# Algorithmique 2

## *Les Fonctions et les Procédures*

Tarek Boutefara

2026-02-07

# Table of Contents

# Chapitre 1. Introduction

Jusqu'à présent, nous avons abordé l'algorithmique comme une suite linéaire d'instructions, un peu comme une recette de cuisine où chaque étape s'enchaîne jusqu'au résultat final. Mais imaginez un instant que vous deviez construire une voiture : vous ne la fabriqueriez pas d'un seul bloc. Vous assembleriez des éléments préexistants (moteur, roues, électronique) qui ont été conçus, testés et réparés séparément. En programmation, cette approche s'appelle la modularité. Elle permet de transformer un problème complexe et intimidant en une série de petits problèmes simples, rapides à résoudre et surtout, faciles à corriger.

Écrire un programme de mille lignes dans un seul bloc (ce qu'on appelle souvent le "code spaghetti") est la garantie d'un cauchemar au moindre bogue. En utilisant les fonctions et les procédures, vous apprenez à créer vos propres "outils" personnalisés. Une fois qu'une procédure de tri ou une fonction de calcul de TVA est écrite, vous n'avez plus jamais besoin de la réécrire ; vous l'appelez simplement par son nom. C'est le passage de l'artisanat au travail d'ingénieur : on ne réinvente pas la roue à chaque projet, on assemble des composants fiables.

Ce chapitre va vous apprendre à structurer votre pensée. Nous allons découvrir comment isoler des tâches spécifiques, comment faire circuler l'information entre les différents modules et comment protéger vos données. C'est cette discipline qui permet aujourd'hui à des milliers de développeurs de collaborer sur un même logiciel sans se marcher sur les pieds, en s'appuyant sur le travail des autres pour construire des systèmes toujours plus ambitieux.

# Chapitre 2. La Modularité et le principe "Diviser pour Régner"

En informatique, le principe "Diviser pour Régner" (en anglais Divide and Conquer) consiste à décomposer un problème complexe en sous-problèmes plus simples, jusqu'à ce que chaque morceau soit facile à résoudre de manière isolée.

## 2.1. L'Analyse Descendante

C'est la méthode de travail standard de l'ingénieur. Au lieu d'écrire tout le code d'un coup, on commence par définir les grandes étapes de l'algorithme, puis on "zoome" sur chaque étape pour en définir les détails.

**Exemple :** Création d'un Bulletin de Notes

Si on demande à un étudiant de créer un logiciel de bulletin, il peut se sentir dépassé. Mais en appliquant la modularité, le problème devient :

- Module Saisie : Récupérer les notes des étudiants.
- Module Calcul : Calculer les moyennes et les rangs.
- Module Affichage : Mettre en forme et imprimer le bulletin.

Chaque module peut ensuite être confié à un développeur différent ou testé indépendamment.

## 2.2. Les avantages de l'approche modulaire

L'adoption d'une structure modulaire repose sur trois piliers fondamentaux qui garantissent la qualité d'un logiciel :

### 2.2.1. La réutilisabilité :

Un module conçu pour accomplir une tâche précise peut être exploité dans plusieurs parties d'un même projet ou même transféré vers d'autres applications. Cette capacité évite la duplication du code et permet de capitaliser sur des solutions déjà testées et validées.

### 2.2.2. La lisibilité :

En déléguant les détails techniques à des sous-programmes, le corps du programme principal s'allège. Il se transforme en une structure claire qui expose la logique globale de l'algorithme. Cette clarté facilite la compréhension du flux de données pour n'importe quel relecteur.

### 2.2.3. La maintenabilité :

La segmentation du code permet d'isoler les erreurs potentielles. Lorsqu'une anomalie est détectée ou qu'une évolution est nécessaire, l'intervention se limite au module concerné sans perturber l'ensemble du système. Cela réduit considérablement les risques d'effets de bord lors des phases de

mise à jour.

## 2.3. Observation de la structure d'un programme principal

L'utilisation de modules transforme la structure du programme principal en un résumé de haut niveau. Dans cet exemple de gestion de pharmacie, la séquence d'exécution devient explicite :

```
Début
    InitialiserStock()
    AfficherMenu()
    Si (Choix = "Vente") Alors
        GererTransaction()
    Sinon
        MettreAJourInventaire()
    FinSi
    SauvegarderDonnees()
Fin
```

L'architecture modulaire permet de saisir l'enchaînement des opérations en consultant uniquement le bloc principal. La complexité interne de chaque tâche (comme le détail des calculs financiers) est masquée, ce qui offre une vision d'ensemble sur les fonctionnalités du logiciel. Cette abstraction est essentielle pour gérer des projets dont la taille rendrait une écriture linéaire impossible à maîtriser.

