CHAPITRE III

Bases d’algorithme et de programmation en C

Chapitre III : Bases de l'algorithme et de programmation en C

III.1. Objectif de ce chapitre

A l'issu de ce chapitre, l'apprenant sera capable de :

- Comprendre la définition et le rôle des algorithmes dans la résolution de problèmes informatiques.
- Identifier les différentes étapes d'un algorithme, de la spécification à l'exécution.
- Appréhender les concepts de variables, de types de données et d'opérations élémentaires dans la programmation.
- Analyser des exemples concrets d'algorithmes simples et comprendre comment ils sont traduits en programmes exécutables.

III.2. Concept d'un algorithme

- Le mot « Algorithme » est inventé par le mathématicien « AL-KHAWARISMI ». Un Algorithme est l'énoncé d'une séquence d'actions primitives réalisant un traitement. Il décrit le plan ou les séquences d'actions de résolution d'un problème donné.
- Un algorithme est un ensemble d'*actions* (ou d'*instructions*) séquentielles et logiquement ordonnées, permettant de transformer des données en entrée (*Inputs*) en données de sorties (*outputs* ou les *résultats*), afin de résoudre un problème.

Donc, un algorithme représente une solution pour un problème donné. Cette solution est spécifiée à travers un ensembles d'instructions (séquentielles avec un ordre logique) qui manipulent des données. Une fois l'algorithme est écrit (avec n'importe quelle langues : français, anglais, arabe,*etc.*), il sera transformé, après avoir choisi un langage de programmation, en un programme code source qui sera compilé (traduit) et exécuté par l'ordinateur.

Pour le langage de programmation qui sera utilisé, ça sera le langage ***C***.

III.3. La démarche et analyse d'un problème

Comme vu dans le point précédent, un algorithme représente une solution à un problème donné. Pour atteindre à cette solution algorithmique un processus d'analyse et de résolution sera appliqué.Ce processus est constitué des étapes illustrées dans la figure III.1.

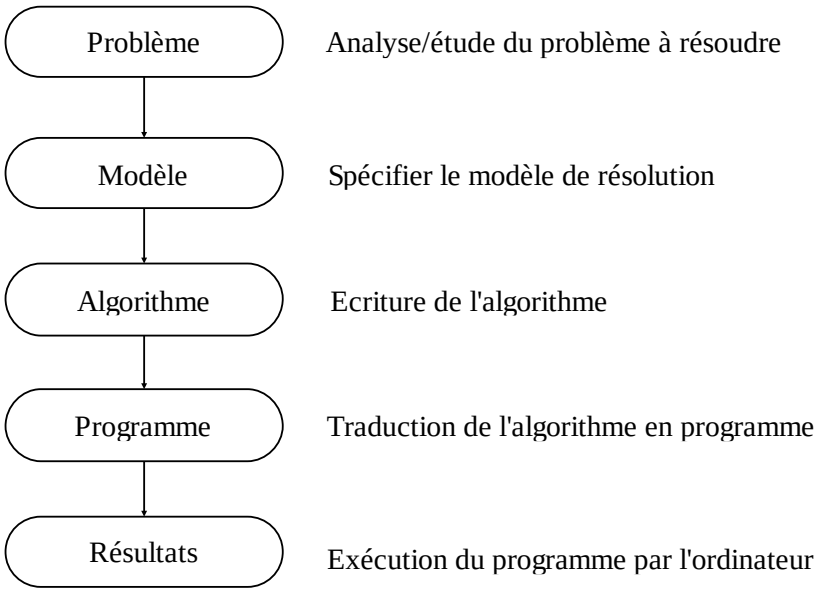


Figure III.1 : Etapes d'analyse d'un problème

III.4. Données, identificateurs et types de données simple

Un algorithme permet de réaliser un traitement sur un ensemble de données en entrées pour produire des données en sorties. Les données en sorties représentent la solution du problème traité par l'algorithme.

Un algorithme peut être schématisé comme suit :

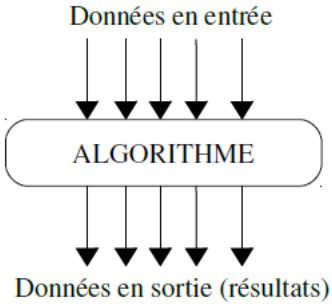


Figure III.2 : Structure d'un algorithme

Toute les données d'un programme sont des objets dans la mémoire vive (c'est un espace réservé dans la RAM). Chaque objet (espace mémoire) est désigné par une appellation dite : *identificateur.*

III.4.1. Notion d'identificateur

Un identificateur est une chaîne de caractères contenant uniquement des caractère alpha-numériques (alphabétiques de [a-z] et [A-Z] et numérique [0-9]) et tiré 8 '_' (trait souligné), et qui doit commencer soit par un une lettre alphabétique ou _.

Un identificateur permet d'identifier d'une manière unique un algorithme (ou un programme),

une variable, une constante, une procédure ou une fonction.

