ccellule^.suiv ← pt_ncellule ;
pt_ncellule ←Nil;
pt_ccellule ←Nil;
```

Nous voyons bien que la procédure '**insertion_milListe**' conçue pour insérer une nouvelle cellule en milieu de liste permet aussi d'insérer la nouvelle cellule en fin de liste. Nous pouvons ainsi dire que les opérations d'insertion en milieu de liste et en fin de liste reviennent au même.

b. Suppression d'une cellule d'une llc

Comme l'opération d'insertion d'une cellule dans une llc, la suppression d'une cellule d'une llc est à considérer en :

1. Tête de liste
2. Milieu[6] de liste
3. Fin de liste

iii. Suppression en tête de liste

Le principe est simple et consiste à :

1- Mémoriser dans un pointeur **Temp** l'adresse de la cellule qui suit la première cellule.
2- Libérer la première cellule repérée par le pointeur **pt_llc**.
3- Affecter à **pt_llc** l'adresse mémorisée par **Temp**.

Les suites d'instructions suivantes permettent la suppression en tête de liste, comme le montre Figure 9.15.

[6] Le milieu veut dire une position qui n'est ni la première ni la dernière de la liste

```
var Temp : pointeur de tCellule ;

Temp ← pt_llc ^.suiv;
Liberer (pt_llc);
pt_llc ← Temp ;
Temp←Nil) ;
```

Figure 9.15 : Suppression en tête de liste.

1. Suppression en milieu de liste (fin de liste)

Nous considérons la suppression d'une cellule d'une llc dans le cas où nous connaissons l'adresse de la cellule qui la précède. Cette adresse est repérée par le pointeur **pt_ccellule**. Nous aurons donc à supprimer la cellule repérée par le pointeur **pt_ccellule^.suiv**.

Les Figures 9.16 et Figure 9.17 schématisent cette opération, avant la suppression et après la suppression d'une cellule en milieu de liste.

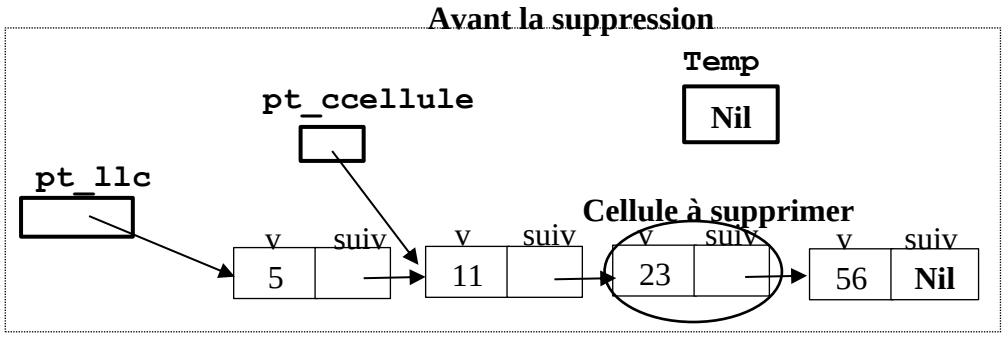


Figure 9.16 : Etat de la llc avant suppression d'une cellule en milieu de liste.

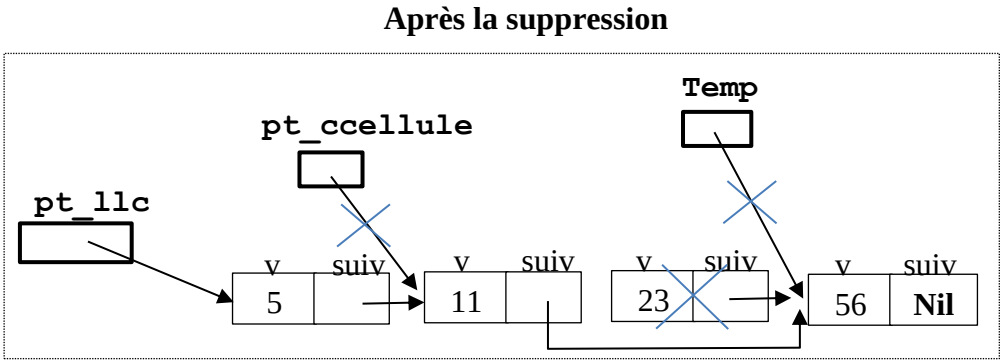


Figure 9.17 : Etat de la llc après suppression d'une cellule en milieu de liste.

En se basant sur les Figures 3.16 et 3.17 nous pouvons écrire la procédure de suppression d'une cellule en milieu de liste en connaissant l'adresse de la cellule qui la précède, comme le montre la Figure 9.18.

