UNIVERSITÉ DE DOUALA
ÉCOLE SUPÉRIEURE TECHNIQUE LA SALLE

ALGORITHMIQUE

Support de cours

Classe : **IE4**

Volume horaire : 24 h (CM 20 h — TD 4 h)

Pr Moffo Lonla Bertrand

Année académique 2026-2027

Présentation du cours

Objectifs pédagogiques

À l'issue de ce cours, l'étudiant doit être capable :

— d'appliquer une **méthode de programmation** : analyser un problème, le décomposer, écrire un algorithme clair et structuré en pseudo-code ;

— d'**évaluer la complexité** d'un algorithme (nombre d'opérations en fonction de la taille des données, notation O) ;

— de **simuler** (tracer à la main) l'exécution d'un algorithme pour en vérifier le résultat ou découvrir ce qu'il réalise.

Organisation

Séance	Date	Contenu	CM	TD
1	20/11/2026	Notions de base : éléments d'un algorithme, simulation (chap. 1)	3 h	1 h
2	27/11/2026	Instructions élémentaires et expressions (chap. 2)	3 h	1 h
3	04/12/2026	Structures décisionnelles et boucles (chap. 3)	3 h	1 h
4	04/12/2026	Variables structurées : tableaux, structures ; algorithmes sur les tableaux (chap. 4)	4 h	—
5	11/12/2026	Fonctions et procédures (chap. 5) + **CC (2 h)**	2 h	—
6	18/12/2026	Fonctions, procédures et pointeurs, suite et fin (chap. 5–6)	3 h	1 h

Conventions d'écriture

Tous les algorithmes du cours sont écrits dans le même pseudo-code : mots-clés en **gras**, affectation notée « ← », lignes d'instructions numérotées (ce qui facilite la simulation), commentaires entre `/*` et `*/`. L'annexe A donne la traduction de chaque construction en langage C.

Table des matières

Présentation du cours **1**

1 Notions de base **3**
1.1 Qu'est-ce qu'un algorithme ? . . . 3
1.2 Du problème au programme . . . 3
1.3 Structure d'un algorithme . . . 4
1.4 Les éléments d'un algorithme . . . 4
1.4.1 Les identificateurs . . . 4
1.4.2 Les types de base . . . 5
1.4.3 Les constantes . . . 5
1.4.4 Les variables . . . 5
1.4.5 Les commentaires . . . 6
1.5 Simulation d'un algorithme . . . 6
1.5.1 Méthode : le tableau de simulation . . . 6
1.6 Première approche de la complexité . . . 7

2 Instructions élémentaires et expressions **8**
2.1 L'affectation . . . 8
2.1.1 Exemple classique : la permutation de deux variables . . . 8
2.2 La lecture et l'écriture . . . 9
2.2.1 La lecture : `Lire` . . . 9
2.2.2 L'écriture : `Écrire` . . . 9
2.3 Les expressions . . . 9
2.3.1 Expressions arithmétiques . . . 9
2.3.2 Expressions relationnelles (comparaisons) . . . 10
2.3.3 Expressions logiques . . . 10
2.3.4 Priorité des opérateurs . . . 10
2.3.5 Fonctions prédéfinies usuelles . . . 10
2.4 Un algorithme complet . . . 11

3 Structures décisionnelles et boucles **12**
3.1 Les structures décisionnelles . . . 12
3.1.1 L'alternative simple : Si...alors...FinSi . . . 12
3.1.2 L'alternative complète : Si...alors...Sinon...FinSi . . . 12
3.1.3 Alternatives imbriquées et SinonSi . . . 12
3.1.4 Le choix multiple : Selon . . . 13
3.2 Les structures répétitives (boucles) . . . 13
3.2.1 La boucle Pour . . . 13
3.2.2 La boucle TantQue . . . 14
3.2.3 La boucle Répéter...Jusqu'à . . . 14
3.2.4 Choisir la bonne boucle . . . 15
3.2.5 Schémas classiques . . . 15
3.2.6 Boucles imbriquées . . . 16
3.3 Complexité des algorithmes . . . 16
3.3.1 Compter les opérations . . . 16
3.3.2 La notation O (grand O) . . . 16

4 Variables structurées : tableaux et structures **18**
4.1 Les tableaux à une dimension . . . 18
4.1.1 Déclaration . . . 18
4.1.2 Accès aux éléments . . . 18
4.1.3 Parcours : saisie et affichage . . . 18
4.2 Algorithmes classiques sur les tableaux . . . 19
4.2.1 Somme, moyenne, maximum . . . 19
4.2.2 Recherche séquentielle . . . 19
4.2.3 Recherche dichotomique (tableau trié) . . . 20
4.2.4 Tri par sélection . . . 20
4.2.5 Tri à bulles . . . 21
4.3 Les tableaux à deux dimensions (matrices) . . . 21
4.4 Les structures (enregistrements) . . . 22
4.4.1 Tableaux de structures . . . 22

5 Fonctions et procédures **24**
5.1 Pourquoi des sous-programmes ? . . . 24
5.2 Les fonctions . . . 24
5.2.1 Définition . . . 24
5.2.2 Appel d'une fonction . . . 24
5.3 Les procédures . . . 25
5.4 Variables locales et globales . . . 25
5.5 Modes de passage des paramètres . . . 26
5.5.1 Passage par valeur (paramètre d'entrée, E) . . . 26
5.5.2 Passage par référence (paramètre de sortie S ou d'entrée/sortie E/S) . . . 26
5.6 Simulation d'un appel . . . 27
5.7 La récursivité . . . 27

6 Les pointeurs **28**
6.1 Adresse et pointeur . . . 28
6.2 Notations . . . 28
6.3 Pointeurs et passage de paramètres . . . 29
6.4 Allocation dynamique . . . 29
6.5 Pointeurs et tableaux . . . 30
6.6 Ouverture : les listes chaînées . . . 30

A Correspondance pseudo-code / langage C **31**

CHAPITRE 1

Notions de base

1.1 Qu'est-ce qu'un algorithme ?

Définition

Un **algorithme** est une suite *finie* et *ordonnée* d'instructions *non ambiguës* qui, appliquée à des données d'entrée, permet d'obtenir en un temps fini un résultat (données de sortie).

Le mot vient du nom du mathématicien perse *Al-Khwarizmi* (IX^{e} siècle). Nous utilisons des algorithmes en permanence sans les nommer : une recette de cuisine, la procédure de retrait d'argent à un guichet automatique, la méthode de la division posée apprise à l'école primaire.

