Ecole Marocaine des Sciences d'Ingénieurs

Informatique 1

Version 1.0

Pr. Y. OUASSIT

Date : 15 octobre 2025

Table des matières

Première partie

L'algorithmique

Chapitre 1

Introduction à l'algorithmique

1.1 Introduction

L'informatique, en tant que discipline moderne, repose fondamentalement sur la capacité à résoudre des problèmes à l'aide de méthodes systématiques et répétables. Ces méthodes sont souvent représentées sous la forme d'algorithmes. La notion d'algorithme et l'étude qui en découle, appelée algorithmique, occupent une place centrale dans l'évolution des sciences informatiques, influençant autant le développement de logiciels que la résolution de problèmes complexes dans divers domaines tels que les mathématiques, la physique, l'économie et la biologie. Dans cette première partie du cours, nous allons définir ces concepts essentiels, examiner leur importance, et poser les bases pour comprendre les principes de conception et d'analyse des algorithmes.

1.1.1 Notion d'algorithme

Un algorithme peut être défini comme une suite finie et ordonnée d'instructions ou d'étapes permettant de résoudre un problème ou de réaliser une tâche donnée. Il peut être représenté sous forme de texte, de diagrammes (comme les organigrammes) ou dans des langages de programmation. Le concept d'algorithme trouve son origine dans les travaux du mathématicien perse Al-Khwarizmi au IXe siècle, et est aujourd'hui au cœur de l'informatique moderne.

Plus précisément, un algorithme est une séquence d'instructions qui prend un ou plusieurs entrées (données d'entrée), effectue un ensemble d'opérations définies, et produit une ou plusieurs sorties (résultats).

Un algorithme est une notion attaché à notre vie quotidienne, nous utilisons chaque jour des algorithmes :

1. Une recette de cuisine est un algorithme !
2. Le mode d'emploi d'un magnétoscope est aussi un algorithme !
3. Indiqué un chemin à un touriste égaré ou faire chercher un objet à quelqu'un par téléphone c'est fabriquer - et faire exécuter - des algorithmes.

Une recette de cuisine, par exemple, est un algorithme : à partir des ingrédients, elle explique comment parvenir au plat. De même, un itinéraire routier explique comment, à partir d'une position initiale, rejoindre une position finale en un certain nombre d'étapes.

Un algorithme doit respecter certaines propriétés fondamentales :

1. **Finitude :** Un algorithme doit avoir un nombre fini d'étapes, et il doit se terminer après un temps raisonnable.
2. **Précision :** Chaque étape de l'algorithme doit être définie avec précision et ne doit pas prêter à confusion.
3. **Entrées et sorties :** Un algorithme doit accepter des entrées bien définies et produire des résultats.
4. **Efficacité :** Un bon algorithme doit être optimal, c'est-à-dire qu'il doit résoudre le problème avec un minimum de ressources (temps, espace mémoire, etc.).
5. **Déterminisme :** À partir des mêmes entrées, un algorithme doit toujours produire les mêmes sorties.

Les ordinateurs eux-mêmes ne sont fondamentalement capables d'exécuter que quatre opérations logiques :

1. l'affectation de variables
2. la lecture / écriture
3. les tests
4. les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques-unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes.

1.2 L'algorithmique

L'algorithmique est la branche de l'informatique qui étudie les algorithmes, leurs conceptions, leurs caractéristiques, et leurs performances. C'est une discipline qui combine des aspects théoriques et pratiques et s'intéresse aussi bien à la manière de concevoir un algorithme qu'à la manière d'analyser son efficacité.

L'algorithmique aborde plusieurs aspects essentiels, dont la conception et l'analyse des algorithmes. La conception d'algorithmes se concentre sur la manière de les construire pour résoudre un problème spécifique tout en suivant des principes qui garantissent leur bon fonctionnement et leur efficacité. L'analyse des algorithmes, quant à elle, vise à évaluer leur performance en termes de temps de calcul et d'espace mémoire utilisés, en examinant à la fois leur complexité temporelle, c'est-à-dire le nombre d'opérations nécessaires, et leur complexité spatiale, c'est-à-dire la quantité de mémoire requise.

1.2.1 La démarche de programmation et analyse descendante

La résolution d'un problème passe par toute une suite d'étapes :

— Phase d'analyse et de réflexion (algorithmique)

— Phase de programmation

- choisir un langage de programmation
- traduction de l'algorithme en programme

- programme (ou code) source
- compilation : traduction du code source en code objet
- traduction du code objet en code machine exécutable, compréhensible par l'ordinateur

— Phase de test

— Phase d'exécution

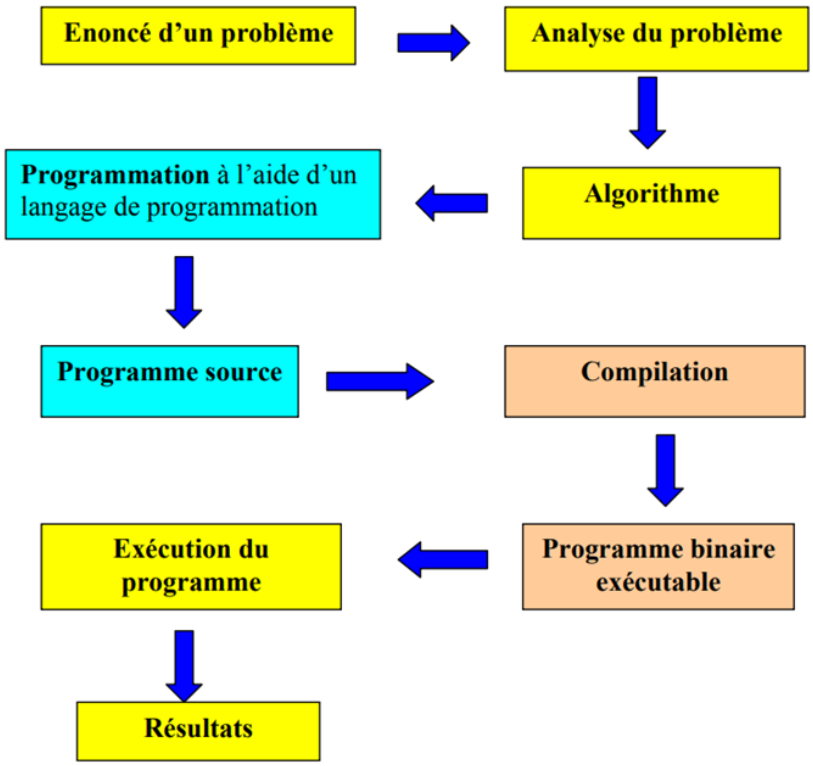

FIGURE 1.1 – Démarche de programmation et analyse descendante

Le travail est ici surtout basé sur l'analyse du problème et l'écriture de l'algorithme. La réalisation d'un programme passe par l'analyse descendante du problème : il faut réussir à trouver les actions élémentaires qui, en partant d'un environnement initial, nous conduisent à l'état final.

L'analyse descendante consiste à décomposer le problème donné en sous-problèmes, et ainsi de suite, jusqu'à descendre au niveau des primitives. Les étapes successives donnent lieu à des sous- algorithmes qui peuvent être considérés comme les primitives de machine intermédiaires.

Le travail de l'analyse est terminé lorsqu'on a obtenu un algorithme ne comportant que :

— Des primitives de la machine initiale,

— Des algorithmes déjà connus.

L'ordre des instructions est essentiel : la machine ne peut exécuter qu'une action à la fois et dans l'ordre donné ; c'est la propriété de séquentialité. Une fois ces actions déterminées, il suffit de les traduire dans le langage de programmation.

1.2.2 Étape d'analyse

L'étape d'analyse dans la démarche de conception des algorithmes consiste à bien comprendre le problème à résoudre avant de concevoir une solution. C'est une phase cruciale qui permet de s'assurer que l'algorithme sera efficace et adapté au problème.

Voici les principaux points de cette étape :

1. Comprendre le problème : Identifier les données d'entrée, les résultats attendus et les contraintes éventuelles. Il s'agit de bien définir ce que l'on cherche à résoudre.
2. Formuler le problème : Le problème doit être décrit de manière claire et précise, souvent sous forme d'énoncé ou de modélisation mathématique.
3. Choisir une méthode de résolution : Décider de l'approche à utiliser (itérative, récursive, diviser pour régner, etc.) pour répondre aux exigences du problème.
4. Vérifier la faisabilité : S'assurer que le problème peut être résolu avec les moyens disponibles (temps, ressources, matériel).

L'analyse permet de poser une base solide avant de passer à la conception proprement dite de l'algorithme. Elle garantit que toutes les spécificités du problème sont prises en compte pour éviter des erreurs ou des solutions inefficaces.