```
procedure supp_millc (var pt_ccellule : pointeur de tCellule);
 var Temp : pointeur de tCellule ;
 debut
  Si pt_ccellule^.suiv<>Nil //Car il n'existe pas de cellule qui succède à
                                pt_ccellule^.suiv
     Alors Temp ← pt_ccellule^.suiv^.suiv
           Liberer (pt_ccellule^.suiv) ;
           pt_ccellule^.suiv ← Temp ;
           pt_ccellule ←Nil ;
           Temp←Nil;
  Fsi ;
 fin ;
```

Figure 9.18 : Procédure de suppression d'une nouvelle cellule en milieu de liste.

Dans le cas où on veut supprimer la cellule en fin de liste, le pointeur **pt_ccellule** aura l'adresse de l'avant dernière cellule et on aura : **pt_ccellule^.suiv^.suiv = Nil.** La procédure '**supp_millc**' sera équivalente à la suite d'instructions suivantes :

```
Temp ← Nil;
Liberer (pt_ccellule^.suiv) ;
pt_ccellule^.suiv ← Nil;
pt_ccellule← Nil ;
Temp← Nil ;
```

Nous voyons bien que la procédure '**supp_millc**' conçue pour supprimer une cellule en milieu de liste permet aussi de supprimer une cellule en fin de liste. Nous pouvons ainsi dire que les opérations de suppression en milieu de liste et en fin de liste reviennent au même.

Exemple.

Avant de clore cette section nous donnons un exemple d'un programme en langage C qui permet de :

- Créer une liste linéaire chainée ou chaque cellule correspond à un article caractérisé par son nom et son prix.
- Afficher cette llc.

```
#include<stdio.h>

// Définition et déclaration des cellules
typedef struct article art;
struct article {
        char nom[30];
        float prix;
        art *suivant;
```

```
// Procédure d'affichage de la llc
void affichage_liste(){
  courant=tete;
  while(courant!=NULL){
        printf("(%s,%f), ",
(*courant).nom,(*courant).prix);
                courant = (*courant).suivant;
        }
```

```
        } ;

art *tete, *nouveau, *courant;

// Procédure de lecture d'une cellule
void lecture_info(art *pt_art){
        printf(" Nom article = ");
        scanf("%s",(* pt_art).nom);
        printf(" Prix =  ");
        scanf("%f",&(* pt_art).prix);
        } ;

// Procédure de création de la llc
void creation_liste(){
        int i,n;
        tete=NULL;
        printf("Combien d'articles  ? ");
        scanf("%i",&n);
        for(i=0;i<n;i++){
           nouveau=(art*)malloc(sizeof(art));
           lecture_info(nouveau);
           (*nouveau).suivant=tete;
           tete=nouveau;
          }
        printf("\n");
        }
```