# Chapitre 3. Les Procédures

La procédure est l'outil de base de la modularité. On peut la comparer à une "commande" personnalisée que l'on ajoute au langage de programmation pour effectuer une action précise.

## 3.1. Définition

Une procédure est un sous-programme qui regroupe un ensemble d'instructions sous un nom unique. Elle est conçue pour réaliser un traitement (comme un affichage, une modification de données ou un calcul complexe) sans renvoyer de valeur directe au programme appelant.

Contrairement à une fonction qui est traitée comme une valeur (ex: y=f(x)), une procédure est traitée comme une instruction à part entière.

## 3.2. Syntaxe et Déclaration

La mise en œuvre d'une procédure se déroule en deux phases : sa définition (avant le programme principal) et son appel (à l'intérieur du programme).

### 3.2.1. Structure de la déclaration

Une procédure possède la même architecture qu'un programme standard, avec sa propre zone de déclaration.

```
Procédure Nom_Procedure ( Paramètres_Optionnels )
Variables
    { Déclaration des variables locales }
Début
    { Corps de la procédure : suite d'instructions }
FinProcédure
```

- L'En-tête : Définit le nom de la procédure et les données dont elle a besoin pour fonctionner.
- Les Variables Locales : Ce sont des variables qui n'existent qu'à l'intérieur de la procédure. Elles sont créées au moment de l'appel et détruites à la fin de l'exécution.
- Le Corps : Contient la logique algorithmique.

### 3.2.2. L'Appel depuis le programme principal

Pour exécuter le code contenu dans une procédure, il suffit d'écrire son nom suivi de parenthèses (contenant les arguments si nécessaire).

## 3.3. Exemple : Affichage d'un menu

C'est l'utilisation la plus fréquente pour clarifier le programme principal.

```
Procédure AfficherMenu()
Début
    Ecrire("----- GESTION DE STOCK -----")
    Ecrire("1. Ajouter un produit")
    Ecrire("2. Supprimer un produit")
    Ecrire("3. Quitter")
    Ecrire("---------------------------")
FinProcédure
```

# 3.4. Les Paramètres et le Passage de Données

Les paramètres sont des canaux de communication qui permettent d'échanger des informations entre le programme appelant (le programme principal) et le sous-programme (la procédure ou la fonction).

## 3.4.1. Paramètres Formels vs Paramètres Effectifs

Il est essentiel de distinguer deux moments dans la vie d'un paramètre :

- Les Paramètres Formels : Ce sont les variables déclarées dans l'en-tête de la procédure. Ils servent de "boîtes de réception" vides qui définissent le type de données attendues.
- Les Paramètres Effectifs (ou Arguments) : Ce sont les valeurs réelles transmises lors de l'appel de la procédure.

Le paramètre formel est comme un formulaire administratif avec des cases vides (Nom, Prénom). Le paramètre effectif est la donnée qu'une personne écrive réellement dans ces cases.

## 3.4.2. Le Passage par Valeur

C'est le mode par défaut. Dans ce cas, le programme principal envoie une copie de la valeur à la procédure.

- Mécanisme : La procédure travaille sur une copie locale.
- Sécurité : Si la procédure modifie la valeur du paramètre à l'intérieur de son code, la variable d'origine dans le programme principal ne change pas.
- Usage : On l'utilise pour transmettre des informations dont la procédure a besoin pour ses calculs ou ses affichages sans risque de modifier l'original.

## 3.4.3. Le Passage par Référence (ou par Variable)

Dans ce mode, on ne transmet pas une copie, mais l'adresse mémoire de la variable d'origine. En Pascal, on utilise le mot-clé VAR devant le paramètre.

- Mécanisme : La procédure pointe directement vers la variable du programme principal.
- Impact : Toute modification effectuée sur le paramètre à l'intérieur de la procédure modifie réellement et immédiatement la variable d'origine.

- Usage : On l'utilise lorsque la procédure a pour but de mettre à jour une donnée ou de renvoyer plusieurs résultats (ex: une procédure Echanger(a, b)).