Exemple 1 :

1. **Énoncé du problème :**
   Nous devons calculer le montant total TTC (Toutes Taxes Comprises) d'un lot d'ordinateurs (PC) à partir du prix unitaire HT (Hors Taxes) de chaque PC, de la quantité achetée, et du taux de TVA applicable.
2. **Analyse du problème :**

   Données d'entrée : Le prix unitaire HT d'un PC. La quantité de PC achetée. Le taux de TVA (en pourcentage).

   Résultat attendu (sortie) : Le montant TTC total du lot de PC.

   Formulation du problème (Traitement) : Le montant total TTC est calculé à partir de la formule suivante :
   Montant TTC = (Prix unitaire HT × Quantité) × (1+TVA/100)

Cet exemple montre comment l'analyse permet de bien structurer le problème avant de passer à la conception de l'algorithme. Afin de simplifier l'expression des opérations mathématiques, nous allons associer à chaque donnée un identificateur, il s'agit d'un nom symbolique qui représentera les données. Nous allons aussi spécifier le type de chaque données (domaine de valeurs) et la catégorie.

- Les données d'entré :

Données	Identificateur	Type	Catégorie
Le prix unitaire hors taxe	PUHT	Réel	Variable
Le nombre de PC	N	Entier	Variable
Le taux de TVA	TVA	Réel	Constante

- Les données de sortie :

Données	Identificateur	Type	Catégorie
Montant TTC	MTTC	Réel	Variable

- Traitement :

MHT = PUHT x N

MTVA = MHT x TVA

MTTC = MHT+ MTVA

1.2.3 Pseudo langage

Une fois l'étape d'analyse est effectuée, nous allons passer à la conception de notre algorithme. Pour cet étape nous allons adopter une structure pour l'écriture d'un algorithme, cette structure ne représente pas un standard mais son objectif c'est d'avoir une meilleure organisation des composants de l'algorithme. La structure générale d'un algorithme se compose de trois parties :

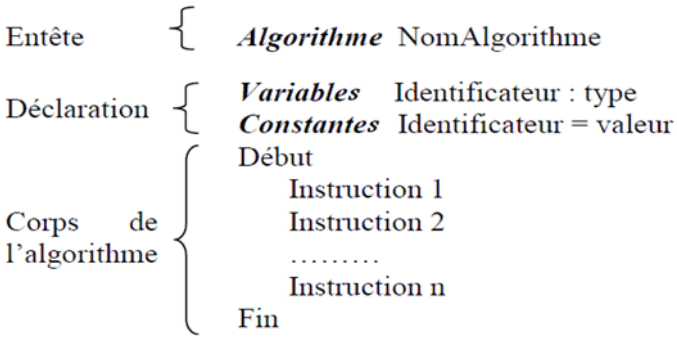

FIGURE 1.2 – Structure générale d'un algorithme

La partie entête :
Cette partie commence par le mot Algorithme et le nom de l'algorithme, ce dernier doit donner une idée sur l'objet de l'algorithme. Exemple : Pour un algorithme qui fait le calcul de la moyen des notes d'une classe, on peut lui donner comme nom (Moyen ; MoyenNote ; Moyen_Note ; ...).

La partie déclaration :
Dans cette partie on doit déclarer l'ensemble des données dont on aura besoin dans l'algorithme

La partie Corps :
Contient l'ensemble des instructions (ordres) à exécuter pour pouvoir résoudre le problème.

Nous allons continuer la conception de l'**Exemple 1** ci-dessous, et l'écrire en pseudo code :

Algorithme 1 Montant TTC
Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : TVA = 0.2
Début
$MHT = PUHT \times N$
$MTVA = MHT \times TVA$
$MTTC = MHT + MTVA$
Fin

L'algorithme ci-dessus 1 est une première version, que nous allons améliorer et compléter dans les sections suivantes.

1.3 Variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier). Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, etc. Dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une variable.

1.3.1 Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la déclaration des variables. Une variable ne peut être utilisée que s'elle est déclarée. La déclaration se fait par la donnée du nom de la variable et du type de la variable.

Syntaxe :

NomVariable1, NomVariable2 : TypeVariable

1.3.2 Noms de variables

Le nom de la variable (l'étiquette de la boîte) obéit à des règles qui changent selon le langage utiliser. Les principales règles à respecter sont :

- Le nom de variable peut comporter des lettres et des chiffres,
- On exclut la plupart des signes de ponctuation, en particulier les espaces.
- Un nom de variable doit commencer par une lettre.
- Le nombre maximal de caractères qui composent le nom d'une variable dépend du langage utilisé.

— Ne pas utiliser les mots clés du langage de programmation.

1.3.3 Types de variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires. Pour pouvoir accéder aux données, le programme (quel que soit le langage dans lequel il est écrit) fait usage d'un grand nombre de variables de différents types. Une VARIABLE possède :

— Un nom : Une variable apparaît en programmation sous un nom de variable.

— Un type : Pour distinguer les uns des autres les divers contenus possibles, on utilise différents types de variables (entiers, réels, chaîne de caractères ...).

— Une valeur : Une fois la variable définie, une valeur lui est assignée, ou affectée.

— Une adresse : C'est l'emplacement dans la mémoire de l'ordinateur, où est stockée la valeur de la variable.

Les Types simples (entier, flottant, caractère) :

— Type Entier : pour manipuler les nombres entiers positifs ou négatifs. Par exemple : 5, -15, etc.

— Type Réel : pour manipuler les nombres à virgule. Par exemple : 3.14, -15.5, etc.

— Type Chaîne : pour manipuler des chaînes de caractères permettant de représenter des mots ou des phrases. Par exemple : "bonjour, Monsieur"

— Type Booléen : pour les expressions logiques. Il n'y a que deux valeurs booléennes : vrai et faux.

A un type donné, correspond un ensemble d'opérations définies pour ce type :

Type	Opérations possibles	Symbole ou mot clé correspondant
Entier	Addition	+
	Soustraction	-
	Multiplication	*
	Division	/
	Division Entière	DIV
	Modulo	MOD
	Exposant (puissance)	^
	Comparaisons	<, =, >, <=, >=, !=
Réel	Addition	+
	Soustraction	-
	Multiplication	*
	Division	/
	Exposant	^
	Comparaisons	<, =, >, <=, >=, !=
Chaîne	Concaténation	<, =, >, <=, >=, !=, +
Booléen	Comparaison	<, =, >, <=, >=, !=
	Logiques	ET, OU, NON

FIGURE 1.3 – Opérations définies pour chaque type

1.4 Les instructions de base (Les primitives)

1.4.1 L'affectation

L'affectation est l'action élémentaire dont l'effet est de donner une valeur à une variable (ranger une valeur à une place). L'affectation est réalisée au moyen de l'opérateur ←. Elle signifie " prendre la valeur se trouvant du côté droit (souvent appelée rvalue) et la copier du côté gauche (souvent appelée lvalue) ". Une rvalue représente toute constante, variable ou expression capable de produire une valeur, mais une lvalue doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat).

Exemple : X ← 3

Signifie mettre la valeur 3 dans la case identifiée par X. A l'exécution de cette instruction, la valeur 3 est rangée en X (nom de la variable).
La valeur correspond au contenu : 3
La variable correspond au contenant : X

On peut représenter la variable X par une boite ou case, et quand elle prend la valeur 3, la valeur est dans la case X :

FIGURE 1.4 – Affectation

On remarque qu'une variable ne peut contenir à un instant donné qu'une seule valeur. La dernière instruction (Y ← X) signifie : copier dans Y la valeur actuelle de X.

Exercice 1: Un petit exercice instructif
Quelles sont les valeurs successives prises par les variables X et Y suit aux instructions suivantes :
X ← 1 ;
Y ← -4 ;
X ← X+3 ;
X ← Y-5 ;
Y ← X+2 ;
Y ← Y-6 ;

Solution :

X	1	1	4	-9	-9	-9
Y		-4	-4	-4	-7	-13

Exercice 2
Soit 2 variables quelconques (nombres ou caractères) x et y ayant respectivement comme valeur a et b ; quelles sont les affectations qui donneront à x la valeur b et à y la valeur a ?

Solution :
Analyse : la première idée est d'écrire : x ← y ; y ← x. Mais ça ne marche pas, les deux variables se retrouvent avec la même valeur b ! Il faut mettre la valeur de x de coté pour ne pas la perdre : on utilise une variable auxiliaire z et on écrit les instructions suivantes :
z ← x ;
x ← y ;
y ← z ;

Revenons à l'algorithme de l'**Exemple 1**. Nous allons le corriger, car dans le code, nous avons utilisé le symbole "=" dans les instructions telles que MHT = PUHT x N. Cette expression était censée demander à la machine de calculer PUHT x N et de stocker le résultat dans MHT. Cependant, comme nous l'avons vu, le symbole "=" est utilisé pour tester l'égalité entre deux expressions. Ainsi, nous devons le remplacer par le symbole d'affectation approprié. Voici la version corrigée de l'algorithme :

Algorithme 2 Montant TTC
Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : $TVA \leftarrow 0.2$
Début
$MHT \leftarrow PUHT \times N$
$MTVA \leftarrow MHT \times TVA$
$MTTC \leftarrow MHT + MTVA$
Fin

1.4.2 Les instructions de lecture / affichage

Les instructions de lecture et d'écriture sont des éléments fondamentaux en algorithmique, car elles permettent de créer une interaction entre la machine et l'utilisateur, ainsi que de manipuler et gérer les données. Comprendre ces instructions est crucial pour développer des algorithmes fonctionnels et efficaces.