```
  }

// Procédure principale main()
main(){
        creation_liste ();
        affichage_liste ();
        }
```

6. Les listes doublement chainées

Une liste doublement chainée est une liste où :

- Chaque cellule de la liste est constituée de 3 parties :
  (1) Un ou plusieurs champs de données
  (2) Un pointeur vers la cellule précédente de la liste qu'on appellera pt_pred
  (3) Un pointeur vers la cellule suivante de la liste qu'on appellera pt_suiv
- L'adresse de la cellule de début de la liste est repérée constamment par la variable pointeur : **`tete`**
- L'adresse de la cellule de la fin de la liste est repérée constamment par la variable pointeur : **`queue`**
- La première cellule de la liste n'a pas de prédécesseur et donc son pt_pred est **`Nil`**
- La dernière cellule de la liste n'a pas de successeur et donc son pt_suiv est **`Nil`**

Notons cependant qu'une Une liste doublement chainée est vide si et seulement si : **`tete = queue = Nil.`** La Figure 9.19 illustre le schéma d'une liste doublement chainée.

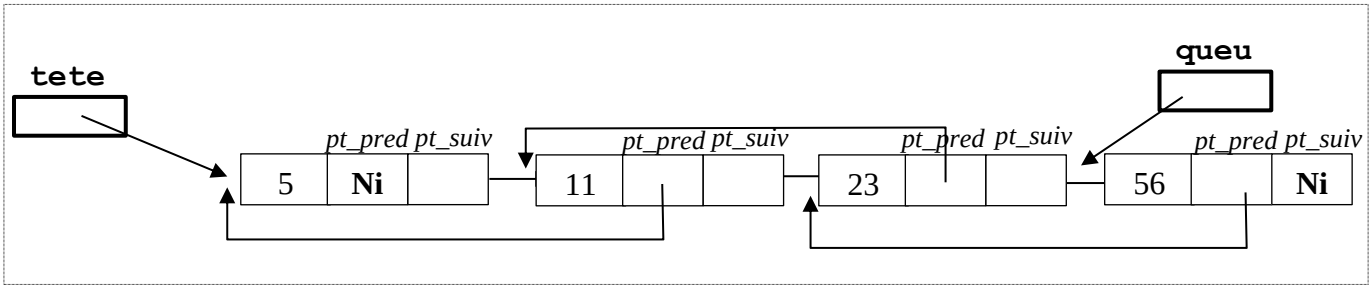


Figure 3.19 : Schéma d'une liste linéaire doublement chainée.

7. Les listes chaînées particulières

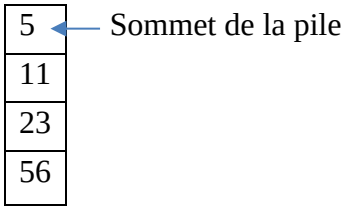
Dans cette section nous abordons deux types de listes particulières, à savoir :

a) Les piles
b) Les files.

a. Les piles

Une pile est un type de données basé le modèle de données liste, dans lequel les opérations sont réalisées à une extrémité de la liste, appelé **sommet de la pile**.

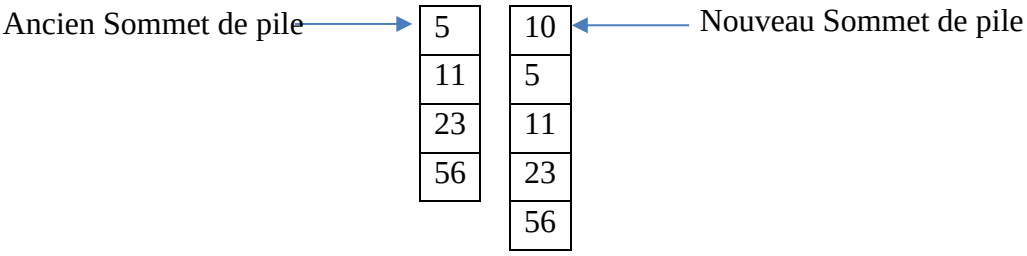
Une pile peut être représentée sous forme d'un tableau.



Les opérations principales qu'on peut faire sur les piles sont **`empiler()`** et **`depiler().`**

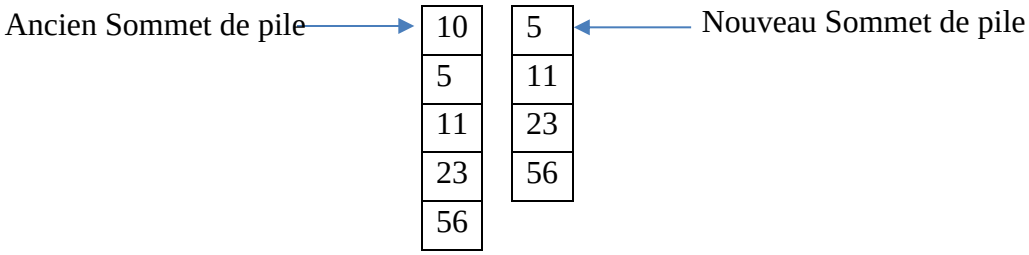
- **`empiler()`** : Ajoute un élément au sommet de la pile.

Par exemple **`empiler(`**`10`**`)`** donne le résultat suivant :



- **`depiler()`** : Supprime le sommet de la pile.

  Par exemple **`depiler`** donne le résultat suivant :



Les opérations d'ajouts (insertions) et suppressions se font à une seule extrémité qui est le sommet de la pile. Ainsi le terme **`LIFO`** (Last-in-first-out) est synonyme de pile.

Un exemple de pile est une pile d'assiette.

L'implémentation d'une pile se fait généralement par une liste chainée grâce aux opérations suivantes :

- **`empiler()`** revient à l'**insertion** d'une cellule en **tête de liste** (procedure **`insertion_teteListe`**) déjà vue dans la section Liste linéaires chainées.

- **`depiler()`** revient à la **suppression** d'une cellule en **tête de liste** déjà vue dans la section Liste linéaires chainées.

b. Les files

Une file est un type de données basé sur le modèle de données liste, dans lequel les éléments sont insérés à une extrémité : la **queue**, et retirés à une autre extrémité : la **tête**.

Le terme **`FIFO`** (First-in-first-out) est un synonyme de file.

Un exemple de file est une file d'attente devant un guichet.

L'implémentation d'une file se fait généralement par une liste chainée grâce aux opérations suivantes :

- **`enfiler()`** revient à l'**insertion** d'une cellule en **queue de liste** (procedure **`insertion_milListe`**, dans le cas où on veut insérer la nouvelle cellule en fin de liste) déjà vue dans la section Liste linéaires chainées.

- **`defiler()`** revient à la **suppression** d'une cellule en **tête de liste** déjà vue dans la section Liste linéaires chainées.

Bibliographie

Anne CANTEAUT. Programmation en langage C. INRIA-projet CODE (2003).

L.*BABA* HAMED. **Algorithmique et Structures de Données Statiques.** Office de Publication Universitaire (OPU). *Disponible dans la biblio du département informatique sous la cote: 04-08-08.*

Jacques TISSEAU. **Initiation à l'algorithmique.** Cours d'informatique S1. Ecole Nationale de BREST.

Bruno WARIN. **L'algorithmique votre passeport informatique pour la programmation**. ELLIPSES (15 juillet 2002). ISBN 2-7298-1140-0.

Djamel Eddine ZEGOUR. **Apprendre et enseigner l'algorithmique.** Tome 1 : Cours et annexes. Institut National d'Informatique Oued Smar-Alger-

Annexe

Cette annexe contient la solution en LA et en Langage C des exercices proposés dans les chapitres précédents

1. Série d'exercices -Introduction à l'informatique-

Pour les questions 1) et 2) se référer au cours.

3) Quelles sont les différences entre langages machine, assembleur et évolué ?

Nous donnons dans ce qui suit des définitions et explications concernant ces concepts.

- Le langage machine appelé aussi langage binaire, c'est avec ce langage que fonctionnent les ordinateurs. Il consiste à utiliser deux états (représentés par les chiffres 0 et 1) pour coder les informations (texte, images etc.).
  Dans le langage machine, le programmeur doit entrer chaque commande et toutes les données sous forme binaire c'est à dire sous forme de suite d'octets, chaque octet comportant une succession de 8 bits, chaque bit étant représenté par 1 ou 0.
- Le langage d'assemblage est un langage de bas niveau proche du langage machine qui peut être directement interprété par le microprocesseur de l'ordinateur tout en restant lisible par un humain.
  Le langage d'assemblage a été créé pour faciliter le travail des programmeurs. Il consiste à représenter les combinaisons de bits employées en langage binaire par des symboles faciles à retenir (mnémoniques) : Pour chaque instruction exprimée en langage machine, le programmeur code ses programmes en langage assembleur, ceux-ci sont ensuite transcrits par un logiciel appelé assembleur en langage machine, puis exécutés par l'ordinateur.
  La transformation du code assembleur en langage machine est accomplie par un programme nommé assembleur, dans le sens contraire par un programme appelé désassembleur. Les opérations correspondantes s'appellent respectivement assemblage et désassemblage.

- Langage évolué se situe au-dessus des langages de bas niveau (langage machine, langage d'assemblage).
- Ce langage est qualifié d'évolué, car il masque la complexité de la programmation.

A titre d'exemple citons comme langage évolué : Pascal, Java, C, C++, C#, Visual Basic (ou VB), Delphi, Python, Perl, PHP, JavaScript, VBscript, ASP etc.