Dans un langage de programmation donnée, on a pas le droit d'utiliser les mots réservés (mots clés) du langage comme des identificateurs. Parmi les mots clés du langage C :

auto, break, case, char, continue, do, double, else, extern, float, for, goto, if, int, long, return short, sizeof, static, struct, switch, typedef, union, unsigned, while, …

Exemples :

Tableau III.1 : Exemples d'identificateurs valides et non valides

Identificateur valide	Identificateur non valide
a1 a_1 A_1 x12y x1_y	x1 y (à cause du blanc ou l'espace) x1-y (à cause du signe –(tiret de 6)) 1xy (commence par un caractère numérique)

III.4.2. Constantes et variables

Les données manipulées par un algorithme (ou un programme) sont soit des constantes ou des variables :

- ***Constantes :*** une constante est un objet contenant une valeur qui ne peut jamais être modifiée. Son objectif est d'éviter d'utiliser une valeur d'une manière direct. Imaginons qu'un algorithme utilise la valeur *3.14* une dizaines de fois (le nombre d'occurrences de la valeur *3.14* est par exemple *15*) et qu'on veut modifier cette valeur par une autre valeur plus précise : *3.14159*. Dansce cas on est amené à modifier toutes les occurrences de *3.14.* Par contre, si on utilise une constante *PI = 3.14* on modifier une seule fois cette constante.
- ***Variables :*** une variable est un objet contenant une valeur pouvant être modifiée.

Toute les données (variable ou constante) d'un algorithme possède un type de données (domaine de valeurs possibles).

III.4.3. Types de données

Dans l'algorithmique, nous avons cinq types de base :

- *Entiers* : représente l'ensemble {…, -4, -3, -2, -1, 0, 1, 2, 3, 4, ...}
- *Réels* : représente les valeurs numériques fractionnels et avec des virgule fixes (ou flottante)
- *Caractères* : représente tous les caractères imprimable.

- *Chaînes de caractères* : une séquence d'un ou plusieurs caractères
- *Booléens (logique)* : représente les deux valeurs *TRUE* et *FALSE*.

Le tableau III.2 montre la correspondance entre les types d'algorithme et les types du langage de programmation Pascal.

Tableau III.2 : Correspondance Algorithme/Langage C

Algorithme	Langage C
Entier	int
Réel	float
Booléen	bool
Caractère	Char
chaîne	Tableau de char (char[ ])

Exemples :

Ci-dessous quelques exemples de déclarations de constantes et variables

const PI = 3.14 ; {constante réelle}

const charge = 1.6E-19; {charge de l'électron}

const charge = 1.6E-19; {masse de l'électron}

III.5. Structure d'un algorithme/programme

Un algorithme manipule des données, les données avant de les utiliser il faut les identifier et les déclarer en utilisant les identificateur. Un algorithme est constitué de trois parties :

- ***Entête :*** dans cette partie on déclare le nom de l'algorithme à travers un identificateur.
- ***Déclarations :*** dans cette partie on déclare toutes les données utilisées par l'algorithme.
- ***Corps :*** représente la séquence d'actions (instructions) Pour écrire un algorithme, il faut suivre la structure suivante :

Tableau III.3 : Structure d'un algorithme

Algorithme	Langage C
Algorithme <identificateur_algo> <Déclarations> **Début** <Corps> **Fin**.	**#include <stdio.h>** **int main ()** **{** <Déclarations> <Instructions>; **return 0 ;** **}**

❖ Remarques

- Pour commenter un programme en langage C, on écrit les commentaires entre /*…. */. Par exemple : /**Ceci est un commentaire*/*
- On peut aussi utiliser les commentaire ligne :
  // Ceci est un commentaire ligne
- En langage C, la partie déclarative peut être avant la fonction main, comme peut être dans la fonction main.
- En langage C, la fonction main est le point d'entrée du programme : la première qui sera exécutée.

III.5.1. Déclarations

Dans la partie déclaration, on déclare toutes les données d'entrées et de sorties sous forme de constantes et de variables.

- **Constantes**

Les constantes sont des objets contenant des valeurs non modifiables. Les constante sont déclarées comme suit :

<identificateur> = <valeur>;

Algorithme		Langage C
PI = 3.14;	Constante réelle.	**const float** PI = 3.14;
MAX = 10;	Constante entière.	**const int** MAX = 10;
cc = 'a';	Constante caractère.	**const char** cc = "a";
ss = 'algo';	Constante chaîne de caractère.	**const char** ss[10] = "algo";
b1 = true;	Constante booléenne.	**const int** b1 = 1; .
b2 = false;	Constante booléenne.	**const int** b2 = 0;

- **Variables**

Les variables sont des objets contenant des valeurs pouvant êtes modifiées. Les variables sont déclarées comme suit :

<identificateur> : <type>;

Une variable appartient à un type de données. On a cinq types de données de base :

- *Entiers*
- *Réels*
- *Caractères*
- *Chaîne de caractères*
- *Booléens*, contenant deux valeurs : *True* ou *False* ;

Exemples :

Tableau III.4 : Exemples de variables

Algorithme	Langage C	Signification
x : réel	**float** x ;	variable réelle
n, m : entier	**int** n, m ;	deux variables entières
s : chaîne de caractères	**char** s[100];	variables chaîne de caractères
b1, b2, b3 : booleen	**bool** b1, b2, b3;	3 variables booléennes
c1 : caractère	**char** c1;	variable caractère

N.B : En plus des constantes et des variables, il est possible de déclarer de nouveaux types, des fonctions et des procédures.

III.5.2. Corps

Le corps d'un algorithme est constitué d'un ensemble d'actions / instructions ordonnées de manière séquentielle et logique. Ces instructions se divisant en cinq types distincts:

- ***Lecture* :** Cette opération consiste à introduire des données dans l'algorithme. Une lecture consiste à donner une valeur arbitraire à une variable.
- ***Écriture* :** Cette opération implique l'affichage de données. Elle permet d'afficher des résultats ou des messages.
- ***Affectation* :** Elle permet de modifier les valeurs des variables en leur assignant de nouvelles valeurs.
- ***Structures de contrôle* :** Ces structures permettent de modifier la séquentialité de l'algorithme. Elles sont utilisées pour sélectionner différents chemins d'exécution ou pour répéter un traitement.
    - *Structure de* ***Test alternatif simple / double***
    - *Structure* répétitives (itérative)
        - *La boucle* ***Pour***
        - *La boucle* ***Tant-que***
        - *La boucle* ***Répéter***

Dans le langage C, chaque instruction se termine par un **point-virgule**.