Pour illustrer l'importance de ces instructions nous allons essayer de critiquer et améliorer l'algorithme suivant :

Algorithme 3 Carré
Variables : X,Y : Réel
Début
$X \leftarrow 12$
$Y \leftarrow X \hat{} 2$
Fin

1.4.2.1 L'instruction de lecture

L'algorithme présenté ci-dessus 3) calcule actuellement le carré du nombre 12. Bien que l'algorithme soit syntaxiquement correct, il manque d'interactivité. En effet, l'algorithme est figé pour un nombre spécifique (12) et, par conséquent, chaque exécution ne produit que le carré de ce nombre. Pour calculer le carré d'un autre nombre, comme 20 par exemple, il est nécessaire de modifier l'algorithme en affectant la nouvelle valeur à la variable X et en relançant l'exécution.

Pour rendre cet algorithme plus flexible et interactif, nous allons ajouter une instruction permettant de récupérer la valeur de la variable X directement auprès de l'utilisateur à chaque exécution de l'algorithme. Ainsi, l'algorithme pourra être utilisé pour calculer le carré de n'importe quelle valeur sans nécessiter de modifications préalables. Pour ce faire, nous intégrerons une instruction de saisie utilisateur.

En algorithmique, l'instruction d'entrée est utilisée pour obtenir des données ou des valeurs de l'utilisateur ou d'une autre source externe. Cette instruction est cruciale pour rendre un algorithme interactif et dynamique.

L'instruction d'entrée ou de lecture permet à l'utilisateur de saisir des données au clavier pour qu'elles soient utilisées par l'algorithme. Cette instruction assigne (affecte) une valeur entrée au clavier dans une variable.

Syntaxe : *Lire(identificateur)*

Exemple : *Lire(A)*

Lire (A) : Cette instruction permet à l'utilisateur de saisir une valeur au clavier qui sera affectée à la variable A.

Remarque:. *Lorsque le programme rencontre cette instruction, l'exécution s'interrompt et attend que l'utilisateur tape une valeur. Cette valeur est rangée en mémoire dans la variable désignée.*

Voici la nouvelle amélioration de l'algorithme du calcule du carré :

Algorithme 4 Carré V2
Variables : X,Y : Réel
Début
Lire(X)
$Y \leftarrow X \hat{} 2$
Fin

Remarque:. *Généralement il faut prévoir à demander à l'utilisateur de saisir toutes les données d'entré du problème.*

Un exemple d'algorithme qui calcule la somme de deux nombres :

Algorithme 5 Somme

Variables : A,B,S : Réel
Début
Lire(A)
Lire(B)
$S \leftarrow A + B$
Fin

Dans l'algorithme ci-dessus 5 nous avons deux données d'entré (les variables A et B), nous avons donc fait appel à l'instruction de lecture Lire(A) et Lire(B).

Voici une nouvelle amélioration de l'algorithme de l'**Exemple 1**, où nous allons récupérer les valeurs des données d'entré auprès de l'utilisateur :

Algorithme 6 Montant TTC

Variables : PUHT, MTTC, MHT, MTVA : Réel
N : Entier
Constantes : $TVA \leftarrow 0.2$
Début
Lire(PUHT)
Lire(N)
$MHT \leftarrow PUHT \times N$
$MTVA \leftarrow MHT \times TVA$
$MTTC \leftarrow MHT + MTVA$
Fin

1.4.2.2 L'instruction de sortie/écriture/affichage

instructions de sortie (ou d'affichage) sont utilisées pour afficher des informations à l'utilisateur via le périphérique standard de sortie (L'écran). Elles permettent de communiquer des résultats, des messages ou des informations sur l'état de l'algorithme.

L'algorithme peut utiliser des instructions d'affichage pour afficher des résultats de calculs, des messages d'information, ou des erreurs. Par exemple, après avoir calculé la carré d'un nombre, l'algorithme peut afficher le résultat à l'utilisateur.

Syntaxe : *Ecrire(expressions)*

L'expression peut être :

— Un identificateur : Ecrire(A) $\Longrightarrow$ Affichera la valeur de A
— Un message : Ecrire(`"Bonjour"`) $\Longrightarrow$ Affichera le message Bonjour
— Une opération : Ecrire(A+B) $\Longrightarrow$ Affichera le résultat du calcul de A+B

Nous allons maintenant améliorer l'algorithme 4 afin qu'il puisse affiché à l'utilisateur le résultat du carré calculé :

Algorithme 7 Carré V3

Variables : X,Y : Réel
Début
Lire(X)
$Y \leftarrow X \hat{} 2$
Ecrire(X)
Fin

L'algorithme maintenant affichera à l'utilisateur le carré de la valeur saisit de X.

Mais avant de lire une variable, il est très fortement conseillé d'afficher des message à l'écran avec l'instruction "Ecrire" , afin de prévenir l'utilisateur de ce qu'il doit frapper :

Voici une autre amélioration du programme :

Algorithme 8 Carré V4

Variables : X,Y : Réel
Début
Ecrire("Donner un nombre : ")
Lire(X)
$Y \leftarrow X \hat{} 2$
Ecrire(X)
Fin

Il est aussi fortement conseillé d'écrire des libellés à l'écran avec l'instruction "Ecrire" avant d'afficher les résultat pour indiquer de quel information s'agit-t-il :
Encore une autre amélioration du programme :

Algorithme 9 Carré V5

Variables : X,Y : Réel
Début
Ecrire("Donner un nombre : ")
Lire(X)
$Y \leftarrow X \hat{} 2$
Ecrire("Le carré est : ")
Ecrire(X)
Fin

Remarque:. *Lorsque plusieurs instructions d'affichage successives sont présentes, il est possible de les regrouper en une seule instruction de sortie. Cela permet de simplifier le code et de rendre les messages affichés plus cohérents et organisés. En regroupant les instructions de sortie, on évite la répétition et on améliore la lisibilité du programme.*

Encore une autre amélioration du programme :

Algorithme 10 Carré V6

Variables : X,Y : Réel
Début
  Ecrire("Donner un nombre : ")
  Lire(X)
  $Y \leftarrow X \hat{} 2$
  Ecrire("Le carré est : ", X)
Fin

Retourne à l'algorithme de l'**Exemple 1**, que nous allons le compléter avec les instructions de sortie :

Algorithme 11 Montant TTC

Variables : PUHT, MTTC, MHT, MTVA : Réel
  N : Entier
Constantes : $TVA \leftarrow 0.2$
Début
  Ecrire("Donner le prix HT d'un PC :")
  Lire(PUHT)
  Ecrire("Donner le nombre de PC : ")
  Lire(N)
  $MHT \leftarrow PUHT \times N$
  $MTVA \leftarrow MHT \times TVA$
  $MTTC \leftarrow MHT + MTVA$
  Ecrire("Le montant TTC est : ", MTTC)
Fin

1.5 Les structures de contrôle

Les structures de contrôle sont des constructions syntaxiques qui définissent l'ordre d'exécution des instructions dans un programme. Elles permettent de modifier le chemin d'exécution en fonction de conditions spécifiques ou de la répétition d'opérations. Les structures de contrôle jouent un rôle crucial dans la conception des algorithmes, car elles permettent de créer des programmes capables de réagir à des situations diverses et d'effectuer des tâches complexes.

Les Structures de contrôle permettent de contrôler l'ordre d'exécution des instructions (le séquencement) d'un algorithme. Il existe deux types de structures de contrôle fondamentales :

- Les Structures Conditionnelles (Testes) : Permettent d'exécuter une suite d'instructions (bloc) si une condition est vérifiée (vrai).
- Les Structures répétitives (Itératives / Boucles) : Permettent de répéter l'exécution d'une suite d'instructions (bloc) un certain nombre de fois.

1.5.1 Les structures de contrôle conditionnelles (Les Testes)

Souvent les problèmes nécessitent l'étude de plusieurs situations qui ne peuvent pas être traitées par les séquences d'actions simples. Puisqu'on a plusieurs situations, et

qu'avant l'exécution, on ne sait pas à quel cas de figure on aura à exécuter, dans l'algorithme on doit prévoir tous les cas possibles.

1.5.1.1 La structure conditionnelle alternative

Syntaxe :

Si condition **Alors**
    Traitement 1
Sinon
    Traitement 2
Fin Si

Fonctionnement :

La "condition" est un prédicat, qui peut être vrai ou faux, selon les valeurs des paramètres la constituant.