La syntaxe des langages évolués est très simplifiée, on y trouve par exemple des mots en anglais (if, do while, switch, integer, string) donc il est plus accessible et compréhensible aux gens que l'assembleur lui même plus accessible que le langage machine. Dans les langages évolués, les commandes sont introduites à l'aide du clavier, à partir d'un programme en mémoire.

Un langage évolué est soit :

Un langage interprété : décodé et exécuté instruction par instruction lors de l'exécution du programme.
Un langage compilé : toutes les instructions sont traduites en code binaire avant d'être exécutées par un compilateur.

4) Qu'est qu'un système informatique ?

Un système informatique est avant tout un système.
Rappeler la définition d'un système : Un système est un ensemble d'éléments dynamiques en interaction, organisés pour un but commun. Informatique a été déjà définie au préalable –question 1-
Un système informatique est un système dans le but commun est le traitement automatique de l'information.
Un ordinateur composé d'éléments en interaction (clavier, souris, carte mère, disque dur, processeur, mémoire centrale, etc. (Hardware) et l'ensemble des programmes (Pilotes, Logiciel système d'exploitation, logiciel de traitement de donnée, de texte, etc. (Software)) dont le but est le traitement automatique de l'information est un système informatique.

Pour la question 5) se référer au cours.

6) A quoi sert une carte mère ?

La carte mère sert à interconnecter les composants d'un ordinateur

Pour la question 7) se référer au cours.

8) Identifier les périphériques suivants, et dire s'ils sont des périphériques d'entrée, de sorties, ou d'entrée /sortie.

Clavier (Entrée), Ecran (Sortie), Imprimante (Sortie), Lecteur DVD (Entrée), Scanner (Entrée), Webcam (Entrée), Graveur DVD (Entrée/Sortie), Souris (Entrée), Lecteur disquette (Entrée/Sortie), Lecteur CDROM (Entrée), Microphone (Entrée), Casque (Sortie), les enceintes (Sortie), Graveur CDROM (Entrée/Sortie), Connecteur USB (Entrée/Sortie).

9) Désigner dans le tableau suivant le type d'opération et le périphérique utilisé (Entrée, Sortie, stockage). :

Opération	**Type de l'opération**	**Périphérique**
Je tape un texte	Entrée	Clavier
Je scanne un graphe	Entrée	Scanner
J'imprime des résultats	Sortie	Imprimante
Je sauvegarde des données	Entrée	Mémoire secondaire
J'écoute de la musique à partir de mon PC	Sortie	Carte son

Pour les questions 10), 11), 12), 13), 14) et 15) se référer au cours.

16) Je suis dans la rue, et je veux rentrer chez moi (dans un immeuble ou il faut prendre l'ascenseur). Proposer un algorithme.

Les algorithmes proposés à ce niveau de cours sont des pseudo-algorithmes, ils ne suivent pas un formalisme. C'est à partir de la 2ème fiche de TD qu'ils devront utiliser le formalisme : Langage Algorithmique (LA) ou le Langage C. A ce stade un algorithme :

1. Est une suite d'actions ordonnées et finie s'exécutant l'une à la suite de l'autre
2. Une action est :
   - Simple : Verbe d'action(Instruction) + Complément(s)(Données)
     Ou bien
   - Conditionnelle :
     - Commence par SI (Condition logique entre parenthèses) et possèdent une partie ALORS (Le SINON n'est pas abordé à ce niveau de connaissance – pas d'instructions alternatives).
     - La partie ALORS est une action simple ou conditionnelle.
3. Commence par DEBUT et se termine par FIN.

Pseudo-algorithme

DEBUT

1. Aller jusqu'à la porte de l'immeuble
2. SI (la porte de l'immeuble est fermée) ALORS Ouvrir la porte de l'immeuble
3. Entrer dans l'immeuble
4. Fermer la porte de l'immeuble
5. Aller jusqu'à la porte de l'ascenseur
6. Appuyer sur le bouton d'appel de l'ascenseur
7. Entrer dans l'ascenseur
8. Appuyer sur le bouton indiquant le numéro de l'étage de l'appartement
9. Sortir de l'ascenseur
10. Aller jusqu'à la porte de l'appartement

FIN.

17) On cherche à résoudre dans R l'équation ax2+bx+c=0 (c.à.d. pour les données réelles a, b et c, calculer la ou les solutions de l'équation). Proposer un pseudo- algorithme.

Pseudo-algorithme

DEBUT

1. SI (a est égal à zéro) ALORS SI (b est égal à zéro) ALORS SI (c est égal à zéro) ALORS L'ensemble des réels est solution.
2. SI (a est égal à zéro) ALORS SI (b est égal à zéro) ALORS SI (c est différent de zéro) ALORS Il y a une erreur dans l'écriture de l'Equation
3. SI (a est égal à zéro) ALORS SI (b est différent de zéro) ALORS La solution est -b/2a
4. SI (a est différent de zéro) ALORS SI (b2 – 4ac est supérieur ou égal à zéro) ALORS Les solutions sont $(–b–\sqrt{}(b2 – 4ac))/2a$ et $(–b+\sqrt{}(b2 – 4ac))/2a$
5. SI a est différent de zéro ALORS SI (b2 – 4ac est inférieur à zéro) ALORS Il n'y a pas de solutions dans l'ensemble des réels.

FIN.

2. Série d'exercices -Algorithme séquentiel simple et traduction en langage C-

Exercice 1

Propositions	Résultat	Observation
Hauteur	Valide	Vérifie la syntaxe d'écriture d'un identificateur
épaisseur	Non valide	Contient la lettre accentuée (é)
POIDS	Valide	Vérifie la syntaxe d'écriture d'un identificateur
niveau_d'eau	Non valide	Contient l'apostrophe (')
_Longueur	Valide	Vérifie la syntaxe d'écriture d'un identificateur
longueur_plus_2	Valide	Vérifie la syntaxe d'écriture d'un identificateur
2_fois_longueur	Non valide	Commence par un chiffre (2)

Exercice 2

LA	Langage C

Algorithme affiche_texte_12 ;
Debut
ecrire('le nombre est 12') ;
Fin ;
	`#include <stdio.h>`
	`int main() {`
	`printf("le nombre est 12") ;`
	`return(0) ;`
	`}`

Algorithme affiche_nbre_12_et_succ ;
Début
ecrire(12, ' ', 12+1) ;
Fin ;
	`#include <stdio.h>`
	`int main() {`
	`printf("%d %d",12,12+1) ;`
	`return(0) ;`
	`}`

Algorithme affiche_calcul ;
var x : entier ;
Debut
x← 323- 17 ;
ecrire('Le resultat de calcul de 323 moins 17 est ', x) ;
Fin ;
	`#include <stdio.h>`
	`int main() {`
	`int x=323-17;`
	`printf("Le resultat de calcul de 323 moins 17 est %d",x) ;`
	`return(0) ;`
	`}`

Algorithme affiche_temps ;
var M, S : entier ;
Debut
M← 3*60+15 ;
S← M*60 ;
ecrire('Trois heures quinze minutes contiennent ', M, ' minutes ou bien ', S, 'secondes') ;
Fin ;
	`#include <stdio.h>`
	`int main() {`
	`int M=3*60+15,S=M*60;`
	`printf("Trois heures quinze minutes contiennent %d minutes ou bien %d secondes", M,S) ;`
	`return(0) ;`

Exercice 3