Un algorithme possède les propriétés suivantes :

— **Finitude** : il se termine après un nombre fini d'étapes ;

— **Précision** : chaque instruction a un sens unique ;

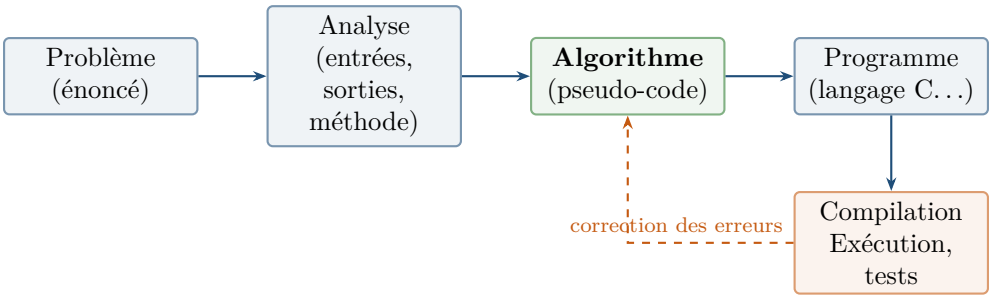
— **Entrées** : zéro, une ou plusieurs données fournies au départ ;

— **Sorties** : au moins un résultat ;

— **Efficacité** : chaque instruction est réalisable avec les moyens disponibles.

1.2 Du problème au programme

La programmation est une démarche en plusieurs étapes. L'algorithme est *indépendant* de tout langage : il décrit *la méthode*, le programme la traduit dans un langage compréhensible par la machine (C, Python, Java…).



Remarque

L'étape d'**analyse** est la plus importante. Avant d'écrire la moindre instruction, il faut répondre à trois questions : **(1)** de quelles données dispose-t-on (entrées) ? **(2)** que veut-on obtenir (sorties) ? **(3)** comment passe-t-on des unes aux autres (traitement) ?

Exemple : analyse du calcul de l'aire d'un cercle

— Entrée : le rayon r (réel positif) ;
— Sortie : l'aire A (réel) ;
— Traitement : $A = \pi r^2$, avec $\pi \approx 3{,}14159$ (constante).

1.3 Structure d'un algorithme

Un algorithme comporte trois parties :

1. l'**en-tête**, qui donne son nom ;
2. la partie **déclarations** (constantes, types, variables) : on y annonce *tous* les objets manipulés ;
3. le **corps**, encadré par **Début** et **Fin**, qui contient les instructions exécutées dans l'ordre.

Structure générale d'un algorithme

Algorithme : NomDeLAlgorithme
Constantes
    NOM_CONSTANTE : = valeur
Types
    nom_type : définition du type
Variables
    nom_variable : type

```
1 Début
2     instruction 1
3     instruction 2
4     ...
5 Fin
```

Exemple : aire d'un cercle

Algorithme : AireCercle
    ** Calcule l'aire d'un cercle dont le rayon est saisi **
Constantes
    PI : = 3.14159
Variables
    r, aire : réel

```
1 Début
2     Écrire("Donner le rayon : ")
3     Lire(r)
4     aire ← PI * r * r
5     Écrire("L'aire est : ", aire)
6 Fin
```

1.4 Les éléments d'un algorithme

1.4.1 Les identificateurs

Définition

Un **identificateur** est le nom donné à un objet de l'algorithme (variable, constante, type, fonction...).

Règles de formation :

— il commence obligatoirement par une **lettre** (ou le caractère _) ;

— il ne contient que des lettres non accentuées, des chiffres et le caractère _ (pas d'espace, pas de tiret, pas de caractères spéciaux) ;

— il ne doit pas être un **mot réservé** (Si, Pour, Début, Fin...) ;

— il doit être **significatif** : `prixHT` est préférable à `x1`.

Corrects	Incorrects	
`note`	`1x`	commence par un chiffre
`prix_HT`	`prix HT`	contient un espace
`moyenne2`	`note-max`	contient un tiret
`nbEtudiants`	`Fin`	mot réservé

1.4.2 Les types de base

Le **type** d'un objet définit l'ensemble des valeurs qu'il peut prendre et les opérations qu'on peut lui appliquer.

Type	Valeurs	Exemples
entier	nombres entiers relatifs	`-12`, `0`, `2026`
réel	nombres à virgule (point décimal)	`3.14`, `-0.5`, `1.0`
caractère	un seul symbole, entre apostrophes	`'A'`, `'7'`, `'?'`
chaîne	suite de caractères, entre guillemets	`"Bonjour"`, `"IE4"`
booléen	deux valeurs logiques	**Vrai**, **Faux**

Attention

`'7'` (caractère), `"7"` (chaîne) et `7` (entier) sont trois objets *différents*. On ne peut pas les additionner entre eux.

1.4.3 Les constantes

Définition

Une **constante** est un objet dont la valeur est fixée lors de la déclaration et ne peut *jamais* être modifiée pendant l'exécution.

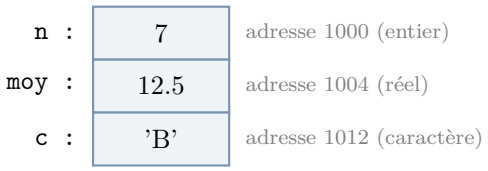
Exemples : `PI = 3.14159`, `TVA = 0.1925`, `NMAX = 100`. Par convention, on écrit les constantes en MAJUSCULES.

1.4.4 Les variables

Définition

Une **variable** est un emplacement de la mémoire, désigné par un identificateur, qui contient une valeur d'un type donné. Cette valeur peut changer au cours de l'exécution.

Une variable possède donc : un **nom**, un **type**, une **valeur** et une **adresse** en mémoire (cette dernière sera exploitée au chapitre 6).



Remarque

Au début de l'algorithme, une variable déclarée mais non initialisée a une **valeur indéterminée**. L'utiliser avant de lui avoir donné une valeur est une erreur classique.

1.4.5 Les commentaires

Un commentaire est un texte ignoré lors de l'exécution, destiné au lecteur. On l'écrit entre `/*` et `*/`. Il explique *pourquoi* on fait quelque chose, pas *ce que* fait une instruction évidente.

1.5 Simulation d'un algorithme

Définition

Simuler (ou *tracer*) un algorithme, c'est exécuter ses instructions à la main, une par une, en notant après chacune la valeur de toutes les variables.

La simulation sert à :

— **vérifier** qu'un algorithme produit le résultat attendu ;

— **découvrir** ce que réalise un algorithme donné ;

— **localiser une erreur**.

1.5.1 Méthode : le tableau de simulation

1. Numéroter les lignes d'instructions.
2. Construire un tableau : une colonne « N° ligne », une colonne par variable, une colonne « Écran » (affichages) ; éventuellement une colonne « Clavier » (valeurs saisies).
3. Exécuter les lignes dans l'ordre réel d'exécution ; à chaque ligne, ne recopier que ce qui change. Une case vide ou « ? » signifie « indéterminé ».

Algorithme à simuler

```
Algorithme : MYSTERE
Variables
    a, b, c : entier
1 Début
2     a ← 5
3     b ← a + 3
4     c ← a * b
5     a ← c - b
6     b ← a mod 4
7     Écrire(a, " ", b, " ", c)
8 Fin
```

Ligne	a	b	c	Écran
début	?	?	?	
2	5	?	?	
3	5	8	?	
4	5	8	40	
5	32	8	40	
6	32	0	40	
7	32	0	40	`32 0 40`

À retenir

— Un algorithme = en-tête + déclarations + corps (Début…Fin).

