

# Table of contents

420-2N2-DM Développement d'applications natives I .....	3
Chapitre 1 : Les bases du langage Go .....	5
Section 1 : Le premier programme .....	6
Section 2 : Les variables et les types de base .....	9
Section 3 : Les différentes façons d'écrire des valeurs .....	13
Section 4 : Les fonctions .....	17
Section 5 : Les conditionnelles .....	20
Section 6 : Les boucles .....	24
Exercices .....	29
Références .....	34
Chapitre 2 : Les pointeurs .....	35
Section 1 : Les pointeurs .....	38
Section 2 : Appels de fonctions et types de paramètres .....	40
Section 3 : Lire des valeurs avec Scanf .....	44
Section 4 : Validation des entrées .....	49
Exercices .....	54
Chapitre 3 : Tableaux et tranches .....	55
Section 1 : Valeur ou pointeur ? .....	56
Section 2 : Tranches .....	60
Section 3 : Chaînes de caractères .....	67
Exercices .....	69
Chapitre 4 : La gestion des fichiers .....	70
Section 1 : Lire le contenu d'un fichier .....	71
Section 2 : Écrire dans un fichier .....	76
Section 3 : Copier un fichier en ajoutant des numéros de ligne .....	82
Section 4 : Autres façons d'ouvrir un fichier .....	85
Chapitre 5 : Les structures .....	90
Section 1 : Valeur vs. pointeur .....	92
Section 2 : Méthodes .....	95
Section 3 : Sérialisation des structures .....	99
Chapitre 6 : Gestion de la mémoire .....	108
Section 1 : La pile d'exécution .....	110
Section 2 : Le tas .....	113
Section 3 : Le vidangeur .....	116

Section 4 : Les tranches .....	117
Opérations sur les tranches .....	120
Chapitre 7 : Structures de données .....	129
Section 1 : Les dictionnaires (map) .....	132
Section 2 : Les listes chaînées .....	137
Chapitre 8 : Fils d'exécution .....	138
Section 1 : Retour sur la nature des fonctions .....	139
Section 2 : Goroutines .....	144
Section 3 : Canaux .....	148
Section 4 : WaitGroup .....	151

# 420-2N2-DM Développement d'applications natives I

## Notes de cours

### Le langage de programmation Go

Go est un langage de programmation **hautes performances** conçu pour l'efficacité et la fonctionnalité. Développé par Google, Go, également connu sous le nom de *Golang*, a été créé pour résoudre les problèmes associés à des projets de grande envergure impliquant un travail d'équipe étendu. Par conséquent, l'objectif de conception principal pour Go repose sur la simplicité, tant pour la construction de logiciels efficaces que pour en améliorer la maintenabilité globale.

Go partage des caractéristiques communes avec Java, comme être un langage à typage statique et disposer d'un *ramasse-miettes* ou *vidangeur* (en anglais : *garbage collector*). Ces caractéristiques offrent des protections contre de nombreuses erreurs potentielles qui peuvent être rencontrées dans les langages à typage dynamique. Néanmoins, Go offre une approche plus directe et simplifiée par rapport à Java.

Java est un langage entièrement orienté objet qui utilise des concepts complexes tels que les classes, l'héritage et le polymorphisme. D'autre part, Go adopte un modèle plus simple avec des fonctionnalités telles que des fonctions, des *structs* (comparables à des classes allégées), et des interfaces permettant une organisation et une réutilisation efficaces du code. Cela conduit à un code plus transparent, donc plus facile à lire et à écrire, ce qui peut s'avérer très bénéfique pour les programmeurs explorant un nouveau langage.

Une fonctionnalité très attendue ajoutée dans Go 1.18 est le support pour les **génériques**. Les génériques permettent d'écrire des fonctions et des types qui peuvent manipuler des valeurs de n'importe quel type, tout en conservant la sécurité des types. C'est un aspect assez similaire à celui des génériques de Java et peut conduire à un code plus réutilisable et plus efficace.