- Si la condition est vérifiée (sa valeur est vrai), c'est la ¡traitement 1¿ qui sera exécutée. Ensuite, le système passe à l'exécution juste après le FinSi.
- Dans le cas contraire, lorsque la condition n'est pas vérifiée (valeur de la condition est faux), le "traitement 2" qui s'exécute, en cas où celle ci existe (facultative). Si elle n'existe pas, le système passe directement à l'instruction qui suit le FinSi.
- Les suites traitements 1 et 2, peuvent être des actions simples ou même des structures conditionnelles.

Organigramme :

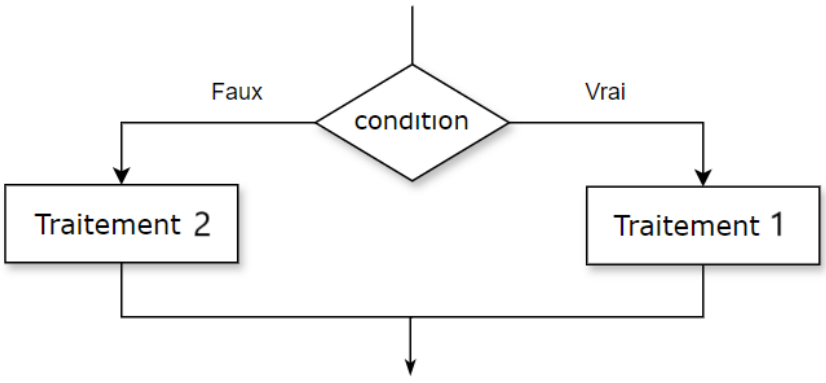

FIGURE 1.5 – Organigramme de la structure conditionnelle alternative

Exemple :

Algorithme 12 POS-NEG

Variables : A : Réel
Début
  Ecrire("Donner un nombre: ")
  Lire(A)
  Si A $<$ 0 **Alors**
    Ecrire("Le nombre est strictement négatif")
  Sinon
    Ecrire("Le nombre est positif")
  Fin Si
Fin

1.5.1.2 La structure conditionnelle simple

Lorsque nous n'avons pas de traitement à effectuer si la condition est fausse, il est possible d'omettre la partie "Sinon" dans la structure conditionnelle alternative. Dans ce cas, on parle d'une structure conditionnelle simple, qui se limite à exécuter une action uniquement si la condition est vraie.

Syntaxe :

Si condition **Alors**
  Traitement
Fin Si

Fonctionnement :

- Si la condition est vérifiée (sa valeur est vrai), le "Traitement" qui sera exécutée. Ensuite, le système passe à l'exécution juste après le FinSi.
- Dans le cas contraire, lorsque la condition n'est pas vérifiée (valeur de la condition est faux), le système passe directement à l'instruction qui suit le FinSi car il y a aucun traitement à exécuté.

Organigramme :

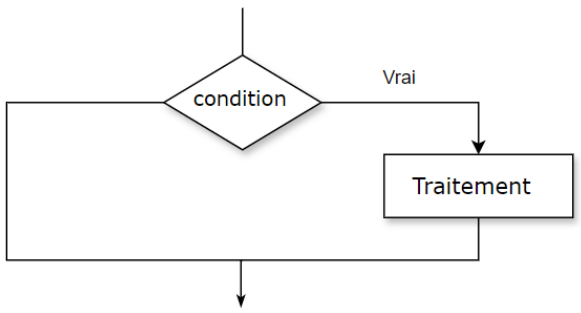

FIGURE 1.6 – Organigramme de la structure conditionnelle simple

Exemple :

Algorithme 13 NUL
Variables : A : Réel
Début
  Ecrire("Donner votre âge : ")
  Lire(A)
  Si A >= 18 **Alors**
    Ecrire("Vous êtes majeur")
  Fin Si
  Ecrire("Fin de l'algorithme")
Fin

1.5.1.3 La structure conditionnelle imbriquée

Les structures conditionnelles imbriquées sont des instructions de contrôle dans lesquelles une instruction conditionnelle SI est placée à l'intérieur d'une autre instruction conditionnelle. Elles permettent de tester plusieurs conditions successives et d'exécuter différentes instructions en fonction des résultats des tests.

Les structures conditionnelles imbriquées permettent de gérer des situations plus complexes où plusieurs décisions doivent être prises de manière séquentielle. Dans ce type de structure, une deuxième condition est évaluée uniquement si la première condition est vraie ou fausse (selon la logique de l'algorithme).

Exemple :

Supposant que nous voulons écrire un algorithme qui fait la résolution des équation du premier degré de la forme $Ax + B = 0$.

Les données d'entré pour ce problème sont les coefficient A et B. Nous avons trois cas de figure à traiter.

- **Si** $A \neq 0$:
  - Si A est différent de 0, on peut résoudre directement l'équation. La solution est donnée par la formule suivante :

$$x = -\frac{B}{A}$$

  - L'algorithme calcule alors la valeur de x et l'affiche comme étant la solution unique.
- **Vérification si** $A = 0$:
  - Si $A = 0$, on examine la valeur de B à l'aide d'une structure **Si...Sinon** imbriquée :
    - **Si** $B = 0$: L'équation devient $0 = 0$, ce qui est toujours vrai. Cela signifie que **toutes les valeurs de** x **sont solutions** (équation indéterminée).
    - **Sinon** $(B \neq 0)$: L'équation devient $B = 0$, ce qui est faux. Il n'existe donc **aucune solution** (équation impossible).
- **Sortie des résultats** :

— Selon les valeurs de A et B, l'algorithme affiche l'une des trois sorties possibles :
  — "L'équation est indéterminée : toutes les valeurs de x sont solutions."
  — "Pas de solution : l'équation est impossible."
  — "La solution est : x."

L'organigramme de la solution proposée est le suivant :

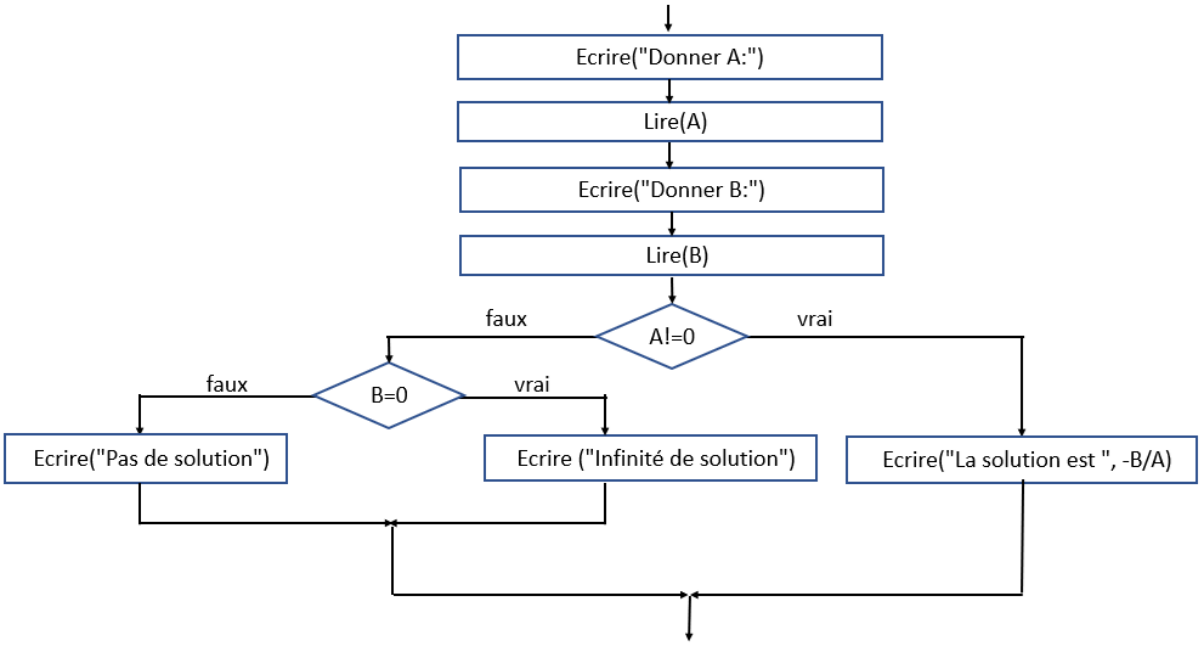

FIGURE 1.7 – Organigramme de la solution

Voici une première version en pseudo code de la solution :

Algorithme 14 EQUATION V1

Variables : A, B : Réel
Début
  Ecrire("Donner A: ")
  Lire(A)
  Ecrire("Donner B: ")
  Lire(B)
  Si A ! = 0 **Alors**
    Ecrire("La solution est : ", -B/A)
  Sinon
    Si B == 0 **Alors**
      Ecrire("Infinité de solutions")
    Sinon
      Ecrire("Pas de solution!")
    Fin Si
  Fin Si
Fin

Structure conditionnelle multiple

Lorsque notre solution nécessite de vérifier plusieurs conditions, l'utilisation de Si imbriqués à plusieurs niveaux peut rendre le code lourd et difficile à lire. Dans ce genre de situation, il est préférable d'utiliser une syntaxe plus claire appelée structure conditionnelle multiple. Cette construction algorithmique permet de tester plusieurs conditions de manière séquentielle et d'exécuter une action en fonction de la première condition vérifiée, améliorant ainsi la lisibilité et la gestion du code.