— Toute variable doit être déclarée avec un type avant d'être utilisée.

— La simulation se fait ligne par ligne dans un tableau ; elle est exigée en évaluation : soyez

méthodiques.

1.6 Première approche de la complexité

Plusieurs algorithmes peuvent résoudre le même problème ; ils ne sont pas tous aussi *rapides*. La **complexité en temps** d'un algorithme mesure le *nombre d'opérations élémentaires* (affectation, comparaison, opération arithmétique, lecture, écriture) qu'il exécute, en fonction de la taille n des données.

Exemple : somme des entiers de 1 à n

— Méthode 1 : additionner un à un $1 + 2 + \cdots + n$: environ n additions.
— Méthode 2 : utiliser la formule $S = n(n+1)/2$: 3 opérations, *quel que soit* n.
Pour $n = 10^6$, la méthode 2 est environ 300 000 fois plus rapide.

Cette notion sera formalisée au chapitre 3 (boucles, notation O) et appliquée aux algorithmes sur les tableaux au chapitre 4.

CHAPITRE 2

Instructions élémentaires et expressions

Les instructions élémentaires sont les briques de base de tout algorithme : **l'affectation**, **la lecture** (entrée) et **l'écriture** (sortie). Elles manipulent des **expressions**.

2.1 L'affectation

Définition

L'**affectation** donne une valeur à une variable. Elle s'écrit

`variable` ← `expression`

et se lit « *variable reçoit expression* ». L'expression de droite est *d'abord évaluée*, puis sa valeur est rangée dans la variable de gauche ; l'ancienne valeur est *écrasée*.

Exemple

`x` ← `3` ; `y` ← `x + 2` ; `x` ← `x + 1` (incrémentation : la nouvelle valeur de `x` est l'ancienne plus 1).

Attention

- À gauche de « ← » il y a **toujours une seule variable** : `a + b` ← `5` et `5` ← `a` n'ont aucun sens.
- L'expression doit être **du même type** que la variable (un réel peut toutefois recevoir un entier).
- L'affectation n'est pas une égalité mathématique : `x` ← `x + 1` est correcte, alors que $x = x + 1$ est absurde en mathématiques.
- L'affectation n'est pas symétrique : `a` ← `b` et `b` ← `a` sont différentes.

2.1.1 Exemple classique : la permutation de deux variables

On veut échanger les contenus de `a` et `b`. La séquence `a` ← `b` ; `b` ← `a` est **fausse** : après la première instruction, l'ancienne valeur de `a` est perdue. Il faut une variable auxiliaire.

Permutation de deux entiers

```
Algorithme : Permutation
Variables
    a, b, tmp : entier
Début
    Écrire("Donner a et b : ")
    Lire(a, b)
    tmp ← a                                  /* on sauvegarde a */
    a ← b
    b ← tmp
    Écrire("a = ", a, " b = ", b)
Fin
```

Simulation avec la saisie `4` puis `9` :

Ligne	a	b	tmp	Écran
3	4	9	?	
4	4	9	4	
5	9	9	4	
6	9	4	4	
7	9	4	4	`a = 9 b = 4`

2.2 La lecture et l'écriture

2.2.1 La lecture : `Lire`

`Lire(v)` suspend l'exécution, attend que l'utilisateur tape une valeur au clavier, puis range cette valeur dans la variable `v`. On peut lire plusieurs variables : **`Lire(a, b, c)`**.

2.2.2 L'écriture : `Écrire`

`Écrire(liste)` affiche à l'écran les éléments de la liste, séparés par des virgules : textes entre guillemets (affichés tels quels), variables ou expressions (dont on affiche la *valeur*).

Exemple

Si `n` vaut 5 : **`Écrire("Le double de ", n, " est ", 2*n)`** affiche `Le double de 5 est 10`.

Remarque

Un bon algorithme **guide l'utilisateur** : chaque `Lire` est précédé d'un `Écrire` qui indique ce qu'il faut saisir, et les résultats sont affichés avec un libellé explicite.

2.3 Les expressions

Définition

Une **expression** est une combinaison d'opérandes (constantes, variables, appels de fonctions) et d'opérateurs, qui possède une **valeur** et un **type**.

2.3.1 Expressions arithmétiques

Opérateur	Signification	Exemple
`+ - *`	addition, soustraction, multiplication	`3 * 4` vaut 12
`/`	division réelle	`7 / 2` vaut 3.5
`div`	quotient de la division entière (entiers seulement)	`7 div 2` vaut 3
`mod`	reste de la division entière (entiers seulement)	`7 mod 2` vaut 1

Remarque : usages courants de `mod` et `div`

- `n mod 2 = 0` $\iff$ n est pair ;
- `n mod d = 0` $\iff$ d divise n ;
- `n mod 10` donne le chiffre des unités de n, `n div 10` le supprime ;
- conversion de secondes : `h` $\leftarrow$ `s div 3600`, `m` $\leftarrow$ `(s mod 3600) div 60`.

2.3.2 Expressions relationnelles (comparaisons)

Les opérateurs `=`, $\neq$, `<`, $\leq$, `>`, $\geq$ comparent deux valeurs du même type. Le résultat est un **booléen**. Exemple : si `x` vaut 4, `x > 3` vaut **Vrai** et `x = 5` vaut **Faux**.

2.3.3 Expressions logiques

Les opérateurs **ET**, **OU**, **NON** combinent des booléens.

A	B	A **ET** B	A **OU** B	**NON** A
Faux	Faux	Faux	Faux	Vrai
Faux	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Vrai	Faux
Vrai	Vrai	Vrai	Vrai	Faux

Exemple

« x appartient à l'intervalle $[0, 20]$ » s'écrit `(x ≥ 0) ET (x ≤ 20)` et **non** `0 ≤ x ≤ 20`.
Lois de De Morgan : `NON(A ET B) = (NON A) OU (NON B)` ; `NON(A OU B) = (NON A) ET (NON B)`.

2.3.4 Priorité des opérateurs

Du plus prioritaire au moins prioritaire :

Rang	Opérateurs
1	parenthèses `( )`
2	**NON**, moins unaire `-`
3	`*`, `/`, `div`, `mod`
4	`+`, `-`
5	`<`, $\leq$, `>`, $\geq$, `=`, $\neq$
6	**ET**
7	**OU**

À priorité égale, l'évaluation se fait de gauche à droite. En cas de doute, **mettre des parenthèses**.