Parmi les détails plus fins de Go, citons également son **excellent support pour la programmation concurrente**, qui utilise très efficacement les processeurs multicœurs. Les **goroutines** et les **canaux** de Go **simplifient les calculs concurrents et parallèles**, offrant des avantages significatifs pour le développement web moderne et le développement logiciel. De plus, la syntaxe propre et concise de Go est souvent appréciée parmi les programmeurs. Que ce soit pour la gestion des chaînes ou l'implémentation de structures complexes, la

bibliothèque standard de Go offre des méthodes simples pour écrire du code rapidement et efficacement.

En outre, Go est équipé d'un ensemble d'outils sophistiqué. Celui-ci comprend un outil de couverture de tests, un outil de mesure de performance, un détecteur de conflits, et de la documentation automatisée, tous disponibles immédiatement à la portée de la main, augmentant considérablement la productivité du programmeur.

En conclusion, Go représente une voie intéressante à explorer. Avec sa communauté en croissance, son solide soutien d'entreprise et son adoption de plus en plus répandue pour l'écriture d'applications modernes, robustes et évolutives, l'apprentissage de Go serait certainement un temps bien passé. Bonne programmation !

# Chapitre 1 : Les bases du langage Go

## Avant de commencer

1. Vous devez Utiliser votre compte JetBrains éducation

- si vous n'en avez pas, créez-en un ici (<https://www.jetbrains.com/fr-fr/community/education/#students>)

2. Installez GoLand (<https://www.jetbrains.com/go/>)

- vous pouvez utiliser d'autres logiciels pour créer vos programmes Go, mais il est fortement recommandé d'utiliser GoLand

3. Installez Go (<https://go.dev>)

1. le plus simple est d'installer à travers l'interface de Goland  
(<https://www.jetbrains.com/go/learn/>)

- créez un nouveau projet Go
- ajoutez un *SDK* Go, en téléchargeant la version la plus récente

2. il est possible d'installer Go à partir du site web (<https://go.dev>)

- c'est l'option à choisir si vous n'utilisez pas GoLand

## Code

Code chapitre 1 (<https://github.com/profdenis/native1/tree/master/chap1>)

# Section 1 : Le premier programme

Ce n'est pas original, mais le premier programme dans un nouveau langage de programmation est presque toujours le fameux *Hello World !*.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World !")
}
```

## Explications

1. package main: tous les fichiers .go doivent débuter par une déclaration de paquet (*package*), et lorsqu'on veut un *exécutable*, le paquet doit être *main*, et il doit y avoir une fonction appelée *main*.
2. import "fmt": on doit importer le paquet *fmt* pour pouvoir utiliser la fonction *Println* pour écrire du texte sur la sortie standard (*standard output*, ou *stdout*).
3. func main(): déclaration de la fonction *main*, qui sera le point de départ de notre programme exécutable.
4. fmt.Println("Hello World !"): le contenu de la fonction est évidemment très simple, on écrit simplement une chaîne de caractères sur le *stdout*
  1. Print--> imprime (ou écrit), In--> ajoute un changement de ligne (un \n) à la fin
  2. les fonctions et les variables **publiques** commencent toujours avec une lettre majuscule en Go : si la fonction était définie avec une lettre minuscule au début, soit *println* à la place, la fonction serait **privée** et on ne pourrait pas l'appeler
  3. il existe plusieurs autres fonctions dans le paquet *fmt*, pour écrire sur le *stdout* ou ailleurs
5. }: les blocs de code sont délimités par des accolades (comme en Java), mais il n'y a pas de point-virgules ; à la fin des lignes qui contiennent des énoncés qui ne nécessitent pas de

définition de bloc

## Compilation et exécution

La compilation avec Go peut être décrite en plusieurs étapes :

1. **Analyse Lexicale et Syntaxique:** Le compilateur Go lit le texte du programme source dans des fichiers `.go` caractère par caractère. En parcourant le texte, il identifie les mots-clés, les identificateurs, les littéraux, etc. C'est l'expression du programme sous forme d'arborescence de syntaxe abstraite (AST).
2. **Analyse Sémantique:** La prochaine étape est l'analyse sémantique où le compilateur vérifie les déclarations et les affectations de types pour s'assurer qu'elles sont logiquement correctes dans le contexte du langage de programmation Go. Par exemple, affecter une valeur de chaîne à une variable déclarée comme un entier serait détecté comme une erreur de type lors de cette étape.
3. **Génération du code intermédiaire:** Go génère un code intermédiaire appelé *SSA (Single Static Assignment)*, qui simplifie l'optimisation et la traduction ultérieure en code machine.
4. **Optimisation:** Go effectue plusieurs optimisations pour améliorer la performance du programme final, y compris l'élimination de code mort (code qui ne peut jamais être exécuté), l'*inlining* (remplacer une invocation de fonction par le corps de la fonction), etc.
5. **Génération du code machine:** Enfin, Go convertit le code SSA en code binaire pour la machine cible. Cela inclut également le regroupement de toutes les dépendances (autres packages Go) nécessaires au programme. Le code machine est écrit dans un fichier (par exemple, un fichier exécutable `.exe` dans Windows).

Le code binaire peut alors être exécuté par l'ordinateur. Dans ce processus, **il est important de comprendre que le GoLang est un langage compilé vers une architecture spécifique (x86, ARM, etc.) et un système d'exploitation spécifique (Linux, Windows, OSX, etc.).**

Par conséquent, pour exécuter un programme Go sur une plate-forme différente, vous devrez le recompiler pour cette plate-forme.

Supposons que l'exemple précédent a été enregistré dans le fichier `main.go`. Pour compiler ce programme, vous utiliserez la commande `go build` pour le compiler, ce qui produira un fichier binaire exécutable.