Syntaxe :

Si condition 1 **Alors**
    Traitement 1
Sinon Si consition 2 **Alors**
    Traitement 2
Sinon Si consition 3 **Alors**
    Traitement 4
    ...
Sinon
    Traitement par défaut
Fin Si

Fonctionnement :

La Condition 1 est vérifiée en premier lieu. Si elle est vraie, alors Traitement 1 est exécuté. Si elle est fausse, l'algorithme passe à la condition suivante.

Si Condition 1 est fausse, alors Condition 2 est évaluée. Si Condition 2 est vraie, Traitement 2 est exécuté. Sinon, l'algorithme passe à la condition suivante.

Ensuite, si Condition 2 est également fausse, Condition 3 est évaluée. Si Condition 3 est vraie, un ensemble de traitements (Traitement 4, etc.) est exécuté.

Si aucune des conditions précédentes n'est vraie, le bloc Else est activé, et Traitement par défaut est exécuté, ce qui permet de gérer les cas où aucune des conditions n'a été satisfaite.

Voici une deuxième version de la solution précédente :

Algorithme 15 EQUATION V2

Variables : A, B : Réel
Début
  Ecrire("Donner A: ")
  Lire(A)
  Ecrire("Donner B: ")
  Lire(B)
  Si A ! = 0 **Alors**
    Ecrire(”La solution est : ”, -B/A)
  Sinon Si B == 0 **Alors**
    Ecrire("Infinité de solutions")
  Sinon
    Ecrire("Pas de solution!")
  Fin Si
Fin

1.5.2 Les structures répétitives (Les boucles)

Dans les problèmes quotidiens, on ne traite pas uniquement des séquences d'actions, sous ou sans conditions, mais il peut être fréquent d'être obligé d'exécuter un traitement (séquence d'actions), plusieurs fois. Ce problème est résolu à l'aide des structures répétitives. Celles ci permettent de donner un ordre de répétition d'une action ou d'une séquence d'actions une ou plusieurs fois.

Les boucles sont des structures de contrôle très importantes en algorithmique. Elles permettent de répéter une séquence d'instructions un certain nombre de fois, soit de manière déterminée, soit jusqu'à ce qu'une condition soit remplie. Elles évitent la redondance de code et permettent d'automatiser des tâches répétitives. On distingue principalement deux types de boucles : les boucles conditionnelles (TantQue) et les boucles à itération fixe (Pour).

1.5.2.1 La boucle Pour

Une boucle ”pour” est une structure de contrôle utilisée en programmation pour répéter un ensemble d'instructions un nombre déterminé de fois. Elle est généralement utilisée lorsque le nombre d'itérations est connu à l'avance.

Syntaxe :

La syntaxe de la boucle est la suivante :

Pour *variable* =VI **à** VF **Faire**
  Traitement
Fin Pour

Explication :

- variable : La variable qui va prendre les différentes valeurs dans la boucle.
- VI : La valeur initial, c'est la première valeur que prendra la variable au début de la boucle.
- VF : La valeur final. C'est la dernière valeur que prendra la variable à la fin de la boucle.
- Traitement : Ce sont les actions qui seront exécutées à chaque itération.

Exemple : La boucle suivante permet d'afficher les nombre 1 jusqu'à 100.

Pour $i = 1$ **à** 100 **Faire**
    Ecrire(i)
Fin Pour

Fonctionnement :

— Initialisation : La boucle commence par initialiser la variable de contrôle (i dans l'exemple) à la valeur initiale VI (1 dans l'exemple).

— Condition de continuation : Avant chaque itération, le programme vérifie si la variable de contrôle est inférieure ou égale à la valeur finale VF (100 dans l'exemple). Si cette condition est vraie, le programme entre dans la boucle.

— Exécution des instructions : Une fois la condition vérifiée, les instructions du traitement à l'intérieur de la boucle sont exécutées. Dans l'exemple, cela signifie afficher la valeur de i.

— Incrémentation : Après l'exécution des instructions, la variable de contrôle (i) est automatiquement incrémentée (généralement de 1, sauf indication contraire).

— Répétition : Le processus se répète : la condition est vérifiée à nouveau. Si elle est toujours vraie, le programme exécute à nouveau les instructions. Sinon, la boucle se termine.

— Fin de la boucle : Une fois que la variable de contrôle dépasse la valeur finale, le programme sort de la boucle et continue avec le code suivant.

Exemples :

Exemple 1 : Une boucle qui affiche 'Bonjour' 10 fois

Pour $i = 1$ **à** 10 **Faire**
    Ecrire("Bonjour")
Fin Pour

Exemple 2 : Une boucle qui calcule la somme de 1 à 5

$s \leftarrow 0$
Pour $i = 1$ **à** 5 **Faire**
    $s \leftarrow s + 1$
Fin Pour

Exemple 3 : Une boucle qui affiche la table de multiplication de 7

$n \leftarrow 7$
Pour $i = 1$ **à** 10 **Faire**
    Ecrire(n, "x", i , "=", n*i)
Fin Pour

1.5.2.2 La boucle TantQue

Une boucle "tant que" est une structure de contrôle de flux utilisée en programmation pour répéter un bloc d'instructions tant qu'une condition spécifiée reste vraie. Elle est particulièrement utile lorsque le nombre d'itérations n'est pas connu à l'avance et dépend d'une condition dynamique.

Syntaxe :

TantQue condition **Faire**
    Traitement
Fin TantQue

Organigramme :

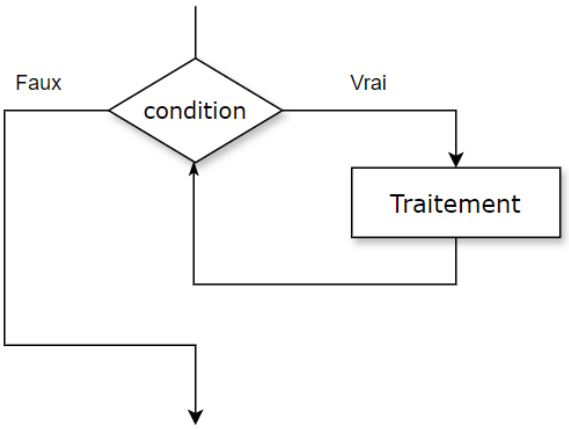

FIGURE 1.8 – La boucle TantQue

Fonctionnement :

- Évaluation de la condition : Avant chaque itération, la condition est vérifiée. Si elle est vraie, le bloc d'instructions à l'intérieur de la boucle est exécuté.
- Exécution des instructions : Les instructions à l'intérieur de la boucle sont exécutées.
- Répétition : Après l'exécution des instructions, la condition est vérifiée à nouveau. Si elle reste vraie, la boucle continue de s'exécuter.
- Fin de la boucle : Lorsque la condition devient fausse, la boucle se termine et le contrôle passe à l'instruction suivante après la boucle.

Attention : Dans le corps de la boucle (instructions), au moins une instruction doit obligatoirement faire évoluer les valeurs des variables testés dans la condition. Sinon la boucle sera infinie (c-à-d qu'elle ne s'arrête jamais).

Exemple : Une boucle infinie qui affiche le mot Informatique à l'infinie

$i \leftarrow 1$
TantQue $i \leq 10$ **Faire**
    Ecrire("Informatique")
Fin TantQue

Exemples :

Exemple 1 : Une boucle qui affiche 'Bonjour' 10 fois

$i \leftarrow 1$
TantQue $i \leq 10$ **Faire**
    Ecrire("Bonjour")
    $i \leftarrow i + 1$
Fin TantQue

Exemple 2 : Une boucle qui calcule la somme de 10 entier entré par l'utilisateur

$s \leftarrow 0$ ▷ Initialiser la variable somme
$i \leftarrow 1$ ▷ Initialiser le compteur de la boucle
TantQue $i \leq 10$ **Faire**
    Ecrire(("Donner un nombre : "))
    Lire(a)
    $s \leftarrow s + a$ ▷ Accumuler les valeurs de a dans la variable s
    $i \leftarrow i + 1$ ▷ Incrémenter le compteur de la boucle i
Fin TantQue

Exemple 3 : Deviner un nombre de l'intervalle [1, 10].