III.6. Types d'instructions

Tous les instructions d'un programme sont écrits dans son corps. (entre *Début* et *Fin*, en C entre

{ et } de la fonction main). On peut regrouper ces instructions en deux grandes types :

- Les instructions séquentielles simples : Entrées, sorties et affectation.
- Les structures de contrôles : tests et boucles.

III.6.1. Instructions d'Entrées/Sorties (Lecture / Écriture)

III.6.1.1. Entrées (Lecture)

Une instruction d'entrée nous permet dans un programme de donner une valeur quelconque à une variable. Ceci se réalise à travers l'opération de lecture.

La syntaxe et la sémantique d'une lecture est comme suit :

Tableau III.5 : La syntaxe et la sémantique d'une lecture

Algorithme	Langage C	Signification
Lire(<id_var>)	scanf (" %c", <id_var>);	Donner une valeur quelconque à la variable dont l'identifiant <*id_var*>.
Lire(<iv1>, <iv2>, …);	scanf (" %c %c ….", <iv1> , <iv2>,…);	Donner des valeurs aux variables <iv1>, <iv2>, *etc.*

❖ **Remarque :** Il est important de noter que l'instruction de lecture concerne uniquement les variables, on peut pas liredes constantes ou des valeurs. Lors de la lecture d'une variable dans un programme C, le programme se bloque en attendant la saisie d'une valeur via le clavier. Une fois la valeur saisie, on valide par la touche ***entrée***, et le programme reprend l'exécution avec l'instruction suivante.

Exemples :

Tableau III.6 : Exemples d'entrées

Algorithme	Langage C
Lire (a, b, c) lire (hauteur)	scanf("%d %d %d", &a, &b, &c); scanf("%f", &hauteur);

III.6.1.2. Sorties (Écriture)

Une instruction de sortie nous permet dans un programme d'afficher un résultat (données traitées) ou bien un message (chaîne de caractères). Ceci se réalise à travers l'opération d'écriture.

La syntaxe et la sémantique d'une écriture est comme suit :

Tableau III.7 : La syntaxe et la sémantique d'une écriture

Algorithme	Langage C	Signification
Ecrire(<id_var> | <id_const> | <valeur>, <expression>)	printf("%f", <id_var> | <id_const> | <valeur>, <expression>);	Afficher une valeur d'une variable, d'une constante, valeur immédiate ou calculée à travers une expression.

❖ **Remarque :** Il est à noter que l'instruction d'écriture ne concerne pas uniquement les variables, on peut écrire des constantes, valeurs ou des expressions (arithmétiques ou logiques).

Exemples :

Tableau III.8 : Exemples de sorties

Algorithme	Langage C	Signification
écrire("Bonjour") écrire(a, b, c)	printf ("Bonjour"); printf("%f %f %f", a, b, c);	Afficher le message Bonjour Afficher les valeurs des variables a, b et c
écrire(5+2)	printf("%f",5+2);	Afficher le résultat de la somme de 5 et 2 : afficher 7
écrire("La valeur de x : ", x)	printf("La valeur de x : %d", x);	

III.6.2. Instruction d'affectation

Une affectation consiste à donner une valeur (immédiate, constante, variable ou calculée à travers une expression) à une variable.

La syntaxe d'une affectation est :

Tableau III.9 : La syntaxe d'une affectation

Algorithme	Langage C
<id_varialbe> ← <valeur> | <id_variable> | <expression>	<id_varialbe> = <valeur> | <id_variable> | <expression> ;

Une affectation possède deux parties : la partie gauche qui représente toujours une variable, et la partie droite qui peut être : une valeur, variable ou une expression. La condition qu'une affectation soit correcte est que : la partie droite doit être du même type (ou de type compatible) avec la partie gauche.

Exemples :

Tableau III.10 : Exemples d'affectation

Algorithme	Langage C	Signification
a ←5 *b ← a+5* *sup ← a>b*	*a=5;* *b=a+5;* *sup=a>b;*	Mettre la valeur 5 dans la variable a Mettre la valeur de l'expression a+5 dans la variable B a>b donne un résultat booléen, donc sup est une variable booléenne

III.6.3. Structures de contrôles

En générale, les instructions d'un programme sont exécutés d'une manière séquentielle : la première instruction, ensuite la deuxième, après la troisième et ainsi de suite. Cependant, dans plusieurs cas, on est amené soit à choisir entre deux ou plusieurs chemins d'exécution (un choix entre deux ou plusieurs options), ou bien à répéter l'exécution d'un ensemble d'instructions, pour cela nous avons besoins de structures de contrôle pour contrôler et choisir les chemins d'exécutionsou refaire un traitement plusieurs fois. Les structures de contrôle sont de deux types : Structures decontrôles conditionnelles (Tests) et structures de contrôle répétitives (itératives, boucles).

III.6.3.1. Structures de contrôle conditionnelle

Ces structures sont utilisées pour décider de l'exécution d'un bloc d'instruction : est ce que ce bloc est exécuté ou non. Ou bien pour choisir entre l'exécution de deux blocs différents. Nous avons deux types de structures conditionnelles :

a) Test alternatif simple

Un test simple contient un seul bloc d'instructions. Selon une condition (expression logique), on décide est ce que le bloc d'instructions est exécuté ou non. Si la condition est vraie, on exécute le bloc, sinon on l'exécute pas.