```
go build main.go
```

Ensuite, vous pouvez exécuter le programme compilé :

```
./main
```

Il imprimera "Hello, world!" à la sortie.

Notez que vous pouvez également utiliser la commande `go run` pour compiler et exécuter le programme en une seule étape, bien que cela ne laisse pas d'exécutable persistant.

C'est un aperçu de haut niveau, et en réalité le processus est assez complexe. Le but ici n'est pas de comprendre les détails du processus de compilation, mais de comprendre l'idée générale.

# Section 2 : Les variables et les types de base

Référence : A tour of Go (<https://go.dev/tour/basics/11>)

catégorie	types
logique	bool
caractères	string, rune (alias pour int32)
nombres entiers (signés)	int, int8, int16, int32, int64
nombres entiers positifs (non signés)	uint, uint8, uint16, uint32, uint64, uintptr
données brutes	byte (alias for uint8)
nombres réels	float32, float64
nombres complexes	complex64, complex128

Les types int, uint, et uintptr ont une largeur de 32 bits sur un système 32-bit et une largeur de 64 bits sur un système 64-bit. Quand vous avez besoin d'un nombre entier, vous devriez utiliser int à moins d'avoir une raison spécifique d'utiliser une taille précise ou un nombre non-signé.

## Définition avec var et sans valeur initiale

Une déclaration de type var commence avec le mot-clé var, suivi par le nom de la variable ou une liste de noms de variables séparées par des virgules, et suivi par le type de la ou les variable(s).

```
package main
```

```
import "fmt"
```

```

func variables2a() {
    var i int
    var finished bool
    var name string
    var x, y float32

    fmt.Println(i)
    fmt.Println(finished)
    fmt.Println(name)
    fmt.Println(x, y)
}

```

Les variables vont être automatiquement initialisées avec leur **valeur zéro** (ou leur valeur par défaut), qui dépend du type de la variable. Pour les nombres, la valeur par défaut est 0, pour les booléens la valeur est `false`, et pour les chaînes de caractères, la chaîne vide `""` est utilisée comme valeur zéro.

## Définition avec var et valeur initiale

On peut aussi préciser une valeur pour initialiser une variable au moment de la déclaration.

```

package main

import "fmt"

func variables2b() {
    var i int = 5
    var finished bool = true
    var name string = "Denis"
    var x, y float32 = 2.5, -1.4

    fmt.Println(i)
    fmt.Println(finished)
    fmt.Println(name)
    fmt.Println(x, y)
}

```

Si une valeur initiale est spécifiée, alors il n'est pas nécessaire de spécifier le type. La variable prendra automatiquement le type de la valeur utilisée. L'exemple suivant est équivalent à

l'exemple précédent, à une différence près : les variables `x` et `y` sont maintenant de type `float64`, et non `float32`. Le type pour les nombres réels est `float64` par défaut.