```
int n = 10, p = 4; long q = 2; float x = 1.75;
```

Expression	Valeur	Type
n+q	12	int
n+x	11.75	float
p+q	6	int
n<p	0	int
n>=p	1	int
n>q	1	int
q+3*(n>p)	5	int
q&&n	1	int
(q-2)&&(n-3)	0	int
x*(q==2)	1.75	float
x*(q=5)	8.75	float

Exercice 4

LA	Langage C

Algorithme affiche_nombre_saisi ;
var nbr :entier ;
Debut
ecrire('Veuillez entrer un nombre :') ;
lire(nbr) ;
ecrire('Votre nombre est : ', nbr) ;
Fin ;
	`#include <stdio.h>`
	`int main() {`
	`int nbr;`
	`printf("Veuillez entrer un nombre :");`
	`scanf("%d", &nbr) ;`
	`printf("Votre nombre est : %d",nbr);`
	`return(0) ;`
	`}`

3. Série d'exercices -Instructions conditionnelles, composées et de choix multiple-

Exercice 1

LA	Langage C

Algorithme affiche_nombre_saisi ;
var a, b :entier ;
Debut
ecrire('Entrer un nombre :') ;
lire(a) ;
ecrire('Entrer un autre nombre :') ;
lire(b) ;
si ((a<0) et (b<0)) ou ((a>0) et (b>0))
alors ecrire('Positif') ;
sinon si (a<>0) et (b<>0)
alors ecrire('Négatif') ;
Fsi ;
Fsi ;
Fin ;
	#include <stdio.h>
	int main() {
	int a,b;
	printf("Entrer un nombre :");
	scanf("%d", &a) ;
	printf("Entrer un autre nombre :");
	scanf("%d", &b) ;
	if (((a<0)&&(b<0)) || ((a>0)&&(b>0)))
	printf("Positif");
	else if (a!=0 && b!=0)
	printf("Negatif");
	return(0) ;
	}

Exercice 2

LA	Langage C

Algorithme tri_par_echange ;
var A, B, C, AIDE :entier ;
Debut
ecrire('Entrer 3 nombres entiers :') ;
lire(A,B,C) ;
si A<B
alors AIDE←A ;
A←B ;
B← AIDE ;
fsi ;
si A<C
alors AIDE←A ;
A←C ;
C← AIDE ;
fsi ;
si B<C
alors AIDE←B ;
B←C ;
C← AIDE ;
fsi ;
 	
ecrire('Après le tri A=',A, ' B=',B,' C=', C) ;
Fin ;
	#include <stdio.h>
	main()
	{
	int A, B, C, AIDE;
	printf("Entrez 3 nombres entiers :");
	scanf("%i %i %i", &A, &B, &C);
	printf("Avant le tri : \tA = %i\tB = %i\tC = %i\n", A, B, C);
	if (A<B)
	{
	AIDE = A;
	A = B;
	B = AIDE;
	}
	if (A<C)
	{
	AIDE = A;
	A = C;
	C = AIDE;
	}
	if (B<C)
	{
	AIDE = B;
	B = C;
	C = AIDE;

	}
	printf("Apres le tri : \tA = %i\tB = %i\tC = %i\n", A, B, C);
	return 0;
	}

Exercice 3

LA	Langage C

Algorithme calculette ;
var x,y,res : reel ;
op :caractere ;
erreur :logique ;
Debut
ecrire('Operation(+ - * /): ') ;
lire(op) ;
ecrire(' Premier Opérande: ') ;
lire(x) ;
ecrire(' Deuxième Opérande: ') ;
lire(y) ;
erreur←faux ;
selon op
'*' : res←x*y ;
'/' : si y<>0
alors res←x/y ;
sinon ecrire('Erreur /' !) ;
erreur←vrai ;
fsi ;
'+' : res←x+y ;
'-' : res←x-y ;
Sinon erreur←vrai ;
ecrire('Erreur opérateur !') ;
Fselon ;
si not(erreur)
alors ecrire('Résultat :', res) ;
fsi ;
Fin ;
	#include <stdio.h>
	void main(void) {
	float x,y, res;
	char op; int impos=0;
	printf("Operation(+ - * /): ");
	scanf("%c", &op);
	printf("Premier operande: ");
	scanf("%f", &x);
	printf("Deuxieme operande: ");
	scanf("%f", &y);
	switch (op)
	{
	case '*': res = x * y; break;
	case '/': if(y!=0) res = x / y;
	else {
	printf("Erreur /!\n");
	impos=1;
	}break;
	case '+': res = x + y; break;
	case '-': res = x - y; break;
	default: impos = 1;
	printf("Erreur Operateur!\n");
	}
	if(impos ==0)
	printf("Resultat : %f\n", res);
	return 0;
	}

4. Série d'exercices -Instructions de Boucle-

Exercice 1

LANGAGE C