Exemple : évaluation pas à pas

Avec `a = 3`, `b = 7`, `c = 2` :

$$\texttt{a + b mod c * 4} = 3 + ((7 \bmod 2) \times 4) = 3 + (1 \times 4) = 7$$

$$\texttt{(a < b) ET NON(b > 10) OU (c = 0)} = (\text{Vrai ET Vrai}) \text{ OU Faux} = \text{Vrai}$$

2.3.5 Fonctions prédéfinies usuelles

On admet l'existence des fonctions suivantes : `abs(x)` (valeur absolue), `racine(x)` (racine carrée), `puissance(x, n)` (x^n), `ent(x)` (partie entière), `longueur(ch)` (nombre de caractères d'une chaîne).

2.4 Un algorithme complet

Conversion d'une durée en heures, minutes, secondes

```
Algorithme : ConversionDuree
Variables
    total, h, m, s : entier
Début
    Écrire("Durée en secondes : ")
    Lire(total)
    h ← total div 3600
    m ← (total mod 3600) div 60
    s ← total mod 60
    Écrire(total, " s = ", h, " h ", m, " min ", s, " s")
Fin
```

Pour `total = 7384` : $h = 2$, $m = (7384 \bmod 3600) \operatorname{div} 60 = 184 \operatorname{div} 60 = 3$, $s = 4$; affichage `7384 s = 2 h 3 min 4 s`.

À retenir

- `v ← expr` : on évalue `expr`, puis on range le résultat dans `v`.
- Permuter deux variables exige une variable auxiliaire.
- `div` et `mod` ne s'appliquent qu'aux entiers ; `/` donne un réel.
- Les comparaisons et les opérateurs ET/OU/NON produisent des booléens.

CHAPITRE 3

Structures décisionnelles et boucles

Jusqu'ici, les instructions s'exécutaient toutes, une fois, dans l'ordre (structure **séquentielle**). Les **structures de contrôle** permettent de *choisir* les instructions à exécuter (décisions) ou de les *répéter* (boucles).

3.1 Les structures décisionnelles

3.1.1 L'alternative simple : Si...alors...FinSi

```
1 Si condition alors
2     instructions
3 FinSi
```

La *condition* est une expression booléenne. Si elle vaut **Vrai**, le bloc est exécuté ; sinon il est ignoré et l'exécution continue après **FinSi**.

3.1.2 L'alternative complète : Si...alors...Sinon...FinSi

Parité d'un entier

```
Algorithme : Parite
Variables
    n : entier
1 Début
2     Écrire("Donner un entier : ")
3     Lire(n)
4     Si n mod 2 = 0 alors
5         Écrire(n, " est pair")
6     Sinon
7         Écrire(n, " est impair")
8     FinSi
9 Fin
```

3.1.3 Alternatives imbriquées et SinonSi

Lorsque plus de deux cas sont possibles, on enchaîne les tests. La forme **SinonSi** évite des imbrications profondes.

Mention d'un étudiant à partir de sa moyenne

```
Algorithme : Mention
Variables
    moy : réel
1 Début
2     Lire(moy)
3     Si moy < 10 alors
4         Écrire("Ajourné")
5     SinonSi moy < 12 alors
6         Écrire("Passable")
7     SinonSi moy < 14 alors
8         Écrire("Assez bien")
9     SinonSi moy < 16 alors
```

```
        Écrire("Bien")
    Sinon
        Écrire("Très bien")
    FinSi
Fin
```

Remarque

Les conditions sont testées dans l'ordre : dès qu'une condition est vraie, son bloc est exécuté et les suivantes ne sont pas examinées. Ainsi, à la ligne du test `moy < 12`, on sait déjà que `moy` $\geq$ `10` : inutile de le retester.

3.1.4 Le choix multiple : Selon

Lorsque le choix porte sur les *valeurs* d'une variable entière ou caractère, la structure **Selon** est plus lisible.

Calculatrice élémentaire

```
Algorithme : Calculatrice
Variables
    a, b : réel
    op : caractère
Début
    Écrire("Opération (a op b) : ")
    Lire(a, op, b)
    Selon op faire
        '+' : Écrire(a + b)
        '-' : Écrire(a - b)
        '*' : Écrire(a * b)
        '/' : Si b ≠ 0 alors Écrire(a / b) Sinon Écrire("Division par zéro") FinSi
        Autrement : Écrire("Opérateur inconnu")
    FinSelon
Fin
```

3.2 Les structures répétitives (boucles)

Une **boucle** répète un bloc d'instructions. Chaque passage est une **itération**. Il existe trois formes.

3.2.1 La boucle Pour

On l'utilise lorsque le **nombre d'itérations est connu** à l'avance.

```
Pour i ← valeur_initiale à valeur_finale faire
    instructions
FinPour
```

Le **compteur** `i` (entier) prend successivement les valeurs `valeur_initiale`, `valeur_initiale + 1`, …, `valeur_finale`. Il est géré automatiquement : on ne doit **pas** le modifier dans la boucle. On peut préciser un pas : **Pour** i $\leftarrow$ 10 **à** 0 **pas** -2 **faire**.

Somme des n premiers entiers

```
Algorithme : SommeN
Variables
    n, i, s : entier
1  Début
2      Lire(n)
3      s ← 0                                   /* initialisation de l'accumulateur */
4      Pour i ← 1 à n faire
5          s ← s + i
6      FinPour
7      Écrire("Somme = ", s)
8  Fin
```

Simulation pour $n = 4$:

Ligne	n	i	s	Écran
2	4	?	?	
3	4	?	0	
4–5	4	1	1	
4–5	4	2	3	
4–5	4	3	6	
4–5	4	4	10	
7	4	–	10	`Somme = 10`

3.2.2 La boucle TantQue

On l'utilise lorsque le nombre d'itérations est **inconnu** et que la boucle peut ne s'exécuter **aucune fois**. La condition est testée *avant* chaque itération.

```
1  TantQue condition faire
2      instructions
3  FinTantQue
```

Attention

Le bloc doit **modifier** au moins une variable de la condition, sinon la boucle ne s'arrête jamais (*boucle infinie*).

PGCD par la méthode d'Euclide

```
Algorithme : PGCD
Variables
    a, b, r : entier
1  Début
2      Lire(a, b)                              /* a > 0 et b > 0 */
3      TantQue b ≠ 0 faire
4          r ← a mod b
5          a ← b
6          b ← r
7      FinTantQue
8      Écrire("PGCD = ", a)
9  Fin
```

3.2.3 La boucle Répéter…Jusqu'à

La condition est testée *après* chaque itération : le bloc s'exécute **au moins une fois**. La boucle s'arrête quand la condition devient **vraie**. Elle est idéale pour la **saisie contrôlée**.