Algorithme 16 Deviner un nombre

Variables : essai, N, nombre_secret : Entier
Début

```
1:  nombre_secret ← 7
2:  essai ← −1
3:  TantQue essai ≠ nombre_secret Faire
4:      Lire essai
5:      Si essai < nombre_secret Alors
6:          Ecrire("Trop petit")
7:      Sinon Si essai > nombre_secret Alors
8:          Ecrire("Trop grand")
9:      Sinon
10:         Ecrire ("Bravo !")
11:     Fin Si
12: Fin TantQue
```

1.5.2.3 La boucle Répétez. . .Jusqu'à

Une boucle « **Répétez. . .Jusqu'à** » est une structure de contrôle qui **exécute d'abord le bloc d'instructions**, puis teste la condition d'arrêt. Elle est particulièrement utile lorsque l'on veut garantir **au moins une exécution** du bloc, et que le **nombre d'itérations** n'est pas connu à l'avance.

Syntaxe :

Répétez
    Traitement
Jusqu'à condition_d_arrêt

Organigramme :

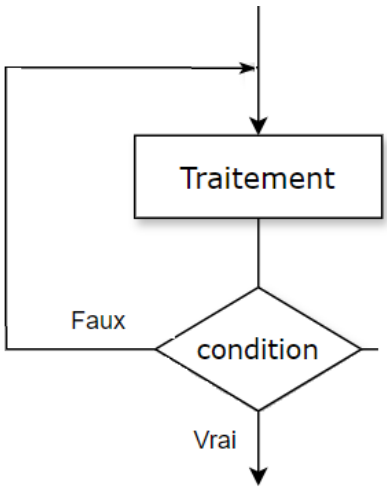

FIGURE 1.9 – La boucle Répétez. . .Jusqu'à

Fonctionnement :

- **Exécution du bloc** : le corps de la boucle est exécuté une première fois sans test préalable.
- **Test de la condition d'arrêt** : à la fin de chaque itération, on teste la condition.
- **Arrêt si vrai** : si la condition devient **vraie**, la boucle s'arrête ; sinon, une nouvelle itération commence.
- **Garantie d'une itération** : le bloc est toujours exécuté au moins une fois.

Attention :
La condition doit **finir par devenir vraie**. Si aucune instruction ne fait évoluer les variables impliquées dans la condition d'arrêt, la boucle peut devenir **infinie**.

Exemple (boucle infinie) : affiche « Informatique » à l'infini (car la condition d'arrêt n'est jamais vraie).

Répétez
    Ecrire("Informatique")
Jusqu'à faux

Exemples :

Exemple 1 : Afficher Bonjour 10 fois

$i \leftarrow 1$
Répétez
    Ecrire("Bonjour")
    $i \leftarrow i + 1$
Jusqu'à $i > 10$

Exemple 2 : Somme de valeurs jusqu'à saisie de 0 (sentinelle)

$s \leftarrow 0$ ▷ Somme
Répétez
    Ecrire("Donner un nombre (0 pour terminer) :")
    Lire(a)
    $s \leftarrow s + a$
Jusqu'à $a = 0$
Ecrire("Somme = ", s)

Exemple 3 : Deviner un nombre dans l'intervalle $[1, 10]$

Algorithme 17 Deviner un nombre (version Répétez...Jusqu'à)

Variables : essai, nombre_secret : Entier
Début

```
1: nombre_secret ← 7
2: Répétez
3:     Lire essai
4:     Si essai < nombre_secret Alors
5:         Ecrire("Trop petit")
6:     Sinon Si essai > nombre_secret Alors
7:         Ecrire("Trop grand")
8:     Sinon
9:         Ecrire("Bravo !")
10:    Fin Si
11: Jusqu'à essai = nombre_secret
```

Deuxième partie

Programmation C

Chapitre 2

Programmation C

2.1 Introduction Générale

2.1.1 Historique

Le langage `C` est l'un des langages les plus influents de l'informatique. Conçu au début des années 1970 chez Bell Labs, il a servi de base à de nombreux langages modernes (C++, Objective-C, C#, Java, Go, Rust, etc.) et reste omniprésent dans les systèmes d'exploitation, l'embarqué et les bibliothèques de bas niveau.

Repères historiques :

— **1966** : Martin Richards crée **BCPL**, un langage simple et portable destiné aux compilateurs.

— **1969** : Ken Thompson conçoit **B** (inspiré de BCPL) chez Bell Labs pour le développement d'UNIX.

— **1972–1973** : Dennis Ritchie développe **C** chez Bell Labs, en y ajoutant des types et des structures de données plus riches que B.

— **1973** : le noyau d'**UNIX** est réécrit en C, démontrant sa puissance et sa portabilité.

— **1978** : Publication du livre *The C Programming Language* par Brian W. Kernighan et Dennis M. Ritchie (*K&R*). Ce livre sert de référence de facto et popularise massivement C.

— **1989** : normalisation **ANSI C** (ANSI X3.159–1989, dit **C89**).

— **1990** : adoption ISO du standard (ISO/IEC 9899 :1990, dit **C90**).

— **1995** : **Amendement 1** (**C95**) : ajouts mineurs (internationalisation, élargissements de la bibliothèque).

— **1999** : **C99** (ISO/IEC 9899 :1999) : *inline*, commentaires `//`, types entiers exacts `stdint.h`, tableaux à longueur variable (VLA), `restrict`, `long long`, fonctions math étendues, `complex.h`.

— **2011** : **C11** (ISO/IEC 9899 :2011) : `Atomic`, threads (`threads.h`), alignement (`alignas`/`alignof`), assertions statiques, améliorations `Unicode` (`uchar.h`), atomiques mémoire.

— **2018** : **C17** (souvent appelé **C18**) : révision de maintenance corrigeant des errata de C11, sans nouvelles fonctionnalités majeures.

— **2023** : **C23** : révision moderne avec des améliorations de convivialité (littéraux, attributs, qualité de vie du langage), corrections et extensions de bibliothèque ; orientation vers une meilleure interopérabilité et une écriture plus sûre.

2.1.2 Caractéristiques du langage C

Le langage C se distingue par plusieurs caractéristiques majeures :

— **Langage compilé** : le code source doit être traduit en langage machine avant exécution.

— **Langage structuré** : il permet de découper un programme en fonctions.

— **Portabilité** : un programme en C peut être exécuté sur de nombreux systèmes après recompilation.

— **Proche du matériel** : il permet un contrôle précis de la mémoire et du matériel, contrairement aux langages interprétés.

— **Performances élevées** : il est utilisé dans les systèmes d'exploitation, les pilotes, l'embarqué et les applications nécessitant une forte efficacité.

2.1.3 Environnement de développement

Pour écrire, compiler et exécuter un programme en langage `C`, il est nécessaire de disposer d'un **environnement de développement**, c'est-à-dire un ensemble d'outils permettant de transformer le code source en programme exécutable.

1. Les composants essentiels

Un environnement de programmation en C comprend généralement :

— **Un éditeur de texte** : pour saisir et enregistrer le code source (ex. : VS Code, Sublime Text, Notepad++, Vim).

— **Un compilateur** : traduit le code source (`.c`) en langage machine. Le plus utilisé est `gcc` (GNU Compiler Collection).

— **Un éditeur de liens (linker)** : assemble les fichiers objets et les bibliothèques pour produire un exécutable.

— **Un débogueur** : outil permettant de tester et corriger les erreurs d'exécution.

2. Environnements intégrés (IDE)

Un **IDE** (Integrated Development Environment) regroupe tous ces outils dans une interface unique et conviviale. Voici quelques environnements populaires pour débuter en C :

— **Code : :Blocks** : simple, gratuit et multiplateforme.

— **Visual Studio Code** : léger et extensible (avec extensions `C/C++` et terminal intégré).

— **Dev-C++** : interface intuitive, adaptée aux débutants.

— **CLion** ou **Eclipse CDT** : plus complets, destinés aux projets plus avancés.

3. Les étapes de compilation d'un programme en C

La transformation d'un programme en langage C en un exécutable passe par plusieurs étapes successives, réalisées par le compilateur :

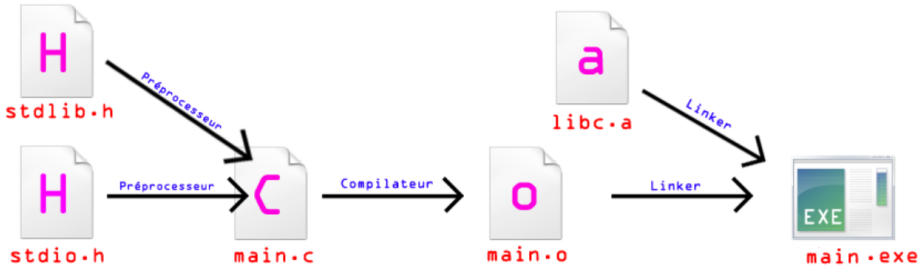

FIGURE 2.1 – Etapes de compilation

— **Prétraitement** : le préprocesseur (`cpp`) traite les directives commençant par #, comme `#include` et `#define`, et produit un fichier source étendu.