```
#include <stdio.h>
int main ()
{
    int a,i,stop,temoin;
    printf("a ? ");
    scanf("%d",&a);
    stop=a/2;
    i=2;
    temoin=1;
    while ((i<=stop)&& temoin )
    {
        if (a%i==0)
        {
            printf("\n%d n'est pas premier car il est divisible par %d
qui donne un resultat = %d\n ",a,i,a/i);
            temoin=0;
        }
        else
            i+=1;
    }
    if (temoin)
       printf("\n%d est premier\n",a);
    return 0;
}
```

Exercice 2

LANGAGE C

```
#include <stdio.h>
int main(){
     int n=0;
     float a,b;
     printf("Entrer a : ");
     scanf("%f",&a);
     printf("Entrer b : ");
     scanf("%f",&b);
     while(a<b){
          a=a+a*0.08;
          b=b+b*0.05;
```

```
     n++;
     }
     printf("Le nombre d'annee necessaire est : %d",n);
     return 0;
}
```

Exercice 3

LANGAGE C

```
#include <stdio.h>
int main ()
{
    int a,sa,i,max;
    printf("a ? ");
    scanf("%d",&a);
    sa=a;
    i=1;
    max=0;
    while ((sa/10)!=0)
    {
        i+=1;
        if (max<(sa%10))
            max=sa%10;
        sa=sa/10;

    }
    printf("\nle nombre de chiffres dans %d est %d\n",a,i);
    printf("\nle max des chiffres dans %d est %d\n",a,max);
    return 0;
}
```

5. Série d'exercices – Tableaux et Chaines de caractères -

Exercice 1

LANGAGE C

```
#include <stdio.h>
#define N 10
main()
{
    int T[N], nbe,i;
    do
    {
     printf("Le nombre d'elements du tableau est:");
     scanf("%d",&nbe);
    }
    while (nbe>N || nbe<1);
    for (i=0; i<nbe; i++)
    {
        printf("T[%d]=",i+1);
        scanf("%d",&T[i]);
    }
    printf("\n\n La liste des %d elements que vous avez saisis
est:\n",nbe);
    for (i=0; i<nbe; i++)
      printf("\nT[%d]=%d",i+1,T[i]);

return 0;
}
```

Exercice 2

LANGAGE C

```
#include <stdio.h>
int main()
{
 const int  n=5;
 const int m=4;
 int mat[n][m],mati[n][m];
 int i=0,j=0,s=0;

// Initialisation de la matrice à zéro
 for (i=0;i<n;i++)
    for (j=0;j<m;j++)
```

```
        mat[i][j]=0;
// Affichage de la matrice
 for (i=0;i<n;i++)
    {for (j=0;j<m;j++)
        printf("%d ",mat[i][j]);
     printf("\n");
    }