```
package main

import "fmt"

func variables2c() {
    var i = 5
    var finished = true
    var name = "Denis"
    var x, y = 2.5, -1.4

    fmt.Println(i)
    fmt.Println(finished)
    fmt.Println(name)
    fmt.Println(x, y)
    fmt.Printf("%T\n", x)
}
```

Sortie :

```
5
true
Denis
2.5 -1.4
float64
```

La dernière ligne de code écrit le type de la variable `x` sur la sortie standard (le `stdout`). Plus de détails sur les sorties seront donnés dans la section suivante.

## Constantes

On peut définir une constante en remplaçant `var` par `const`. Essayer de modifier la valeur d'une constante va créer une erreur de compilation.

## Définition avec le raccourci :=

Il est possible d'utiliser un raccourci pour les définitions de variables avec valeurs initiales. L'exemple suivant est équivalent à l'exemple précédent, sauf que `x` et `y` sont définies

séparément.

```
package main

import "fmt"

func variables2d() {
    i := 5
    finished := true
    name := "Denis"
    x := 2.5
    y := -1.4

    fmt.Println(i)
    fmt.Println(finished)
    fmt.Println(name)
    fmt.Println(x, y)
    fmt.Printf("%T\n", x)
}
```

Il est aussi possible de définir 2 variables ou plus à la fois sur une même ligne, dans la même déclaration comme ceci `x, y := 2.5, -1.4`, mais il est souvent préférable en général de définir seulement une variable à la fois pour faciliter la lecture et la compréhension du code. Dans des situations simples, comme dans l'exemple qui vient d'être donné, ça peut aller, mais avec des initialisations plus complexes, c'est préférable de séparer les déclarations. Un endroit courant où plus d'une variable est définie dans une seule déclaration est lorsqu'une fonction retourne plus d'une valeur, ou en dans une boucle sur un intervalle, comme un `range` sur un tableau par exemple. Plus de détails viendront dans les prochaines sections.

# Section 3 : Les différentes façons d'écrire des valeurs

En plus des fonctions `Print`, `Println` et `Printf` qui sont utilisées pour écrire sur le `stdout`, le paquet `fmt` contient des fonctions pour écrire dans une chaîne de caractères (`Sprint`, `Sprintln`, `Sprintf`) ou dans un fichier (`Fprint`, `Fprintln`, `Fprintf`).

- Les fonctions se terminant par `/n` ajoutent un changement de ligne (`\n`) à la fin de la chaîne de caractères à écrire.
- Les fonctions se terminant par un `f` utilisent une syntaxe spéciale pour **formatter** les chaînes de caractères.
- Les fonctions commençant avec un `S` écrivent dans une chaîne de caractères (`S` pour *string*)
- Les fonctions commençant avec un `F` écrivent dans un fichier (`F` pour *file*)
  - Les fonctions sur les fichiers seront présentées plus loin, dans le chapitre consacré à la gestion des fichiers.

## Écrire dans une chaîne de caractères

```
package main

import "fmt"

func print1() {
    year := 2024
    output := fmt.Sprintln("année =", year)
    fmt.Println(output)
}
```

Dans cet exemple simplifié, il serait préférable d'utiliser `Println` directement sur les deux valeurs à écrire (`fmt.Println("année =", year)`), mais dans un exemple plus complexe, lorsqu'il est nécessaire d'écrire la même chaîne de caractères à plusieurs reprises, possiblement à des endroits différents (`stdout`, fichier, ...), écrire dans une chaîne en premier peut simplifier le

code plus loin dans le programme, surtout si on a besoin de formatter les valeurs de façon particulière.

## Écrire selon un format

Le même principe s'applique aux fonctions `Printf`, `Sprintf` et `Fprintf`: une chaîne de caractères décrivant le format doit être donnée comme premier argument aux fonctions, suivie d'une ou plusieurs valeurs à formatter.

### Nombres entiers

```
package main

import "fmt"

func print2() {
    year := 2024
    month := 1
    day := 5

    fmt.Printf("année = %d\n", year)
    fmt.Printf("%d-%d-%d\n", year, month, day)
    fmt.Printf("%4d-%2d-%2d\n", year, month, day)
    fmt.Printf("%4d-%02d-%02d\n", year, month, day)
}
```

Sortie :