Saisie contrôlée d'une note entre 0 et 20

Algorithme : SaisieNote
Variables
note : réel
Début
  Répéter
    Écrire("Note (0 à 20) : ")
    Lire(note)
  Jusqu'à (note ≥ 0) **ET** (note ≤ 20)
  Écrire("Note acceptée : ", note)
Fin

3.2.4 Choisir la bonne boucle

Boucle	Quand l'utiliser ?	Nombre d'itérations
Pour	nombre d'itérations connu avant d'entrer	exactement $(fin - début + 1)$
TantQue	arrêt sur une condition, peut-être aucune itération	≥ 0
Répéter	arrêt sur une condition, au moins une itération	≥ 1

Remarque

Toute boucle **Pour** peut s'écrire avec un **TantQue** : `i ← 1 ; TantQue i ≤ n faire ... i ← i + 1 FinTantQue`. L'inverse n'est pas vrai.

3.2.5 Schémas classiques

— **Compteur** : `cpt ← 0` avant la boucle, `cpt ← cpt + 1` dedans ;
— **Accumulateur** (somme) : `s ← 0` avant, `s ← s + x` dedans ;
— **Produit** : `p ← 1` avant, `p ← p * x` dedans (factorielle, puissance) ;
— **Maximum** : `max ← premier élément`, puis `Si x > max alors max ← x` ;
— **Valeur sentinelle** : on lit des valeurs jusqu'à une valeur particulière (par exemple -1) qui marque la fin de la saisie.

Moyenne d'une suite de notes terminée par -1 (sentinelle)

Algorithme : MoyenneSentinelle
Variables
x, s : réel
n : entier
Début
  s ← 0
  n ← 0
  Lire(x)
  TantQue x ≠ -1 **faire**
    s ← s + x
    n ← n + 1
    Lire(x)
  FinTantQue
  Si n > 0 **alors**
    Écrire("Moyenne = ", s / n)

```
12      Sinon
13          Écrire("Aucune note saisie")
14      FinSi
15  Fin
```

3.2.6 Boucles imbriquées

Table de multiplication de 1 à 9

```
    Algorithme : TablesMultiplication
    Variables
        i, j : entier
1   Début
2       Pour i ← 1 à 9 faire
3           Pour j ← 1 à 9 faire
4               Écrire(i, " x ", j, " = ", i * j)
5           FinPour
6       FinPour
7   Fin
```

L'instruction d'écriture est exécutée $9 \times 9 = 81$ fois.

3.3 Complexité des algorithmes

3.3.1 Compter les opérations

Pour évaluer la complexité, on compte les opérations élémentaires en fonction de la taille n des données. On suppose que chaque opération élémentaire coûte une unité de temps.

- Séquence : on **additionne** les coûts.
- Alternative : on prend le coût **du bloc le plus coûteux** (pire des cas).
- Boucle : coût du corps **multiplié** par le nombre d'itérations.

Exemple : algorithme SommeN

Hors de la boucle : `Lire`, `s` ← `0`, `Écrire` : 3 opérations. Dans la boucle : 1 addition + 1 affectation, plus la gestion du compteur (1 comparaison + 1 incrémentation) : 4 opérations, n fois. Total : $T(n) = 4n + 3$.

3.3.2 La notation O (grand O)

Définition

On dit que $T(n)$ est en $O(f(n))$ si, pour n assez grand, $T(n) \leq c \cdot f(n)$ pour une constante $c > 0$. Concrètement : on garde le **terme dominant** et on supprime les constantes multiplicatives.

Exemples : $4n + 3 = O(n)$; $3n^2 + 10n + 7 = O(n^2)$; $25 = O(1)$.

Notation	Nom	Exemple d'algorithme	$n = 1000$
$O(1)$	constante	formule $n(n+1)/2$, accès à `T[i]`	1
$O(\log n)$	logarithmique	recherche dichotomique, PGCD d'Euclide	≈ 10
$O(n)$	linéaire	somme, maximum, recherche séquentielle	10^3
$O(n \log n)$	quasi-linéaire	tris rapides (tri fusion)	$\approx 10^4$
$O(n^2)$	quadratique	deux boucles imbriquées, tri par sélection	10^6
$O(2^n)$	exponentielle	énumération de tous les sous-ensembles	$\approx 10^{301}$

À retenir

- Une boucle simple de n itérations à corps constant : $O(n)$.
- Deux boucles imbriquées de n itérations chacune : $O(n^2)$.
- Une boucle qui divise le problème par 2 à chaque itération : $O(\log n)$.
- On s'intéresse en général au **pire des cas**.

CHAPITRE 4

Variables structurées : tableaux et structures

Les types de base ne permettent de ranger qu'*une* valeur par variable. Pour mémoriser les 60 notes d'une classe, il serait absurde de déclarer 60 variables. Les **variables structurées** regroupent plusieurs valeurs sous un même nom.

4.1 Les tableaux à une dimension

Définition

Un **tableau** est une suite de cases **de même type**, désignée par un nom unique. Chaque case est repérée par un **indice** entier. Sa taille est fixée à la déclaration.

4.1.1 Déclaration

Déclaration d'un tableau

Constantes
    NMAX : = 60
Types
    t_notes : = tableau[1..NMAX] de réel
Variables
    notes : t_notes
    T : tableau[1..10] d'entier *(* déclaration directe */*

indices	1	2	3	4	5	6	7	8	9	10
T	12	5	-3	8	20	7	0	14	9	11

4.1.2 Accès aux éléments

`T[i]` désigne la case d'indice `i` ; elle s'utilise exactement comme une variable simple : `T[3]` ← `7`, `Lire(T[i])`, `s ← s + T[i]`. L'indice peut être une expression : `T[i+1]`, `T[n-i+1]`.

Attention

L'indice doit toujours rester dans les bornes déclarées (ici $1 \leq i \leq 10$). Accéder à `T[11]` est une erreur grave (débordement de tableau).

4.1.3 Parcours : saisie et affichage

Le parcours complet d'un tableau se fait naturellement avec une boucle **Pour**.

Saisie de n valeurs, puis affichage

Algorithme : SaisieAffichage
Constantes
    NMAX : = 100
Variables
    T : tableau[1..NMAX] d'entier
    n, i : entier
1 **Début**

```
    Répéter
        Écrire("Nombre d'éléments (1 à ", NMAX, ") : ")
        Lire(n)
    Jusqu'à (n ≥ 1) ET (n ≤ NMAX)
    Pour i ← 1 à n faire
        Écrire("T[", i, "] = ")
        Lire(T[i])
    FinPour
    Pour i ← 1 à n faire
        Écrire(T[i], " ")
    FinPour
Fin
```

Remarque

On déclare souvent un tableau de taille maximale `NMAX` et on n'en utilise que les `n` premières cases (*taille utile*).

4.2 Algorithmes classiques sur les tableaux

Dans cette section, `T` est un tableau d'entiers dont les `n` premières cases sont remplies.

4.2.1 Somme, moyenne, maximum

Somme, moyenne et maximum

```
s ← 0
max ← T[1]
Pour i ← 1 à n faire
    s ← s + T[i]
    Si T[i] > max alors
        max ← T[i]
    FinSi
FinPour
Écrire("Somme = ", s, " Moyenne = ", s / n, " Max = ", max)
```

Complexité : une boucle de n itérations, soit $O(n)$.