// Saisie de la matrice
 for (i=0;i<n;i++)
    for (j=0;j<m;j++)
     {
         printf("mat[%d,%d]= ",i+1,j+1);
         scanf("%d",&mat[i][j]);
     }
// Affichage de la matrice
for (i=0;i<n;i++)
    {for (j=0;j<m;j++)
        printf("%d ",mat[i][j]);
     printf("\n");
    }
// Calcul de la somme des elements de la matrice
for (i=0;i<n;i++)
    for (j=0;j<m;j++)
        s=s+mat[i][j];
printf("\n S= %d\n",s);
return 0;
}
```

6. Série d'exercices – Enregistrements et tableaux d'enregistrements -

Exercice 1

LA

```
Algorithme personne x ;
        type personne = enregistrement
                                nom : chaine [15] ;
                                prenom : chaine [15] ;
                                age : entier ;
                        fin ;
        Var pers1, pers2 : personne ;
 Début
        ecrire ('faire entrer le nom, prénom et l'âge de la 1ière personne') ;
        lire(pers1 •nom, pers1 •prenom,pers1 • age) ;
        ecrire ('faire entrer le nom, prénom et l'âge de la 2ième personne') ;
        lire(pers2 •nom, pers2 •prenom,pers2 • age) ;
        si (pers1 • age > pers2 • age)
                alors ecrire ('la différence d'âge est', pers1 • age – pers2 • age)
                sinon ecrire ('la différence d'âge est', pers2 • age – pers1 • age) ;
        fsi ;
fin.
```

LANGAGE C

```
#include <stdio.h>
struct personne {
    char nom[25] ;
    char prenom[25] ;
    int age ;
} ;
int main() {
    struct personne pers1 ,pers2;
    printf (" faire entrer le nom, prénom et l'âge de la 1ière
personne : ");
    scanf ("%s%s%d", &pers1.nom,&pers1.prenom,&pers1.age);
    printf (" faire entrer le nom, prénom et l'âge de la 2ième
personne : ");
    scanf ("%s%s%d", &pers2.nom,&pers2.prenom,&pers2.age);
if (pers1.age>pers2.age)
    printf("la différence d'âge est:%d",pers1.age- pers2.age);
else printf("la différence d'âge est:%d",pers2.age- pers1.age);
return 0;
}
```

Exercice 2

LA

```
Algorithme tab-enreg ;
        type    etudiant = enregistrement
                        id : entier ;
                        nom, prenom : chaine [30] ;
                        groupe : 1..4;
                        note : Réel ;
                        fin ;
                tab = tableau [1..180] de etudiant ;
        var etd : tab; i : entier;
Début /*Lecture du tableau étudiant*/
        pour i de 1 à n
                faire   ecrire(‘ Saisir l'identificateur de l’étudiant N° :’,i)
                        lire (etd[i].id) ; écrire(‘Saisir le nom de l'étudiant N°:’,i);
                        lire(etd[i].nom);
                        ecrire(‘Saisir le prénom de l'étudiant N°:’,i);
                        lire(etd[i].prenom);
                        ecrire(‘Saisir le groupe de l'étudiant N°:’,i);
                        lire(etd[i].groupe);
                        ecrire (‘Saisir la note de l'étudiant N°:’,i);
                        lire(etd[i].note);
                ffpour ;
        pour i de 1 à n
                faire
                        si (etd[i].groupe = 2) et (etd[i].note < 10)
                                alors ecrire (etd[i]. nom, etd[i].prenom) ;
                        fsi ;
                ffpour ;
fin.
```

```
#include
# define N 100
struct etudiant {
     int id;
     char nom[30];
     char prenom[30];
     int groupe;
     float note;
};
int main() {
int i;
struct etudiant etd[N];
for (i=0;i<N, i+1){
  printf("Saisir l'identificateur de l'étudiant N°:%d\n",i+1);
  scanf("%d",&etd[i].id);
  printf("Saisir le nom de l'étudiant N°:%d\n",i+1);
  scanf("%s",etd[i].nom);
  printf("Saisir le prénom de l'étudiant N°:%d\n",i+1);
  scanf("%s",etd[i].prenom);
  printf("Saisir le groupe de l'étudiant N°:%d\n",i+1);
  scanf("%d",&etd[i].groupe);
  printf("Saisir la note de l'étudiant N°:%d\n",i+1);
  scanf("%f",&etd[i].note); }
  for (i=0;i<N, i+1){
       if ((etd[i].groupe==2)&&(etd[i].note<10))
  printf("Nom est:%s et le prénom est:%s\n",etd[i].nom,etd[i].prenom);
}
}
```

7. Série d'exercices – Les sous-programmes : les fonctions et les procédures-

Exercice 01

En utilisant des procédures/fonctions, écrire un algorithme qui permet de calculer $e^y + e^z$, sachant que :

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + x^4/4! + ... + x^n/n! \ , \forall x \ \text{(Euler)}$$

LA	Langage C

algorithme somme_exponentielle;
 	
var x,y, somme_expo : reel ;
n : entier ;
fonction fact(n :entier) :entier ;
var i,f :entier ;
debut
f←1 ;
pour i←2 jusqu'a n
faire
f←f*i;
ffpour;
retouner f ;
fin ;
 	
fonction puis(a :reel ;n :entier) :reel ;
var i:entier ;
p: reel;
debut
p←a ;
pour i←2 jusqu'a n
faire
p←p*a;
ffpour;
retouner p ;
fin ;
 	
fonction expo(a :reel ;n :entier) :reel ;
var i:entier ;
ex: reel;
debut
ex←1 ;
pour i←1 jusqu'a n
faire
ex←ex+ **puis**(a,i)/**fact**(i);
ffpour;
retouner ex ;
fin ;
 	
debut
	#include <stdio.h>
	int fact(int n)
	{
	int i, f=1;
	for (i=2; i<=n; i++)
	{
	f=f*i;
	}
	return f;
	}
	
	float puis(float a,int n)
	{
	int i;
	float p=a;
	for (i=2; i<=n; i++)
	{
	p=p*a;
	}
	return p;
	}
	
	float expo(float a, int n)
	{
	float ex;
	int i;
	ex=1;
	for (i=1; i<=n; i++)
	{
	ex=ex+ puis(a,i)/fact(i);
	}
	return ex;
	}
	int main()
	{
	float x,y,somme_expo;
	int n;
	printf("\nDonner x:");scanf("%f",&x);
	printf("\nDonner y:");scanf("%f",&y);
	printf("\nDonner le rang
	:");scanf("%d",&n);
	somme_expo=expo(x,n)+expo(y,n);
	printf("\nExponentiel %.2f + Exponenteil
	%.2f est :%.2f",x,y,somme_expo);

ecrire('Donner x : '); lire (x);
ecrire('Donner y : '); lire (y);
ecrire('Donner le rang n : '); lire (n);
somme_expo= **expo(**x**,**n**)**+**expo(**y**,**n**)**;
ecrire(somme_expo) ;
fin.
	return 0;
	}

Exercice 02

En utilisant des procédures/fonctions, écrire un algorithme qui permute le nombre réel maximum avec le nombre réel minimum d'un vecteur contenant n (n>0) nombres réels.