— **Compilation** : le compilateur (`gcc`) traduit le code C en langage assembleur, puis en code machine pour générer un ou plusieurs **fichiers objets** (`.o`).

— **Édition de liens (linking)** : l'éditeur de liens (`ld`) assemble les fichiers objets et les bibliothèques nécessaires pour produire le programme exécutable.

— **Exécution** : le fichier exécutable obtenu peut être lancé par le système d'exploitation ; il s'exécute en mémoire et effectue les instructions écrites par le programmeur.

4. Structure des fichiers

Un projet en C contient généralement :

— Un ou plusieurs fichiers source : `.c`

— Des fichiers d'en-tête : `.h`, contenant les déclarations de fonctions et constantes

— Un fichier exécutable : produit après compilation

5. Bonnes pratiques

— Sauvegarder les fichiers fréquemment.

— Compiler souvent pour détecter rapidement les erreurs.

— Ajouter des `commentaires` pour documenter le code.

— Respecter une indentation claire et cohérente.

2.2 Variables, Constantes et Opérateurs

2.2.1 Les variables

Une **variable** est une zone mémoire identifiée par un nom, utilisée pour stocker une valeur pouvant changer au cours du programme.

2.2.1.1 Déclaration d'une variable

Avant utilisation, chaque variable doit être déclarée avec un **type** et un **nom** :

```
type nom_variable;
type nom_variable = valeur_initiale;
```

Exemples :

```
int age = 20;
float temperature = 23.5;
char lettre = 'A';
```

2.2.1.2 Règles de nommage :

- Le nom doit commencer par une lettre ou un underscore (_).
- Il peut contenir des chiffres, mais pas d'espace ni de caractère spécial.
- Les majuscules et minuscules sont distinctes.

Remarque:. *Une variable doit être* ***déclarée avant sa première utilisation****, sinon le compilateur renverra une erreur.*

2.2.2 Les constantes

Une **constante** est une valeur fixe qui ne change pas durant l'exécution du programme.

Définition avec `const` :

```
const float PI = 3.14;
```

Définition avec `#define` :

```
#define PI 3.14
#define MAX 100
```

Différence :

- `const` est gérée par le compilateur (avec typage).
- `#define` est remplacée textuellement par le préprocesseur avant compilation.

2.2.3 Les opérateurs en C

Les **opérateurs** permettent de manipuler les variables et les valeurs. On distingue plusieurs familles :

1. Opérateurs arithmétiques :

Opérateur	Signification
`+`	Addition
`-`	Soustraction
`*`	Multiplication
`/`	Division
`%`	Modulo (reste de la division entière)

Exemples :

```
int a = 10, b = 3;
printf("%d\n", a / b);   // Affiche 3
printf("%d\n", a % b);   // Affiche 1
```

2. Opérateurs de comparaison :

Ils permettent de comparer deux valeurs et retournent `1` (vrai) ou `0` (faux).

Opérateur	Signification
`==`	Égal à
`!=`	Différent de
`>`	Supérieur à
`<`	Inférieur à
`>=`	Supérieur ou égal à
`<=`	Inférieur ou égal à

3. Opérateurs logiques

Ils permettent de combiner plusieurs conditions.

Opérateur	Signification
`&&`	ET logique (AND)
`||`	OU logique (OR)
`!`	NON logique (NOT)

Exemple :

```
if (age > 18 && age < 65)
 printf("Adulte\n");
```

4. Opérateurs d'affectation

Ils permettent d'attribuer une valeur à une variable.

Opérateur	Signification
`=`	Affectation simple
`+=`	Ajoute et affecte
`-=`	Soustrait et affecte
`*=`	Multiplie et affecte
`/=`	Divise et affecte

Exemple :

```
int x = 10;
x += 5;    // équivaut à x = x + 5
```

5. Opérateurs d'incrémentation et de décrémentation

```
i++;   // incrémente de 1
i--;   // décrémente de 1
```

Exemple :

```
int i = 5;
i++;  // i devient 6
++i;  // i devient 7
```

2.3 Fonctions d'Entrées/Sorties et Mathématiques

2.3.1 La fonction d'affichage `printf()`

La fonction `printf()` est utilisée pour **afficher des informations à l'écran** : du texte, des nombres ou des résultats de calculs. Elle est définie dans la bibliothèque standard `<stdio.h>`.

Syntaxe générale :

```
printf("chaîne de format", liste_de_variables);
```

Explications :

- La **chaîne de format** est placée entre guillemets ‘" "‘.
- Elle peut contenir :
  - du texte (affiché tel quel),
  - des **spécificateurs de format** (comme `%d`, `%f`, etc.) qui seront remplacés par les valeurs des variables.
- La **liste de variables** contient les valeurs à afficher, séparées par des virgules.

Exemples simples :

```
int age = 20;
float taille = 1.75;
char lettre = 'A';

printf("Bonjour !\n");
printf("Vous avez %d ans.\n", age);
printf("Votre taille est de %.2f m.\n", taille);
printf("La première lettre est %c.\n", lettre);
```

Résultat à l'écran :

```
Bonjour !
Vous avez 20 ans.
Votre taille est de 1.75 m.
La première lettre est A.
```

Principaux formats d'affichage :

Format	Type de donnée	Exemple d'utilisation
`%d`	Entier (`int`)	`printf("%d", a);`
`%f`	Réel simple précision (`float`)	`printf("%f", x);`
`%lf`	Réel double précision (`double`)	`printf("%lf", y);`
`%c`	Caractère (`char`)	`printf("%c", lettre);`
`%s`	Chaîne de caractères	`printf("%s", nom);`
`%%`	Affiche le caractère %	`printf("%%");`

Affichage formaté :

On peut contrôler la présentation des nombres :

— `%.2f` : affiche un nombre réel avec deux décimales.

— `%10d` : réserve un espace de 10 caractères pour afficher un entier (aligné à droite).

— `%-10d` : aligne à gauche dans un espace de 10 caractères.

Exemples :

```
float x = 3.14159;
printf("x = %.2f\n", x);     // affiche 3.14
printf("x = %.4f\n", x);     // affiche 3.1416
printf("Valeur = %10.2f\n", x);  // aligné à droite
```

Caractères spéciaux utiles :

Caractère	Effet à l'affichage
`\n`	Retour à la ligne
`\t`	Tabulation horizontale
`\"`	Affiche un guillemet double
`\\`	Affiche le caractère antislash (\)

Exemple d'utilisation :

```
printf("Bonjour\tà\ttous !\n");
printf("Ceci est une nouvelle ligne.\n");
```

Résultat :

```
Bonjour    à    tous !
Ceci est une nouvelle ligne.
```

Exemple complet :

```c
#include <stdio.h>

int main() {
    char nom[20] = "Alice";
    int age = 22;
    float note = 15.75;

    printf("Etudiante : %s\n", nom);
    printf("Âge       : %d ans\n", age);
    printf("Moyenne   : %.2f / 20\n", note);

    return 0;
}
```

Résumé :

- `printf()` affiche du texte et des valeurs à l'écran.
- Elle utilise des **formats de conversion** (`%d`, `%f`, `%s`, etc.) selon le type de variable.
- Les caractères spéciaux (`\n`, `\t`, ...) permettent de formater l'affichage.
- Il est possible de contrôler la précision et l'alignement des nombres.

2.3.2 La fonction de lecture `scanf()`

La fonction `scanf()` permet de **lire des données saisies par l'utilisateur** au clavier et de les stocker dans des variables. Elle est définie dans la bibliothèque standard `<stdio.h>`.

Syntaxe générale :

```c
scanf("format", &variable1, &variable2, ...);
```

Explications :

- `"format"` : indique le type de donnée attendu (entier, réel, caractère, etc.).
- Le symbole `&` (esperluette) : fournit à `scanf()` **l'adresse mémoire** de la variable où stocker la donnée.
- Chaque variable lue doit correspondre à un **format de lecture** dans la chaîne.

Exemples simples :

```c
int age;
float prix;
char lettre;

scanf("%d", &age);          // lecture d'un entier
scanf("%f", &prix);         // lecture d'un réel (float)
scanf(" %c", &lettre);      // lecture d'un caractère
```

Remarquez l'espace avant `%c` : elle permet d'ignorer les retours à la ligne et espaces restants dans le tampon clavier.

Lecture multiple :

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

L'utilisateur doit entrer deux entiers séparés par un espace ou une touche "Entrée".