La syntaxe d'un test alternatif simple est comme suit :

Tableau III.11 : La syntaxe d'un test alternatif simple

Algorithme	Langage C
si <condition> **alors** <bloc_instructions_si> **finsi**;	**if** <condition> **{** <bloc_instructions_si>; **}**

Exemples :

Tableau III.12 : Exemples d'un test alternatif simple

Algorithme	Langage C

lire(x)
si x > 2 **alors**
x ← x + 3 ;
finsi
écrire (x)
	scanf("x",&x);
	if (x > 2)
	{
	x= x + 3;
	}
	printf("%d", x);

❖ **Remarque :** Dans le langage C, un bloc est délimité par les deux accolades **{** et **}**.

Si le bloc contient une seule instruction, les accolades **{** et **}** sont facultatifs (on peut les enlever).

b) Test alternatif double

Un test double contient deux blocs d'instructions : on est amené à décider entre le premier bloc ou le seconds. Cette décision est réalisée selon une condition (expression logique ou booléenne) qui peut être vraie ou fausse. Si la condition est vraie on exécute le premier bloc, sinon on exécute le second.

La syntaxe d'un test alternatif double est :

Tableau III.13 : La syntaxe d'un test alternatif double

Algorithme	Langage C
si <condition> **alors** <bloc_instructions_si> ; **sinon** <bloc_instrucitons_sinon> ; **finsi**	**if** <condition> **{** <bloc_instructions_si>; **}** **else** **{** <bloc_instructions_sinon>; **}**

Exemple :

Tableau III.14 : Exemples d'un test alternatif double

Algorithme	Langage C

lire(x)
si (x > 2) **alors**
x ← x + 3
sinon
x ← x - 2
finsi
écrire (x)
	scanf("%d", x);
	if (x > 2)
	{
	x:= x + 3;
	}
	else
	{
	x:= x - 2;
	}
	printf("%d", x);

❖ **Remarques :**

- Dans le langage C, il faut jamais mettre de pointvirgule après la condition (erreur logique).
- Dans l'exemple précédent, on peut enlever **{ }** du **if** et ceux du **else** puisqu'il y a uneseule instruction dans les deux blocs.

Exemples :

Écrire un algorithme (et un programme C) qui permet d'indiquer si un nombre entier est pair ou non.

II.6.3.2. Structures de contrôle répétitives

Les structures répétitives nous permettent de répéter un traitement un nombre fini de fois. Par exemple, on veut afficher tous les nombre premier entre 1 et N (N nombre entier positif donné). Nous avons trois types de structures itératives (boucles) :

a) Boucle Pour (For)

La structure de contrôle répétitive pour (for en langage C) utilise un indice entier qui varie (avec un incrément = 1) d'une valeur initiale jusqu'à une valeur finale. À la fin de chaque itération, l'indice est incrémenté de 1 d'une manière automatique (implicite).

La syntaxe de la boucle **pour** est comme suit :

Tableau III.15 : La syntaxe de la boucle **pour**

Algorithme	Langage C
pour <indice> ← <vi> **à** <vf> **faire** <instruction(s)> **finPour**;	**for** (<indice>=<vi>; <indice><=<vf>; <indice>++) **{** <instruction(s)>; **}**

<indice> : variable entière (compteur de la boucle pour)

<vi> : valeur initiale <vf> : valeur finale

La boucle pour contient un bloc d'instructions (les instructions à répéter). Si le bloc contient une seule instruction, les accolades **{** et **}** sont facultatifs.

Le bloc sera répété un nombre de fois = (<vf> - <vi> + 1) si la valeur finale est supérieure ou égale à la valeur initiale. Le bloc sera exécuté pour <indice> = <vi>, pour <indice> = <vi>+1, pour <indice> = <vi>+2, …, pour <indice> = <vf>.

b) Boucle Tant-que (While)

La structure de contrôle répétitive **tantque** (**while** en langage C) utilise une expression logique ou booléenne comme condition d'accès à la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) donc on entre à la boucle, sinon on la quitte.

La syntaxe de la boucle **tantque** est comme suit :

Tableau III.16 : La syntaxe de la boucle **tantque**

Algorithme	Langage C
tant-que <condition> **faire** <instruction(s)> **fin tant-que**;	**while** (<condition>) **{** <instruction(s)>; **}**

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions tant que la condition est vraie. Une fois la condition est fausse, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après fin **Tant que** (après **}**).

❖ **Remarque :** Il est possible de remplacer toute boucle "**pour**" par une boucle "**tantque**", cependant, l'inverse n'est pas toujours réalisable.

c) Boucle Répéter (Repeat)

La structure de contrôle répétitive **répéter** utilise une expression logique ou booléenne comme condition de sortie de la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) on sort de la boucle, sinon on y accède (on répète l'exécution du bloc).

La syntaxe de la boucle **répéter** est comme suit :

Tableau III.17 : La syntaxe de la boucle **répéter**

Algorithme	Langage C
repéter <instruction(s)> **Jusqu'à** <condition>;	**do{** <instruction(s)>; **while** (<condition2>);

<condition> : expression logique qui peut être vraie ou fausse.

<condition2> : expression logique inverse de <condition>.

On exécute le bloc d'instructions jusqu'à avoir la condition correcte. Une fois la condition est vérifiée, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après **jusqu'à**. Dans la boucle **do …while** (en langage C) la condition est l'inverse de la condition de répéter.