4.2.2 Recherche séquentielle

On cherche si une valeur `x` est présente dans `T`, et à quelle position.

Recherche séquentielle

```
i ← 1
TantQue (i ≤ n) ET (T[i] ≠ x) faire
    i ← i + 1
FinTantQue
Si i ≤ n alors
    Écrire(x, " trouvé à la position ", i)
Sinon
    Écrire(x, " absent du tableau")
FinSi
```

On s'arrête dès que l'élément est trouvé. Pire des cas (élément absent) : n comparaisons, soit $O(n)$.

4.2.3 Recherche dichotomique (tableau trié)

Si le tableau est **trié par ordre croissant**, on compare `x` à l'élément du milieu : on élimine à chaque étape la moitié de la zone de recherche.

Recherche dichotomique

```
g ← 1
d ← n
trouve ← Faux
TantQue (g ≤ d) ET (NON trouve) faire
    m ← (g + d) div 2
    Si T[m] = x alors
        trouve ← Vrai
    SinonSi T[m] < x alors
        g ← m + 1
    Sinon
        d ← m - 1
    FinSi
FinTantQue
```

Exemple : recherche de $x = 23$

`T = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]`, $n = 10$.

Étape	g	d	m	T[m]
1	1	10	5	$16 < 23 \rightarrow$ g $\leftarrow 6$
2	6	10	8	$56 > 23 \rightarrow$ d $\leftarrow 7$
3	6	7	6	$23 = 23 \rightarrow$ trouvé

3 comparaisons au lieu de 6 en recherche séquentielle.

Après k étapes, la zone contient au plus $n/2^k$ éléments ; on s'arrête au plus quand $n/2^k < 1$, soit $k \approx \log_2 n$. Complexité : $O(\log n)$. Pour $n = 10^6$: 20 comparaisons au plus, contre 10^6 en recherche séquentielle.

4.2.4 Tri par sélection

Principe : pour chaque position `i` de 1 à $n-1$, on cherche le plus petit élément de la partie `T[i..n]` et on l'échange avec `T[i]`.

Tri par sélection (ordre croissant)

```
Pour i ← 1 à n - 1 faire
    posMin ← i
    Pour j ← i + 1 à n faire
        Si T[j] < T[posMin] alors
            posMin ← j
        FinSi
    FinPour
    Si posMin ≠ i alors
        tmp ← T[i]
        T[i] ← T[posMin]
```

```
11          T[posMin] ← tmp
12      FinSi
13  FinPour
```

Exemple : déroulement sur `[7, 3, 9, 1, 5]`

i	Tableau après l'itération	Échange
1	`[1, 3, 9, 7, 5]`	T[1] ↔ T[4]
2	`[1, 3, 9, 7, 5]`	aucun
3	`[1, 3, 5, 7, 9]`	T[3] ↔ T[5]
4	`[1, 3, 5, 7, 9]`	aucun

Nombre de comparaisons : $(n-1) + (n-2) + \cdots + 1 = \dfrac{n(n-1)}{2}$, soit $O(n^2)$.

4.2.5 Tri à bulles

Principe : on parcourt le tableau en comparant les éléments voisins et on les échange s'ils sont mal ordonnés ; après chaque passage, le plus grand élément restant est « remonté » à la fin. On s'arrête quand un passage n'a fait aucun échange.

Tri à bulles optimisé

```
1  k ← n
2  Répéter
3      echange ← Faux
4      Pour i ← 1 à k - 1 faire
5          Si T[i] > T[i + 1] alors
6              tmp ← T[i]
7              T[i] ← T[i + 1]
8              T[i + 1] ← tmp
9              echange ← Vrai
10         FinSi
11     FinPour
12     k ← k - 1
13 Jusqu'à (NON echange) OU (k = 1)
```

Pire des cas (tableau trié à l'envers) : $O(n^2)$. Meilleur cas (déjà trié) : un seul passage, $O(n)$.

4.3 Les tableaux à deux dimensions (matrices)

Déclaration et parcours d'une matrice

```
   Variables
       M : tableau[1..3, 1..4] de réel        /* 3 lignes, 4 colonnes */
       i, j : entier
1  Début
2      Pour i ← 1 à 3 faire
3          Pour j ← 1 à 4 faire
4              Lire(M[i, j])
5          FinPour
6      FinPour
7  Fin
```

`M[i, j]` désigne l'élément de la ligne `i` et de la colonne `j`. Le parcours d'une matrice $n \times m$ coûte $O(n \times m)$.

Exemple : somme de deux matrices

Pour i ← 1 **à** n **faire Pour** j ← 1 **à** m **faire** `C[i, j] ← A[i, j] + B[i, j]` **FinPour FinPour**.

4.4 Les structures (enregistrements)

Définition

Une **structure** (ou *enregistrement*) regroupe sous un même nom des données **de types éventuellement différents**, appelées **champs**.

Déclaration d'une structure

```
Types
    Etudiant = Structure
        matricule : chaîne
        nom : chaîne
        age : entier
        moyenne : réel
    FinStructure
Variables
    e1, e2 : Etudiant
Début
    e1.nom ← "Ngo Bassa"
    e1.age ← 21
    Lire(e1.moyenne)
    e2 ← e1                                   /* copie de tous les champs */
    Écrire(e2.nom, " : ", e2.moyenne)
Fin
```

On accède à un champ par la **notation pointée** : `variable.champ`.

4.4.1 Tableaux de structures

On combine les deux notions pour gérer, par exemple, une classe entière.

Meilleur étudiant d'une classe

```
Types
    t_classe : = tableau[1..60] d'Etudiant
Variables
    classe : t_classe
    n, i, iMax : entier
Début
    /* on suppose classe[1..n] déjà rempli */
    iMax ← 1
    Pour i ← 2 à n faire
        Si classe[i].moyenne > classe[iMax].moyenne alors
            iMax ← i
        FinSi
    FinPour
```

```
8     Écrire("Major : ", classe[iMax].nom, " (", classe[iMax].moyenne, ")")
9 Fin
```

À retenir

- Tableau : éléments **de même type**, accès par indice `T[i]`.
- Structure : champs **de types différents**, accès par `v.champ`.
- Recherche séquentielle $O(n)$; dichotomique $O(\log n)$ (tableau trié) ; tris par sélection et à bulles $O(n^2)$.

CHAPITRE 5

Fonctions et procédures

5.1 Pourquoi des sous-programmes ?

Lorsqu'un algorithme grandit, certaines suites d'instructions se répètent (saisie contrôlée, calcul d'un maximum…) et l'ensemble devient illisible. Un **sous-programme** est un bloc d'instructions nommé, écrit une seule fois et **appelé** autant de fois que nécessaire. On distingue :

— la **fonction**, qui calcule et **retourne une valeur** ;

— la **procédure**, qui réalise une action et ne retourne pas de valeur (elle peut toutefois modifier ses paramètres).