```
année = 2024
2024-1-5
2024- 1- 5
2024-01-05
```

La chaîne de caractères décrivant le format (*chaîne de format*, ou *format string* en anglais) doit inclure au moins une entrée débutant par le caractère `%`, suivi par la description du format. Dans l'exemple précédent, il y a 4 sorties. Dans la première, la sortie sera identique à l'exemple précédent. Le `%d` va être remplacé par la valeur de la variable `year`. Le `d` indique que la valeur sera un nombre entier en base 10 (*décimal*). On pourrait utiliser

- `%x` pour un nombre entier en base 16 (*hexadécimal*)
- `%o` pour un nombre entier en base 8 (*octal*)
- `%b` pour un nombre entier en base 2 (*binnaire*)

Dans la deuxième sortie, il y a 3 formats à l'intérieur de la chaîne de format, et ils sont séparés par des tirets qui ne font pas partie des formats, mais qui feront partie de la sortie. Le problème est que la sortie n'est pas très belle parce que les valeurs pour le mois et la journée sont plus petites que 10, alors qu'habituellement, on réserve 2 emplacements (ou 2 colonnes) pour ces valeurs même si elles n'en n'ont pas besoin.

On peut remplacer le deuxième format par celui-ci : `"%4d-%2d-%2d\n"`. La sortie sera alors `2024- 1- 5`, ce qui n'est pas vraiment mieux. On spécifie que l'on veut avoir 4 colonnes pour l'année, 2 colonnes pour le mois et aussi 2 colonnes pour le jour, mais par défaut, les colonnes libres sont remplies avec des espaces.

Si on remplace le deuxième format par celui-ci : `"%4d-%02d-%02d"`, alors les colonnes libres seront remplies par des zéros, ce qui correspond au format habituel `2024-01-05`.

Cet exemple montre comment formatter une date dans un format précis, mais il existe plusieurs autres formats possibles. Formatter des dates est un besoin commun, qui doit supporter des formats différents selon les paramètres de localisation de l'application. Le paquet `time` peut être utilisé pour, entre autres, formatter les dates et les heures. Pour plus de détails : Date and time format in Go cheatsheet (<https://gosamples.dev/date-time-format-cheatsheet/>)

## Formatter d'autres types

Référence : Paquet `fmt` (<https://pkg.go.dev/fmt>)

Les formats les plus courants :

Formats	Détails
%v	format par défaut
%d, %x, %b	nombres entiers en bases 10, 16 et 2
%f, %F	nombres réels, avec un point, sans exposant
%e, %E	nombres réels, notation scientifique
%s	chaînes de caractères
%t	booléen
%T	type de la valeur

Pour les nombres réels, on peut spécifier la précision voulue.

```
package main

import "fmt"

func print3() {
    x := 1234.5678
    fmt.Printf("%f\n%.2f\n%.f\n", x, x, x)
}
```

Sortie :

```
1234.567800
1234.57
1235
```

# Section 4 : Les fonctions

Une déclaration de fonction a la structure suivante en Go :

```
func nomFonction(param1 type1, param2 type2, ...) typeRetour {}
```

Une déclaration commence toujours par `func`, suivi du nom de la fonction, qui doit être écrite en `camelCase` ou en `PascalCase`. Comme pour les variables, si le nom de la fonction commence par une **minuscule**, la fonction sera **privée** et accessible seulement dans le paquet qui la contient, tandis que si le nom commence par une **majuscule**, la fonction sera **publique**, donc accessible de l'extérieur du paquet. Donc, pour une fonction privée, on utilise le *camelCase*, et pour une fonction publique, on utilise le *PascalCase*.

Si la fonction ne retourne rien, alors on doit omettre le type de retour. On ne doit **pas** utiliser `void` comme dans d'autres langages.

```
package main

import "fmt"

func allo(name string) {
    fmt.Printf("Allo %s !\n", name)
}
```

Sortie :

```
Allo Alice !
```

Voici une fonction qui fait un calcul simple et retourne le résultat :

```
package main

import "fmt"

func square(x int) int {
    return x * x
}
```

Sortie avec l'appel `fmt.Println(square(4))`: 16

Il est possible pour une fonction de retourner plus qu'une valeur. Les types de retours doivent être séparés par des virgules, et des parenthèses doivent être utilisées autour des types de retour.