La différence entre la boucle **répéter** et la boucle **tantque** est :

- La condition de **répéter** et toujours l'inverse de la condition **tantque** : pour **répéter** c'est la condition de sortie de la boucle, et pour **tantque** c'est la condition d'entrer.
- Le teste de la condition est à la fin de la boucle (la fin de l'itération) pour **répéter**. Par contre, il est au début de l'itération pour la boucle **tantque**. C'est-à-dire, dans **tantque** on teste la condition avant d'entrer à l'itération, et dans **répéter** on fait l'itération après on teste la condition.

III.7. Représentation en organigramme

Un organigramme est la représentation graphique de la résolution d'un problème. Il est similaire à un algorithme. Chaque type d'action dans l'algorithme possède une représentation dans l'organigramme.

Il est préférable d'utiliser la représentation algorithmique que la représentation par organigramme notamment lorsque le problème est complexe.

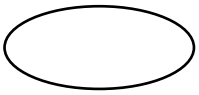
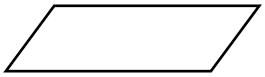
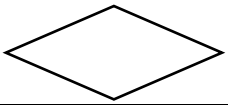
Les inconvénients qu'on peut rencontrer lors de l'utilisation des organigrammes sont :

- Quand l'organigramme est long et tient sur plus d'une page,
- problème de chevauchement des flèches,
- plus difficile à lire et à comprendre qu'un algorithme.

III.7. 1. Les symboles d'organigramme

Les symboles utilisés dans les organigrammes sont illustrés dans le tableau III.22

Tableau III.22 : Les symboles d'organigramme

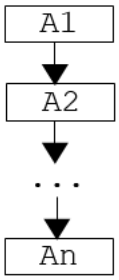
	Représente le début et la Fin de l'organigramme
	Entrées / Sorties : Lecture des données et écriture des résultats.
	Calculs, traitements
	Tests et décision : on écrit le test à l'intérieur du losange
	Ordre d'exécution des opérations (Enchaînement)
	Connecteur

III.8. Représentation des primitives algorithmiques

III.8.1. L'enchaînement

L'enchaînement permet d'exécuter une série d'actions dans l'ordre le leur apparition. Soit A_1, A_2, ...,A_n une série d'actions, leur enchaînement est représenté comme suit :

```
A1;
A2;
 .
 .
An;
```

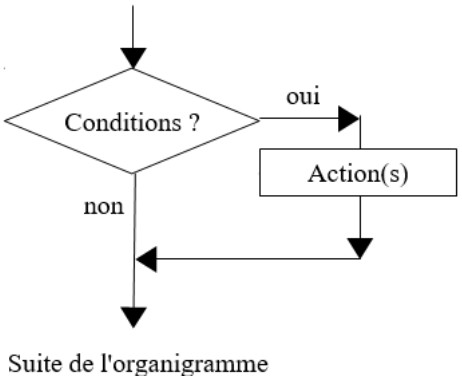


A1, A2, ..., An : peuvent être des actions élémentaires ou complexes.

III.8.2. La structure alternative simple

La syntaxe et l'organigramme de la structure alternative simple sont présentés dans le tableau III.23.

Tableau III.23 : La syntaxe et l'organigramme de la structure alternative simple

Représentation algorithmique	Représentation sous forme d'organigramme
si <condition> **alors** <action(s)>; **finsi**; Si la condition est vérifiée, le bloc <*action(s)*> sera exécuté, sinonrien, et on continue l'exécution de l'instruction après fin si.	

Les condition utilisées pour les teste (simple ou double) sont des expression logique ou booléennes, ça veut dire des expression dont leur évaluation donne soit TRUE (Vrai) ou FALSE (faux). Toute comparaison entre deux nombre représente un expression logique. On peut former des expressions logiques à partir d'autres expressions logique en utilisant les opérateurs suivant : Not, Or et And.

Exemple :

(x >= 5) : est une expression logique, elle est vrai si la valeur de x est supérieur ou égale à 5. elle est fausse dans le cas contraire.

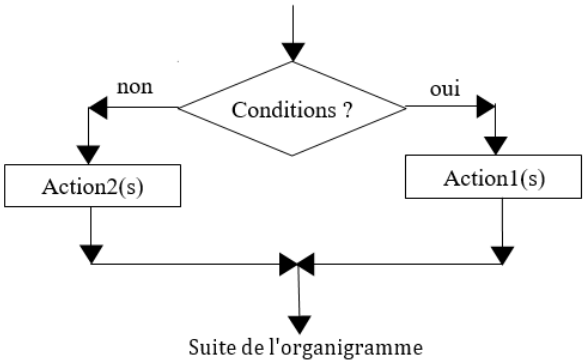
Not (x >= 5) : E.L. qui est vrai uniquement si la valeur de x est inférieur à 5.

(x >=5) And (y<=0) : E.L. qui est vrai si x est supérieur ou égale à 5 et y inférieur ou égale à 0.

III.8.3. La structure alternative double

La syntaxe et l'organigramme de la structure alternative double sont présentés dans le tableau II.24.

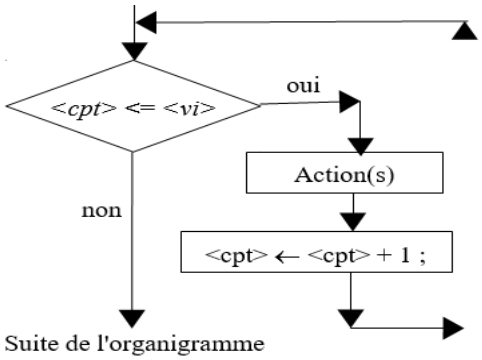
Tableau III.24 : La syntaxe et l'organigramme de la structure alternative double

Représentation algorithmique	Représentation sous forme d'organigramme
si <condition> **alors** <action1(s)>; **sinon** <action2(s)>; **finsi**; Si la condition est vérifiée, le bloc <*action1(s)*> sera exécuté, sinon (si elle est fausse) on exécute <*action2(s)*>.	