Avantages : **lisibilité** (programmation modulaire, « diviser pour régner »), **réutilisation**, **mise au point** plus facile (chaque module se teste séparément).

5.2 Les fonctions

5.2.1 Définition

Syntaxe d'une fonction

```
1 Fonction NomFonction(param1 : type1, param2 : type2) : type_retour
      Variables locales
2     Début
3         instructions
4         Retourner expression
5     Fin
6 FinFonction
```

Fonction calculant le prix TTC

```
1 Fonction PrixTTC(prixHT : réel, tva : réel) : réel
2     Début
3         Retourner prixHT * (1 + tva)
4     Fin
5 FinFonction
```

— Les **paramètres formels** (`prixHT`, `tva`) sont les noms utilisés dans la définition.

— L'instruction **Retourner** termine la fonction et renvoie la valeur au programme appelant.

— Une fonction s'appelle **dans une expression**, comme une fonction mathématique.

5.2.2 Appel d'une fonction

Programme principal utilisant PrixTTC

```
    Algorithme : Facture
    Constantes
        TVA : = 0.1925
    Variables
        ht, ttc : réel
1   Début
2       Écrire("Prix HT : ")
```

```
    Lire(ht)
    ttc ← PrixTTC(ht, TVA)
    Écrire("Prix TTC : ", ttc)
Fin
```

Lors de l'appel `PrixTTC(ht, TVA)`, les **paramètres effectifs** `ht` et `TVA` sont évalués, puis leurs valeurs sont **copiées** dans les paramètres formels `prixHT` et `tva`. Le nombre, l'ordre et les types des paramètres effectifs doivent correspondre à ceux des paramètres formels.

Fonction booléenne : un entier est-il premier ?

```
Fonction EstPremier(n : entier) : booléen
    Variables d : entier
    Début
        Si n < 2 alors
            Retourner Faux
        FinSi
        d ← 2
        TantQue d * d ≤ n faire
            Si n mod d = 0 alors
                Retourner Faux
            FinSi
            d ← d + 1
        FinTantQue
        Retourner Vrai
    Fin
FinFonction
```

5.3 Les procédures

Procédure affichant une ligne de n caractères

```
Procédure Ligne(n : entier, c : caractère)
    Variables i : entier
    Début
        Pour i ← 1 à n faire
            Écrire(c)
        FinPour
        Écrire("\n")                          /* retour à la ligne */
    Fin
FinProcédure
```

Une procédure s'appelle comme une **instruction** : `Ligne(20, '*')`.

	Fonction	**Procédure**
Rôle	calculer *une* valeur	réaliser une action
Retour	**Retourner** obligatoire	pas de valeur de retour
Appel	dans une expression : `x ← f(a)`	comme une instruction : `p(a)`

5.4 Variables locales et globales

— Une variable **locale** est déclarée dans un sous-programme ; elle n'existe que pendant son exécution et n'est visible que de lui.

— Une variable **globale** est déclarée dans l'algorithme principal (partie *Variables globales*) ; elle est visible de tous les sous-programmes.

Attention

L'usage des variables globales doit rester exceptionnel : un sous-programme qui les modifie crée des *effets de bord* difficiles à suivre. On préfère communiquer par les paramètres. Si une variable locale porte le même nom qu'une variable globale, c'est la locale qui est utilisée dans le sous-programme (elle *masque* la globale).

5.5 Modes de passage des paramètres

5.5.1 Passage par valeur (paramètre d'entrée, E)

Le sous-programme reçoit une **copie** de la valeur. Les modifications faites sur le paramètre formel **n'affectent pas** la variable de l'appelant. C'est le mode par défaut.

5.5.2 Passage par référence (paramètre de sortie S ou d'entrée/sortie E/S)

Le sous-programme travaille **directement sur la variable** de l'appelant (il en reçoit l'adresse). Toute modification est visible après l'appel. On le signale par le mot-clé **var** (ou par les mentions E, S, E/S). Le paramètre effectif doit alors obligatoirement être une **variable**.

Procédure d'échange : passage par référence

```
1  Procédure Echanger(var a : entier, var b : entier)
       Variables tmp : entier
2      Début
3          tmp ← a
4          a ← b
5          b ← tmp
6      Fin
7  FinProcédure
```

Exemple : comparaison des deux modes

Avec `x ← 3 ; y ← 8 ; Echanger(x, y)`, on obtient `x = 8, y = 3`. Si l'on retire le mot **var**, `Echanger` permute des *copies* : après l'appel, `x = 3` et `y = 8` sont inchangés.



Procédure à plusieurs résultats : minimum et maximum d'un tableau

```
1  Procédure MinMax(T : t_tab, n : entier, var mini : entier, var maxi : entier)
       Variables i : entier
2      Début
3          mini ← T[1]
4          maxi ← T[1]
5          Pour i ← 2 à n faire
6              Si T[i] < mini alors
7                  mini ← T[i]
8              FinSi
9              Si T[i] > maxi alors
10                 maxi ← T[i]
```

```
            FinSi
        FinPour
    Fin
FinProcédure
```

Remarque

Une fonction ne retourne qu'*une* valeur. Pour produire plusieurs résultats, on utilise une procédure avec plusieurs paramètres passés par référence.

5.6 Simulation d'un appel

Lors de la simulation d'un algorithme comportant des sous-programmes, on ouvre un **nouveau tableau** (ou un nouveau bloc de colonnes) pour chaque appel : ses variables locales et ses paramètres naissent à l'appel et disparaissent au retour.

5.7 La récursivité

Définition

Un sous-programme est **récursif** s'il s'appelle lui-même. Il doit comporter une **condition d'arrêt** (cas de base) atteinte après un nombre fini d'appels.

Factorielle récursive

```
Fonction FACT(n : entier) : entier
    Début
        Si n ≤ 1 alors
            Retourner 1                          /* cas de base */
        Sinon
            Retourner n * FACT(n - 1)
        FinSi
    Fin
FinFonction
```

Déroulement de `Fact(4)` : `Fact(4)` $= 4 \times$ `Fact(3)` $= 4 \times 3 \times$ `Fact(2)` $= 4 \times 3 \times 2 \times$ `Fact(1)` $= 4 \times 3 \times 2 \times 1 = 24$. Le nombre d'appels est n : complexité $O(n)$.

À retenir

- Fonction : retourne une valeur, appelée dans une expression.
- Procédure : effectue une action, appelée comme une instruction.
- Passage par valeur = copie (entrée) ; par référence (**var**) = la variable elle-même (sortie ou entrée/sortie).
- Toute récursivité exige un cas de base.

CHAPITRE 6

Les pointeurs

6.1 Adresse et pointeur

Chaque variable occupe un emplacement en mémoire repéré par son **adresse** (un numéro de case mémoire). Jusqu'ici, on accédait aux variables par leur nom ; on peut aussi y accéder *par leur adresse*.