### 3.4.4. Exemple Comparatif

Voici un algorithme illustrant la différence entre les deux modes : Delphi

```
Procédure TesterPassages(n1 : Entier ; VAR n2 : Entier)
Début
    n1 <- n1 + 10  { Modification sur une copie }
    n2 <- n2 + 10  { Modification sur l'original }
FinProcédure

{ Programme Principal }
Variables
    A, B : Entier
Début
    A <- 5
    B <- 5
    TesterPassages(A, B)
    Ecrire("A = ", A) { Affichera 5 : Passage par valeur }
    Ecrire("B = ", B) { Affichera 15 : Passage par référence }
Fin
```

### 3.4.5. Passage par Valeur contre Passage par Référence

| **Caractéristique** | **Passage par Valeur** | **Passage par Référence (VAR)** |
|---|---|---|
| Transmission | Envoie une copie de la donnée. | Envoie l'adresse mémoire (le pointeur). |
| Sécurité | Protection totale de la variable d'origine contre toute modification. | La variable d'origine peut être modifiée par le sous-programme. |
| Performance | Peut être coûteux en mémoire pour des structures lourdes (gros tableaux). | Très performant car aucune copie n'est effectuée. |
| Usage Type | Fournir une information nécessaire à un traitement ou un calcul. | Retourner un ou plusieurs résultats ou mettre à jour une donnée. |

# Chapitre 4. Les Fonctions

Si la procédure est une "action", la fonction est un "calcul". Elle se comporte exactement comme une fonction mathématique : elle reçoit des données en entrée, effectue un traitement, et produit un résultat que l'on peut immédiatement utiliser.

## 4.1. Définition

Une fonction est un sous-programme qui, après avoir effectué un ensemble d'instructions, retourne obligatoirement un résultat unique au programme appelant.

Contrairement à la procédure qui s'utilise comme une instruction autonome, la fonction est perçue par le programme principal comme une valeur. Elle peut donc être utilisée dans une affectation, une condition ou une opération arithmétique.

## 4.2. Syntaxe et Typage

La déclaration d'une fonction impose de préciser deux éléments supplémentaires par rapport à la procédure : le type du résultat et l'instruction de retour.

### 4.2.1. Structure de la déclaration

```
Fonction Nom_Fonction ( Paramètres ) : Type_du_Résultat
Variables
    { Déclarations locales }
Début
    { Instructions }
    Nom_Fonction <- Valeur_Finale  { Instruction de retour }
FinFonction
```

- Type_du_Résultat : On doit préciser si la fonction renvoie un Entier, un Réel, un Booléen ou une Chaîne.
- L'instruction de retour : En algorithmique (et en Pascal), on affecte le résultat final au nom même de la fonction pour "renvoyer" la valeur.

### 4.2.2. Exemple d'application : Calcul de la surface d'un cercle

```
Fonction CalculerSurface(Rayon : Réel) : Réel
Constantes
    PI = 3.14159
Début
    CalculerSurface <- PI * (Rayon * Rayon)
FinFonction
```

### 4.2.3. L’appel de la fonction

Puisqu’elle renvoie une valeur, on l’appelle généralement à l’intérieur d’une autre instruction :

**Dans une affectation :**

```
S <- CalculerSurface(5.0)
```

**Dans un affichage :**

```
Ecrire("La surface est : ", CalculerSurface(R))
```

**Dans une condition :**

```
Si (CalculerSurface(R) > 100) Alors ...
```

## 4.3. Comment choisir entre fonction et procédure

Le choix entre une fonction et une procédure dépend de la nature du résultat attendu.

| Critère | Choisir une Fonction | Choisir une Procédure |
|---|---|---|
| Nombre de résultats | Toujours un seul résultat. | Zéro, un ou plusieurs résultats. |
| Usage | Pour un calcul ou une vérification. | Pour une action ou un traitement complexe. |
| Appel | Intégrée dans une expression (ex: x = f(y)). | Appelée comme une instruction seule. |
| Effet | On attend une réponse (valeur). | On attend un effet (affichage, modif de variable). |

# Chapitre 5. La Portée des Variables

## 5.1. Les Variables Globales

Une variable est dite globale lorsqu'elle est déclarée au tout début du programme, en dehors de tout sous-programme.