III.8.4. La structure itérative POUR (Boucle POUR)

La syntaxe et l'organigramme de la structure itérative POUR sont présentés dans le tableau II.25.

Tableau III.25 : La syntaxe et l'organigramme de la structure itérative POUR

Représentation algorithmique	Représentation sous forme d'organigramme
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour**;	

Dans la boucle **POUR**, on exécute le bloc <*acitons*> *(<vf> - <vi> + 1)* fois. Ceci dans le cas où <*vf*> est supérieur ou égale à <*vi*>. Dans le cas contraire, le bloc d'actions ne sera jamais exécuté. Le déroulement de la boucle POUR est exprimé comme suit :

1. La variable entière <*cpt*> (le compteur) prends la valeur initiale <*vi*> ;

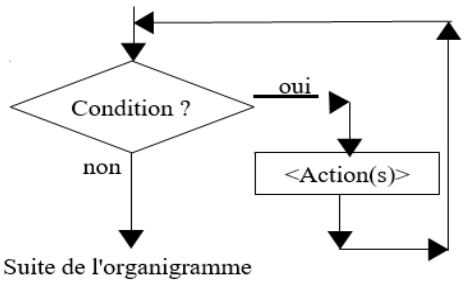
2. on compare la valeur de <*cpt*> à celle de <*vf*> ; si <*cpt*> est supérieur à <*vf*> on sort de laboucle ;
3. Si <*cpt*> est inférieur ou égale à <*vf*> on exécute le bloc <*action(s)*> ;
4. La boucle *POUR* incrémente automatiquement le compteur <*cpt*>, c'est-à-dire elle lui ajoute *un* (<cpt> ← <cpt> + 1);
5. On revient à 2 (pour refaire le teste <*cpt*> <= <*vi*> C'est pour cela qu'on dit la boucle);

❖ **Remarque :** La boucle **POUR** est souvent utilisée pour les structures de données itératives (les tableaux et les matrices – variables indicées).

III.8.5. La structure itérative Tant-que (Boucle Tant-que)

La syntaxe et l'organigramme de la structure itérative Tant-que sont présentés dans le tableau III.26.

Tableau III.26 : La syntaxe et l'organigramme de la structure itérative Tant-que

Représentation algorithmique	Représentation sous forme d'organigramme
Tant-que <condition> **faire** <action(s)>; **fin Tant-que**;	

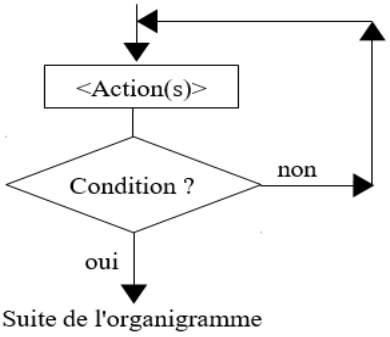
On exécute le bloc d'instructions <actions> tant que la <*condition*> est vérifiée (c'est-à-dire elle est vraie). Le déroulement de la boucle est comme suit :

1. On évalue la condition : si la condition est fausse on sort de la boucle ;
2. Si la condition est vraie, on exécute le bloc <actions> ; sinon va à **4**.
3. On revient à 1 ;
4. On continue la suite de l'algorithme

III.8.6. La structure itérative Répéter (Boucle Répéter)

La syntaxe et l'organigramme de la structure itérative répéter sont présentés dans le tableau III.27.

Tableau III.27 : La syntaxe et l'organigramme de la structure itérative répéter

Représentation algorithmique	Représentation sous forme d'organigramme
Répéter <action(s)>; **Jusqu'à** <condition>;	

On répète l'exécution du bloc <*action(s)*> jusqu'à avoir la condition correcte. Le déroulement est comme suit :

1. On exécute le bloc <*action(s)*> ;
2. On évalue la condition : si la condition est vérifiée (elle est vraie) on sort de la boucle (on continue la suite de l'algorithme);
3. Si la condition n'est pas vérifiée (elle est fausse) on revient à 1.

❖ **Remarques :**

- ✓ N'importe quelle boucle ***POUR*** peut être remplacée par une boucle ***Tant-que***, cependant l'inverse n'est pas toujours correcte, c'est-à-dire, il y a des cas où la boucle ***Tant-que*** ne peut pas être remplacée par une boucle ***POUR***.

- ✓ On transforme une boucle ***POUR*** à une boucle ***Tant-que*** comme suit :

Tableau III.28 : Transformation de la boucle POUR à la boucle Tant-que

Boucle **POUR**	Boucle **Tant-Que**
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour**;	<cpt> ← <vi>; **Tant-que** <cpt> <= <vf> **faire** <action(s)>; <cpt> ← <cpt> + 1; **fin Tant-que**;

- ✓ La boucle Répéter possède une condition de sortie (c'est-à-dire si elle est vraie on sort de la boucle), alors que la boucle Tant-que possède une condition d'entrée (c'est-à-dire si elle est vraie onentre dans la boucle).
- ✓ La boucle Répéter exécute le bloc <*action(s)*> au moins une fois, le teste vient après

l'exécution du bloc.

- ✓ La boucle Tant-que peut ne pas exécuter le bloc <action(s)> (dans le cas où la condition est fausse dès le début), puisque le teste est avant l'exécution du bloc.

III.9. Exercices corrigés

Exercice N°01 : *Calcul du périmètre d'un cercle*

Écrire un programme en langage C qui demande à l'utilisateur de saisir le rayon d'un cercle, puis qui calcule et affiche le périmètre de ce cercle.