```
algorithme Permut_max_min;
  const n= ;
  type tTableau= tableau[1..n] de reels ;
  var tab : tTableau ;
      i : entier ;

  fonction posMax(var t : tTableau) :entier ;
    var i,pmx :entier ;
     debut
      pmx← 1 ;
        pour i←2 jusqu'a n
          faire
             si t[pmx]<t[i]
                alors pmx← i;
              fsi ;
        ffpour;
        retouner pmx ;
    fin ;

   fonction posMin(var t : tTableau) :entier ;
    var i,pmn :entier ;
     debut
      pmn← 1;
        pour i←2 jusqu'a n
          faire
             si t[pmn]>t[i]
                alors pmn← i;
              fsi ;
        ffpour;
        retouner pmn ;
     fin ;

  procedure permuter(var t : tTableau ;pmx,pmn :entier);
    var intermediaire:reel ;
     debut
      intermediaire←t[pmx] ;
      t[pmx] ←t[pmn] ;
```

t[pmn]←intermediaire ;
fin ;

debut
(* *Saisie des éléments du vecteur* *)
Pour i←1 jusqu'a n **faire** lire(tab[i]) ; **ffpour** ;

(*Appel de la *procédure permuter**)
permuter(tab, **posMax(**tab**)**, **posMin(**tab**))**;

fin.

Exercice 03

LA	Langage C

algorithme Rech_Max_Recrusivement;
const n= ;
type tTableau= tableau[1..n] de reels ;
var tab : tTableau ;
i : entier ;
fonction ChercherMax(T: tTableau; indiceInit:entier; tailleVecteur:entier; Max:reel) : reel;
 	
debut
si indiceInit = 1
alors Max ←T[1];
fsi;
 	
si Max < T[indiceInit]
alors Max ← T[indiceInit];
fsi ;
 	
si indiceInit < tailleVecteur
alors Max ← ChercherMax(T, indiceInit+1, tailleVecteur, Max);
fsi ;
retourner Max ;
fin ; (**Fin de la fonciton ChercherMax**)
 	
debut
(* *Saisie des éléments du vecteur* *)
Pour i←1 jusqu'a n **faire** lire(tab[i]) ; **ffpour** ;
 	
(*Ecriture du Maximum du tableau en faisant Appel à la *fonction* **ChercherMax** *)
	`#include<stdio.h>`
	
	`float maxVecteur(float v[],int taille, int posCourante, float max)`
	`{`
	`if (posCourante==0)`
	`max=v[0];`
	`if (max<v[posCourante])`
	`max=v[posCourante];`
	`if (posCourante<taille-1)`
	
	`max=maxVecteur(v,taille,posCourante+1,max);`
	`return max;`
	`}`
	`int main()`
	`{`
	`int const n1=4 , n2=6;`
	`float v1[n1], v2[n2];`
	`int i;`
	`printf(" \n---- Vecteur 1 ----------\n");`
	`for (i=0;i<n1;i++)`
	`{`
	`printf("v1[%d]= ",i);`
	`scanf("%f",&v1[i]);`
	`}`
	`printf("\nLe maximum est :%.2f",maxVecteur(v1,n1,0,0));`
	
	`printf(" \n---- Vecteur 2----------\n");`
	`for (i=0;i<n2;i++)`
	`{`

Ecrire('La plus grande valeur dans le vecteur V est : ', **ChercherMax(** V**,** 1**,** N**,** 0**))**;
 	
fin. (**Fin de l'algorithme**)
	`printf("v2[%d]= ",i);`
	`scanf("%f",&v2[i]);`
	`}`
	`printf("\nLe maximum est :%.2f",maxVecteur(v2,n2,0,0));`
	`return 0;`
	
	`}`