Définition

Un **pointeur** est une variable dont la valeur est l'**adresse** d'une autre variable. On dit qu'il « pointe sur » cette variable.



6.2 Notations

Notation	Signification	En C
`p : ↑entier`	déclaration : `p` est un pointeur sur entier	`int *p;`
`&x`	adresse de la variable `x`	`&x`
`p↑`	variable pointée par `p` (contenu à l'adresse `p`)	`*p`
`NIL`	pointeur qui ne pointe sur rien	`NULL`
`Allouer(p)`	réserve dynamiquement une zone et range son adresse dans `p`	`p = malloc(...);`
`Libérer(p)`	rend au système la zone pointée par `p`	`free(p);`

Manipulation élémentaire d'un pointeur

Algorithme : DemoPointeur
Variables
x, y : entier
p : ↑entier
1 **Début**
2 x ← 17
3 p ← &x *// p pointe sur x*
4 p↑ ← p↑ + 3 *// modifie x à travers p*
5 y ← p↑ * 2
6 **Écrire**(x, " ", y) *// affiche 20 40*
7 **Fin**

Ligne	x	y	p	p↑
2	17	?	?	–
3	17	?	&x	17
4	20	?	&x	20
5	20	40	&x	20

Attention

— Ne jamais utiliser `p↑` si `p` n'a pas été initialisé ou vaut `NIL`.
— `p` (une adresse) et `p↑` (une valeur) sont de types différents : `p ← 5` est une erreur.

6.3 Pointeurs et passage de paramètres

Le passage par référence du chapitre 5 est en réalité un passage d'**adresse**. En langage C, qui ne connaît que le passage par valeur, on le réalise explicitement avec des pointeurs :

Échange avec des pointeurs

```
1 Procédure Echanger(pa : ↑entier, pb : ↑entier)
      Variables tmp : entier
2     Début
3         tmp ← pa↑
4         pa↑ ← pb↑
5         pb↑ ← tmp
6     Fin
7 FinProcédure
```

Appel : `Echanger(&x, &y)`.

```c
void echanger(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
/* appel : echanger(&x, &y); */
```

6.4 Allocation dynamique

Les variables déclarées dans la partie *Variables* sont **statiques** : leur taille est fixée avant l'exécution. Grâce aux pointeurs, on peut réserver de la mémoire **pendant** l'exécution, selon les besoins.

Tableau de taille choisie à l'exécution

```
    Algorithme : TableauDynamique
    Variables
        T : ↑entier
        n, i : entier
1   Début
2       Lire(n)
3       Allouer(T, n)                    /* réserve n entiers consécutifs */
4       Si T = NIL alors
5           Écrire("Mémoire insuffisante")
6       Sinon
7           Pour i ← 1 à n faire
8               Lire(T[i])
9           FinPour
        /* ... traitements ... */
10          Libérer(T)
11      FinSi
12  Fin
```

```c
int n, i;
int *T;
scanf("%d", &n);
T = (int *) malloc(n * sizeof(int));
if (T == NULL) printf("Mémoire insuffisante\n");
else {
    for (i = 0; i < n; i++) scanf("%d", &T[i]); /* indices 0..n-1 en C */
    free(T);
}
```

Remarque

Toute zone allouée doit être libérée quand elle n'est plus utile ; sinon la mémoire « fuit ». Après `Libérer(p)`, il est prudent d'écrire `p` ← `NIL`.

6.5 Pointeurs et tableaux

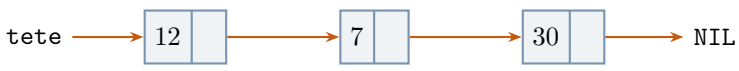
Le nom d'un tableau est l'adresse de son premier élément. En C, `T[i]` équivaut à `*(T + i)` : ajouter `i` à un pointeur le déplace de `i` *éléments* (et non de `i` octets).

6.6 Ouverture : les listes chaînées

Un pointeur peut être un champ d'une structure. On construit ainsi des **listes chaînées**, dont chaque élément (maillon) contient une valeur et l'adresse du suivant. Contrairement au tableau, la liste peut grandir ou rétrécir à volonté.

Déclaration d'un maillon et parcours d'une liste

```
Types
    Maillon = Structure
        valeur : entier
        suivant : ↑Maillon
    FinStructure
Variables
    tete, p : ↑Maillon
Début
    p ← tete
    TantQue p ≠ NIL faire
        Écrire(p↑.valeur)
        p ← p↑.suivant
    FinTantQue
Fin
```



Les listes, piles et files feront l'objet d'un approfondissement en structures de données.

À retenir

- Un pointeur contient une adresse ; `&x` donne l'adresse de `x`, `p↑` la valeur pointée.
- Le passage par référence repose sur les adresses.
- `Allouer` / `Libérer` permettent de gérer la mémoire pendant l'exécution.

ANNEXE A

Correspondance pseudo-code / langage C

Pseudo-code	Langage C
Algorithme : Nom … **Début** … **Fin**	`int main(void) { ... return 0; }`
Constantes PI = 3.14159	`#define PI 3.14159` ou `const double PI = 3.14159;`
x : entier / réel / caractère	`int x;` / `double x;` / `char x;`
b : booléen	`int b;` (0 = faux) ou `bool b;` avec `<stdbool.h>`
x ← 5	`x = 5;`
Lire(x)	`scanf("%d", &x);` (`%lf` réel, `%c` caractère)
Écrire("x = ", x)	`printf("x = %d\n", x);`
= ≠ ≤ ≥	`==` `!=` `<=` `>=`
ET **OU** **NON**	`&&` `||` `!`
a **div** b a **mod** b	`a / b` (entre entiers) `a % b`
Si c **alors** … **Sinon** … **FinSi**	`if (c) { ... } else { ... }`
Selon x **faire** 1 : … **Autrement** : …	`switch (x) { case 1: ... break; default: ... }`
Pour i ← 1 **à** n **faire**	`for (i = 1; i <= n; i++) { ... }`
TantQue c **faire**	`while (c) { ... }`
Répéter … **Jusqu'à** c	`do { ... } while (!(c));`
T : tableau[1..10] d'entier	`int T[10];` (indices **0** à **9**)
M : tableau[1..3, 1..4] de réel	`double M[3][4];`
Structure / champ `e.nom`	`struct Etudiant { ...};` / `e.nom`
Fonction F(a : entier) : réel	`double F(int a) { ... return ...; }`
Procédure P(a : entier)	`void P(int a) { ... }`
var a : entier (référence)	`int *a` et `*a` dans le corps ; appel `P(&x)`
p : ↑entier ; p↑ ; NIL	`int *p;` `*p` `NULL`
Allouer(p) ; **Libérer**(p)	`p = malloc(sizeof(int));` `free(p);`

Attention

En C : les tableaux commencent à l'indice **0** ; la division `/` entre deux entiers est une division entière (`7/2` vaut 3) ; l'opérateur d'égalité est `==` (et non `=`, qui est l'affectation).