Rappel : Le périmètre **P** d'un cercle de rayon **R** est donné par la formule : **P = 2 * π * R**, où π est la constante mathématique pi (approximativement 3.14159).

Algorithme	**Programme C**

Algorithme Exo1;
Constante
pi=3.14 ;
Variables
R, P : **réels**;
 	
Début
//Les entrées
Écrire(" Donner le rayon R : ") ;
Lire(R) ;
 	
//Traitement
P ← 2*pi*R ;
 	
//Les sorties
Écrire("Le périmètre = ", P) ;
Fin.
	#include <stdio.h>
	
	int main()
	{
	const float pi=3.14;
	float R,P;
	
	//Les entrées
	printf("Donner le rayon R : \n");
	scanf("%f",&R);
	
	//Traitement
	P=2*pi*R;
	
	//Les sorties
	printf("Le périmètre = %.3f", P);
	
	return 0;
	}

Exercice N°02 : *Identificateurs*

Identifier les identificateurs valides et non valides : 1A ; R? ; K2 ; T280 ; 12R ; Hauteur ; Prix-HT ; Prix_HT ; Exo 04 ; Exo_04 ; Exo-04 ; Program ; read.

Corrigé de l'exercice N°02 :

Les variables valides et non valides :

Variable valide	Variable non valide
K2 T280 Hauteur Prix_HT Exo_04	1A R? 12R Prix-HT Exo 04 Exo-04 Program read

Exercice N°03 : *(Enoncé du problème ➔ Algorithme ➔ Programme)*

Écrire un algorithme, puis traduit le en programme PASCAL, pour chacun des problèmes suivants :

1) Permuter entre les deux variables X et Y ?
2) Calculer le quotient de la division euclidienne de a par b ?

Corrigé de l'exercice N°03 :

1) Permuter entre les deux variables X et Y ?

Algorithme	Programme C

Algorithme Exo3_1;
Variables
x, y, t : **entier**;
 	
Début
//Les entrées
Lire(x, y) ;
 	
//Traitement
t ← x;
x ← y;
y ← t ;
 	
//Les sorties
Écrire('x=', x, 'y=', y) ;
Fin.
	#include <stdio.h>
	
	int main()
	{
	int x, y, t;
	
	//Les entrées
	scanf("%d %d",&x, &y);
	
	//Traitement
	t=x;
	x=y ;
	y=t ;
	
	//Les sorties
	printf("x=%d , y=%d", x, y);
	
	return 0;
	}

2) Calculer le quotient de la division euclidienne de a par b ?

Algorithme	**Programme C**

Algorithme Exo3_2;
Variables
a, b, Q : **entier**;
 	
Début
//Les entrées
Lire(a, b) ;
 	
//Traitement
Q ← a div b;
 	
//Les sorties
Écrire("Le quotient est : ", Q) ;
Fin.
	#include <stdio.h>
	
	int main()
	{
	int a, b, Q;
	
	//Les entrées
	scanf("%d %d", &a, &b);
	
	//Traitement
	Q= a % b ;
	
	//Les sorties
	printf("Le quotient est : %d ", Q) ;
	
	return 0;
	}

Exercice N°04 :

Ecrire un programme Pascal intitulé **PARITE** qui saisit un nombre entier et détecte si ce nombre est pair ou impair.

Corrigé de l'exercice N°04 :

Algorithme	**Programme C**

Algorithme Partie ;
Variables
N, R : **entier**;
 	
Début
 	
//Les entrées
Écrire ("Donner un entier : ") ;
Lire (N) ;
 	
//Traitement
R ← N mod 2;
 	
//Les sorties
Si (R = 0) **alors**
Écrire (N, " est pair")
Sinon
Écrire (N, " est impair") ;
	#include <stdio.h>
	
	int main()
	{
	int N, R;
	
	//Les entrées
	printf("Donner un entier : ");
	scanf("%d", &N);
	
	//Traitement
	R= N % 2 ;
	
	//Les sorties
	if(R==0)
	printf("%d est pair", N) ;
	else
	printf("%d est impair", N) ;

Fin-Si ; **Fin.**	**return 0;** **}**

Exercice N°05 :

Ecrire un algorithme/programme en langage C pour chaque cas suivant :

1) Calculer la somme $S = 1^2 + 3^2 + 5^2 + \cdots + (2N + 1)^2$
2) Calculer le produit $P = 1 * 2 * 3 * ... * N$

Corrigé de l'exercice N°05 :

1) Calculer la somme $S = 1^2 + 3^2 + 5^2 + \cdots + (2 * N + 1)^2$

Algorithme	Programme C

Algorithme exo2_1;
variables
S, N, i: **entier**;
Début
//Les entrées
Ecrire("Donner la valeur de N : ");
Lire(N);
//Traitement
S ← 0;
Pour i ← 0 **à** N **faire**
S ← S + sqr(2*i+1);
Fin-pour
//Les sorties
Ecrire ("S =", S);
Fin.
	#include <stdio.h>
	int main()
	{
	int a, b, Q;
	//Les entrées
	scanf("%d %d", &a, &b);
	//Traitement
	Q= a % b ;
	//Les sorties
	printf("Le quotient est : %d ", Q) ;
	return 0;
	}