- Visibilité : Elle est accessible et modifiable partout (dans le programme principal et dans toutes les fonctions et procédures).
- Durée de vie : Elle existe pendant toute la durée d'exécution du programme.
- Usage : On les utilise pour des données qui concernent l'ensemble de l'application (ex: un taux de TVA, le nom de l'utilisateur connecté).

## 5.2. Les Variables Locales

Une variable est dite locale lorsqu'elle est déclarée à l'intérieur d'une procédure ou d'une fonction.

- Visibilité : Elle n'est accessible qu'à l'intérieur du bloc où elle a été déclarée. Le programme principal ne peut pas voir ce qui se passe à l'intérieur d'une fonction.
- Durée de vie : Elle est "éphémère". Elle est créée au moment de l'appel du sous-programme et détruite dès que celui-ci se termine.
- Usage : On les utilise pour des calculs intermédiaires ou des compteurs de boucles (ex: le i d'une boucle Pour).

| **Critère** | **Variable Globale** | **Variable Locale** |
|---|---|---|
| Lieu de déclaration | En haut du programme (hors modules). | À l'intérieur d'un module. |
| Accessibilité | Partout dans le code. | Uniquement dans son module. |
| Durée de vie | Toute la durée du programme. | Le temps d'exécution du module. |
| Risque | Élevé (conflits, bogues invisibles). | Faible (confinement des erreurs). |

## 5.3. Le Masquage (Shadowing)

Le masquage est lorsqu'une variable locale a le même identifiant qu'une variable globale.

À l'intérieur du sous-programme, la variable locale est prioritaire. Elle "cache" la variable globale. Une fois sorti du sous-programme, on retrouve la valeur de la variable globale intacte.

**Exemple :**

```
Variables
    X : Entier { Variable GLOBALE }
```

```
Procédure Tester()
Variables
    X : Entier { Variable LOCALE }
Début
    X <- 20
    Ecrire("Dans la procédure, X vaut : ", X)
FinProcédure

{ Programme Principal }
Début
    X <- 10
    Tester()
    Ecrire("Dans le programme principal, X vaut : ", X)
Fin
```

**Résultat :**

- Dans la procédure, X vaut : 20
- Dans le programme principal, X vaut : 10 (La globale n'est pas modifiée)

# Chapitre 6. La Récursion

## 6.1. Définition

Une fonction est dite récursive lorsqu'elle s'appelle elle-même à l'intérieur de son propre corps d'instructions.

L'idée fondamentale est de résoudre un problème en le décomposant en une version plus petite de lui-même. Au lieu d'utiliser une boucle (Pour ou TantQue), la fonction délègue le travail à une "copie" d'elle-même, jusqu'à atteindre un cas si simple qu'il peut être résolu directement.

## 6.2. Les règles de la récursion

Pour qu'une fonction récursive ne tourne pas à l'infini, elle doit respecter deux conditions :

- Le Cas de Base (ou Condition d'arrêt) : C'est l'état le plus simple du problème où la fonction s'arrête de s'appeler et renvoie une valeur directe.
- L'Appel Récursif (ou Étape de progression) : La fonction s'appelle elle-même avec un paramètre "plus petit", se rapprochant ainsi progressivement du cas de base.

## 6.3. Exemple : La Factorielle

En mathématiques, la factorielle d'un nombre entier n (notée n!) est le produit de tous les entiers de 1 à n.

- Par définition : $n! = n \times (n-1)!$
- Cas de base : $0! = 1$

Algorithme récursif :

```
Fonction Factorielle(n : Entier) : Entier
Début
    { 1. Cas de base }
    Si (n = 0) Alors
        Factorielle <- 1
    { 2. Appel récursif }
    Sinon
        Factorielle <- n * Factorielle(n - 1)
    FinSi
FinFonction
```

# Chapitre 7. Conclusion

L’étude des procédures et des fonctions constitue une étape fondamentale dans l’apprentissage de l’algorithmique. Elle marque la transition entre une programmation séquentielle rudimentaire et une méthodologie de conception logicielle structurée. En adoptant le paradigme modulaire, le concepteur n’écrit plus seulement du code exécutable, mais élabore une architecture de composants cohérents.

La maîtrise de la modularité est le fondement du génie logiciel moderne. Elle permet la réutilisabilité du code à travers diverses applications. Dans un contexte professionnel, la capacité à décomposer un problème complexe en sous-ensembles logiques est la compétence qui permet de garantir la maintenabilité et l’extensibilité des systèmes d’information.