Formats de lecture usuels :

Symbole	Type de donnée	Exemple
`%d`	Entier signé (`int`)	`scanf("%d", &n);`
`%f`	Réel simple précision (`float`)	`scanf("%f", &x);`
`%lf`	Réel double précision (`double`)	`scanf("%lf", &y);`
`%c`	Caractère unique	`scanf(" %c", &lettre);`
`%s`	Chaîne de caractères	`scanf("%s", nom);`

Exemple complet :

```c
#include <stdio.h>

int main() {
    int age;
    float taille;
    char nom[20];

    printf("Entrez votre nom : ");
    scanf("%s", nom);

    printf("Entrez votre âge : ");
    scanf("%d", &age);

    printf("Entrez votre taille (en m) : ");
    scanf("%f", &taille);

    printf("\nBonjour %s, vous avez %d ans et mesurez %.2f m
        .\n", nom, age, taille);
    return 0;
}
```

Remarques importantes :

- Ne jamais oublier le symbole `&` devant les variables (sauf pour les tableaux et chaînes).
- Toujours vérifier la correspondance entre les formats (`%d`, `%f`, etc.) et les types de variables.
- Utiliser un espace avant `%c` pour éviter les erreurs de lecture de caractères.

2.4 Les fonctions mathématiques en C

Le langage C fournit de nombreuses fonctions mathématiques prêtes à l'emploi regroupées dans la bibliothèque standard <math.h>. Ces fonctions permettent d'effectuer des calculs courants : puissances, racines carrées, fonctions trigonométriques, exponentielles, logarithmes, etc.

Inclusion de la bibliothèque : Avant d'utiliser ces fonctions, il faut inclure la bibliothèque <math.h> au début du programme :

```
#include <math.h>
```

Principales fonctions mathématiques

Fonction	Description	Exemple d'utilisation
sqrt(x)	Racine carrée de x	sqrt(16.0) → 4.0
pow(x, y)	Puissance : x^y	pow(2, 3) → 8.0
fabs(x)	Valeur absolue (float)	fabs(-3.5) → 3.5
exp(x)	Exponentielle e^x	exp(1) → 2.71828
log(x)	Logarithme népérien (base e)	log(2.71828) → 1.0
log10(x)	Logarithme décimal (base 10)	log10(100) → 2.0
sin(x)	Sinus (en radians)	sin(M_PI / 2) → 1.0
cos(x)	Cosinus (en radians)	cos(0) → 1.0
tan(x)	Tangente (en radians)	tan(M_PI / 4) → 1.0
ceil(x)	Arrondit au supérieur	ceil(3.2) → 4.0
floor(x)	Arrondit à l'inférieur	floor(3.9) → 3.0
fmod(x, y)	Reste de la division réelle	fmod(7.0, 3.0) → 1.0

Constantes mathématiques utiles

Certaines constantes sont également disponibles dans math.h (selon le compilateur) :

— M_PI : valeur de π (3.14159265)

— M_E : base du logarithme naturel (2.7182818)

Si elles ne sont pas définies, on peut les déclarer soi-même :

```
#define PI 3.14159265
#define E  2.7182818
```

Exemples d'utilisation

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

int main() {
    double x = 9.0, y = 2.0;

    printf("Racine de %.2f = %.2f\n", x, sqrt(x));
    printf("%.2f ^ %.2f = %.2f\n", x, y, pow(x, y));
```

```
9         printf("Logarithme de %.2f = %.2f\n", x, log(x));
10        return 0;
11    }
```

Remarques importantes :

— Les fonctions mathématiques utilisent le type `double` par défaut ; utilisez `float` si vous voulez moins de précision.

— Les angles dans les fonctions trigonométriques sont exprimés en **radians** (et non en degrés).

2.5 structures conditionnelles

Les **structures conditionnelles** permettent d'exécuter des instructions différentes selon qu'une condition soit vraie ou fausse. Elles sont essentielles pour contrôler le déroulement d'un programme.

2.5.1 La structure `if`

Syntaxe générale :

```
if (condition) {
 // instructions exécutées si la condition est vraie
}
```

Exemple :

```
int age = 20;

if (age >= 18) {
    printf("Vous êtes majeur.\n");
}
```

2.5.2 La structure `if ... else`

Syntaxe :

```
if (condition) {
    // instructions si la condition est vraie
}
else {
    // instructions si la condition est fausse
}
```

Exemple :

```c
int note = 12;

if (note >= 10) {
   printf("Admis !\n");
}
else {
   printf("Ajourné.\n");
}
```

2.5.3 La structure `if ... else if ... else`

Syntaxe :

```c
if (condition1) {
   // bloc 1
}
else if (condition2) {
   // bloc 2
}
else {
   // bloc par défaut
}
```

Exemple :

```c
int note = 15;

if (note >= 16)
    printf("Très bien\n");
else if (note >= 14)
    printf("Bien\n");
else if (note >= 10)
    printf("Passable\n");
else
    printf("Echec\n");
```

2.5.4 Les conditions multiples avec opérateurs logiques

On peut combiner plusieurs conditions à l'aide des opérateurs :

— `&&` → ET logique (toutes les conditions doivent être vraies)

— `||` → OU logique (au moins une condition vraie)

— `!` → NON logique (inverse le résultat)

Exemples :

```c
if (age >= 18 && age <= 65)
    printf("Adulte en activité\n");
```

```
if (note < 10 || absences > 3)
    printf("Non validé\n");

if (!(x > 0))
    printf("x est négatif ou nul\n");
```

2.5.5 La structure `switch`

La structure `switch` permet de simplifier les tests d'égalité multiples sur une même variable. Elle est particulièrement utile pour les menus ou les sélections de cas.

Syntaxe générale :

```
switch (expression) {
 case valeur1:
    // instructions
    break;

 case valeur2:
    // instructions
    break;

 default:
    // instructions par défaut
}
```

Exemple :

```
1    int jour = 3;
2
3    switch (jour) {
4        case 1:
5            printf("Lundi\n");
6            break;
7        case 2:
8            printf("Mardi\n");
9            break;
10       case 3:
11           printf("Mercredi\n");
12           break;
13       default:
14           printf("Jour inconnu\n");
15   }
```

Bonnes pratiques

— Toujours utiliser des **accolades** même pour une seule instruction (meilleure lisibilité).

— Ajouter un **break** après chaque `case` pour éviter l'exécution des cas suivants.
— Soigner l'indentation des blocs conditionnels.
— Tester toutes les possibilités, y compris le `default`.

Astuce :
Toujours utiliser des accolades pour éviter les erreurs de logique dans les blocs conditionnels.

2.6 Les structures itératives (boucles)

Les **structures itératives** permettent de répéter un ensemble d'instructions plusieurs fois. Elles sont utilisées pour automatiser les traitements répétitifs comme les calculs, les saisies multiples ou les parcours de tableaux.

2.6.1 La boucle `while`

Principe :

La boucle `while` répète un bloc d'instructions **tant qu'une condition est vraie**.

Syntaxe :

```
while (condition) {
 // instructions répétées
}
```

Exemple :

```
int i = 1;

while (i <= 5) {
 printf("i = %d\n", i);
 i++;
}
```

Résultat :

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

Attention
Si la condition reste toujours vraie, la boucle devient **infinie**. Il faut donc s'assurer que la condition se modifie à chaque itération.

2.6.2 La boucle `do ... while`

Principe :

La boucle `do ... while` exécute le bloc d'instructions **au moins une fois**, puis répète tant que la condition est vraie.

Syntaxe :

```
do {
 // instructions
} while (condition);
```

Exemple :

```
int n;

do {
 printf("Entrez un nombre positif : ");
 scanf("%d", &n);
} while (n <= 0);

printf("Merci, vous avez entré %d.\n", n);
```

2.6.3 La boucle `for`

Principe :

La boucle `for` est utilisée lorsque le nombre d'itérations est connu à l'avance. Elle regroupe en une seule ligne : - l'initialisation, - la condition, - la mise à jour.

Syntaxe :

```
for (initialisation; condition; mise_a_jour) {
 // instructions répétées
}
```

Exemple :

```
for (int i = 1; i <= 10; i++) {
 printf("%d ", i);
}
```

Résultat :

```
1 2 3 4 5 6 7 8 9 10
```

2.6.4 Instructions de contrôle de boucle

1. `break`

Permet de sortir immédiatement de la boucle, même si la condition est encore vraie.

```
for (int i = 0; i < 10; i++) {
 if (i == 5) break;
 printf("%d ", i);
}
```

Résultat : affiche les nombres de 0 à 4.

2. `continue`

Passe à l'itération suivante sans exécuter les instructions restantes du bloc.

```
for (int i = 1; i <= 5; i++) {
 if (i == 3) continue;
 printf("%d ", i);
}
```

Résultat : affiche 1 2 4 5.

Exemple complet

```c
#include <stdio.h>

int main() {
    int n, i, somme = 0;

    printf("Entrez un entier positif : ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++) {
        somme += i;
    }

    printf("La somme des %d premiers entiers est : %d\n", n,
        somme);
    return 0;
}
```

Résultat :

```
Entrez un entier positif : 5
La somme des 5 premiers entiers est : 15
```