```
func nomFonction(param1 type1, param2 type2, ...) (typeRetour1,  
typeRetour2, ...){}
```

Voici une fonction qui retourne les 2 valeurs données en paramètre, mais en ordre inverse.

```
package main  
  
import "fmt"  
  
func swap(x int, y int) (int, int) {  
    return y, x  
}
```

Appel de la fonction :

```
a, b := 2, 5  
fmt.Println(swap(a, b))
```

Sortie : 5 2

Voici une autre fonction pour échanger des valeurs, mais elle ne fonctionne pas :

```
package main  
  
import "fmt"  
  
func swap2(x int, y int) {  
    temp := x  
    x = y  
    y = temp  
}
```

Appel de la fonction :

```
a, b := 2, 5
fmt.Println(a, b)
swap2(a, b)
fmt.Println(a, b)
```

Sortie :

```
2 5
2 5
```

Plus de détails vont être donnés dans le chapitre suivant, mais la raison pourquoi cette fonction n'a aucun effet sur les variables `a` et `b` est que la fonction travaille sur des copies des valeurs de `a` et `b`. Les variables `x` et `y` sont initialisés avec les valeurs de `a` et `b`, mais sont indépendantes de `a` et `b`.

La meilleure façon d'échanger les valeurs de deux variables en Go est celle-ci :

```
b, a = a, b.
```

`b` va obtenir la valeur de `a`, et `a` va obtenir la valeur de `b`. Ça fonctionne parce que les changements de valeurs vont se faire en parallèle.

# Section 5 : Les conditionnelles

## if ... else

La particularité des conditionnelles dans le langage Go est qu'on ne place pas la condition entre parenthèses, et que les accolades sont obligatoires autour du code à exécuter dans la partie `if` et dans la partie `else`, même s'il y a seulement une ligne de code à exécuter. De plus, l'accolade d'ouverture `{` doit être sur la même ligne que le `if` ou le `else`.

Utilisez les opérateurs `&&` pour la conjonction (et), `||` pour la disjonction (ou) et `!` pour la négation. Ajouter des parenthèses au besoin pour s'assurer que les opérateurs sont appliqués dans le bon ordre.

```
package main

import "fmt"

func conditional1(x int) {
    if x > 0 {
        fmt.Printf("x = %d est plus grand que 0\n", x)
    } else {
        fmt.Printf("x = %d est plus petit ou égal à 0\n", x)
    }
}
```

Sortie pour des appels avec les valeurs 5, 0, et -5 :

```
x = 5 est plus grand que 0
x = 0 est plus petit ou égal à 0
x = -5 est plus petit ou égal à 0
```

Il n'y a pas de syntaxe particulière pour les séquences de `if ... else`, il faut simplement démarrer un autre `if ... else` tout de suite après le premier `else`.

```
package main

import "fmt"
```

```

func conditional2(x int) {
    if x > 0 {
        fmt.Printf("x = %d est plus grand que 0\n", x)
    } else if x == 0 {
        fmt.Println("x est égal à 0")
    } else {
        fmt.Printf("x = %d est plus petit que 0\n", x)
    }
}

```

Sortie pour des appels avec les valeurs 5, 0, et -5 :

```

x = 5 est plus grand que 0
x est égal à 0
x = -5 est plus petit que 0

```

## switch

Un *switch* (*aiguillage*) est une bonne façon de simplifier la syntaxe quand on a besoin d'une série de `if ... else`. Il existe 2 principales formes de `switch` en Go : un `switch` sur une variable avec des cas sur différentes valeurs possibles de la variable, et un `switch` sans variable avec des cas sur des expressions booléennes.

### switch sans variable

```

package main

import "fmt"

func conditional3(x int) {
    switch {
    case x > 0:
        fmt.Printf("x = %d est plus grand que 0\n", x)
    case x == 0:
        fmt.Println("x est égal à 0")
    case x < 0:
        fmt.Printf("x = %d est plus petit que 0\n", x)
    }
}

```

Sortie pour des appels avec les valeurs 5, 0, et -5 : même que l'exemple précédent.

Cet exemple correspond presque exactement à l'exemple précédent écrit avec des `if ... else`, sauf pour le dernier cas qui pourrait être remplacé par le cas spécial `default` qui correspond au dernier `else` d'une série de `if ... else`. On obtiendra la même sortie avec ce changement. Dans ce cas-ci, remplacer `case x < 0` par `default` est approprié parce que la condition `x < 0` couvre toutes les possibilités restantes. Notez qu'il n'est pas nécessaire d'utiliser des `break` pour terminer chaque cas.