2) Calculer le produit $P = 1 * 2 * 3 * ... * N$

Algorithme	Programme C
Algorithme exo2_2; **Variables** N, i, P: **entier** ; **Début** //Les entrées	**#include <stdio.h>** **int main()** **{** **int** a, b, Q; //Les entrées

```
Ecrire("Donner la valeur de N : ");
Lire(N);
//Traitement
P ← 1;
Pour i ← 1 à N faire
    P ← P* i;
Fin-pour;
//Les sorties
Ecrire("P =", P);
Fin.
```

```
    scanf("%d %d", &a, &b);

    //Traitement
    Q= a % b ;

    //Les sorties
    printf("Le quotient est : %d ", Q) ;

    return 0;
}
```

II.10. Exercices supplémentaires

Exercice 01 :

Réaliser les conversions suivantes :

$2021 = (?)_2$

$(10110110001)_2 = (?)_{10}$

$(753)_8 = (?)_{10}$

$(753)_8 = (?)_2$

$(101110011100011)_2 = (?)8 = (?)_{16}$

$(AB0793)_{16} = (?)_8$

Exercice 02 :

En PASCAL, indiquer, parmi cette liste de mots, les identificateurs valides et non-valides :

12K, a, x1, k12, prix unitaire, qte-stock, sinon, while, begin, hateur, largeur

Exercice 03 :

Ecrire un algorithme puis la traduction en langage C d'un programme **Surface_Rectangle**, qui calcule la surface d'un rectangle de dimensions données et affiche le résultat sous la forme suivante : "La surface du rectangle dont la longueur mesure .... m et la largeur mesure .... m, a une surface égale à .... mètres carrés".

Exercice 04 :

Ecrire un algorithme puis la traduction en langage C d'un programme **Trapèze**, qui lit les dimensions d'un trapèze et affiche sa surface.

Exercice 05 :

Ecrire un algorithme puis la traduction en langage C d'un programme qui lit une **température** en degrés Celsius et affiche son équivalent en Fahrenheit.

Exercice 06 :

Exécuter les séquences d'instructions suivantes manuellement et donner les valeurs finales des variables A, B, C et celles de X, Y, Z.

a) A← 5; B← 3; C← B+A; A← 2; B← B+4; C← B-2

b) X← -5; Y← 2*X; X← X+1; Y← sqr(-X-Y); Z ← sqr(-X+Y); X← -(X+3*Y)+2

Ecrire les algorithmes correspondants puis les programmes en langage C correspondants et les exécuter.

Exercice 07 :

Ecrire un algorithme permettant d'effectuer une permutation circulaire de trois nombre entiers a, b et c.

Exemple : a=10, b=20 et c=30

Après permutation : a=30, b=10 et c=20

Exercice 8 :

Ecrire un programme en langage C permettant de lire la valeur de la température de l'eau et d'afficher son état :

« Glace» Si la température <=0,

« Liquide» Si 0 < la température <100,

« Vapeur» Si la température >=100.

Exercice 9 :

Ecrire un programme en langage C qui permet de résoudre l'équation du second degré $ax^2 + bx + c = 0$

Exercice 10 :

On demande d'écrire l'algorithme d'une fiche de paie journalière d'un ouvrier rémunéré à la tâche. Pour cela, on donne :

- La valeur de cette rémunération par pièces réalisées VP,
- Le salaire brut (SB) est calculé selon le nombre de pièces correctes réalisées pendant la journée (NPC) comme suit :

  Si NPC ≤ 100, l'ouvrier touche NPC*VP

  Si NPC > 100, l'ouvrier touche 150*VP
- On enlève à la fin 10% du salaire pour les charges sociales (CS).

Calculer et afficher le salaire journalier brut (SB), les charges sociales (CS) et salaire journalier net (SN).

NB : Salaire brut=salaire totale ; Salaire net=salaire sans les charges sociales.

Exercice 11 :

Ecrire un programme en langage C qui calcule et affiche la **somme** et le **produit**, des 20 premiers entiers (de 1 à 20).

Exercice 12 :

Ecrire un programme en langage C faisant calculer et afficher le **factoriel** d'un entier naturel N donné. Sachant que (pour N>0) : N ! = N x (N-1) x (N-2) x ..... 3 x 2 x 1.

Exercice 13 :

Ecrire un algorithme permettant de calculer la somme de tous les nombres impairs entre deux valeurs N et M.

Exercice 14 :

Ecrire un algorithme permettant de tester si un nombre N est premier en utilisant une boucle **Tant que** puis une boucle **Répéter**.

Exercice 15 :

Ecrire un programme en langage C qui permet de calculer la somme S suivante : (avec X un nombre réel donné)

$$S = \frac{X^2}{2} - \frac{X^4}{4} + \frac{X^6}{6} - \frac{X^8}{8} + \frac{X^{10}}{10} - \frac{X^{12}}{12} + \cdots$$

Exercice 16 :

Ecrire un programme en langage C qui permet d'afficher tous les multiples de 7 positifs strictement et inférieur à 100. Le programme doit aussi calculer et afficher la somme, le produit et la moyenne de ces multiples de 7.

On veut aussi avoir sur l'écran à l'exécution du programme, les affichages sous la forme suivante :

```
Les multiples de 7 inférieurs à 100 sont :
7 14 21 ……
La somme de ces nombres est : ……
Le produit de ces nombres est : …...
La moyenne de ces nombres est : …..
```

Bibliographie

Bibliographie

[1] https://www.techno-science.net/glossaire-definition/Algebre-de-Boole-logique.html

[2] Daniel Etiemble, 'ALGÈBRE DE BOOLE ET FONCTIONS BOOLÉENNES'. Notes de cours. (https://www.lri.fr/)

[3] Farid Tafinine, 'Electronique Numérique', Cours de l'université A.MIRA de Bejaia, département de Génie électrique. Année universitaire : 2016-2017.

[4] Redouane Ouzeggane. Cours informatique 1, Université de Béjaia.