```
package main

import "fmt"

func conditional4(x int) {
    switch {
    case x > 0:
        fmt.Printf("x = %d est plus grand que 0\n", x)
    case x == 0:
        fmt.Println("x est égal à 0")
    default:
        fmt.Printf("x = %d est plus petit que 0\n", x)
    }
}
```

## switch avec variable

```
package main

import "fmt"

func conditional5(name string) {
    switch name {
    case "Denis":
        fmt.Println("Bonjour Prof. Denis !")
    case "":
        fmt.Println("Bonjour Inconnu !")
    default:
        fmt.Printf("Allo %s !\n", name)
```

```
    }  
}
```

Sortie pour des appels avec les valeurs "Denis", "", et "Alice":

```
Bonjour Prof. Denis !  
Bonjour Inconnu !  
Allo Alice !
```

Lorsqu'on *switch* sur une variable, les cas doivent être des valeurs spécifiques de la variable. Si on veut grouper plusieurs valeurs dans un même cas, on peut les séparer par des virgules, comme dans l'exemple suivant.

```
package main  
  
import "fmt"  
  
func conditional6(name string) {  
    switch name {  
    case "Denis", "Benoit":  
        fmt.Printf("Bonjour Prof. %s !\n", name)  
    case "":  
        fmt.Println("Bonjour Inconnu !")  
    default:  
        fmt.Printf("Allo %s !\n", name)  
    }  
}
```

Sortie pour des appels avec les valeurs "Denis" et "Benoit":

```
Bonjour Prof. Denis !  
Bonjur Prof. Benoit !
```

# Section 6 : Les boucles

## Boucle avec un compteur

À la base, les boucles *pour* en Go ressemblent aux boucles *pour* dans d'autres langages comme C et Java, avec quelques différences :

- il n'y a pas de parenthèses () autour des 3 composantes principales de la boucle *pour*:  
initialisation ; condition ; incrémentation
- le compteur de la boucle est normalement défini avec la notation abrégée utilisant le symbole :=
- les 3 composantes principales sont optionnelles

```
package main

import "fmt"

func loop1() {
    for i := 0; i < 5; i++ {
        fmt.Println(i, " ")
    }
    fmt.Println()
}
```

Sortie :

```
0 1 2 3 4
```

## Les boucles de type while

Techniquement, il n'y a pas de boucles *while* en Go. Le seul type de boucle utilise *for*, comme démontré dans l'exemple précédent. Puisque les parties *initialisation* et *incrémentation* sont optionnelles, on peut écrire une boucle uniquement avec la condition.

```
package main
```

```
import "fmt"

func loop2() {
    i := 0
    for i < 5 {
        fmt.Println(i, " ")
        i++
    }
    fmt.Println()
}
```

La sortie est la même que l'exemple précédent.

## Boucle sans condition

Voici une **très mauvaise** manière d'écrire une boucle qui produit le même résultat, qui est techniquement légale, elle obéit à la syntaxe du langage, mais qui est beaucoup plus difficile à lire, et qui peut causer des problèmes de maintenance de code plus tard.

```
package main

import "fmt"

func loop3() {
    i := 0
    for {
        fmt.Println(i, " ")
        i++
        if i >= 5 {
            break
        }
    }
    fmt.Println()
}
```

À première vue, la boucle ressemble à une boucle infinie parce qu'il n'y a pas de condition après le `for`, ce qui est légal en Go. C'est la même chose que s'il y avait une condition qui est toujours vraie, comme `for true` (ou comme `while (true)` en Java). Mais cette boucle n'est pas

vraiment infinie parce qu'il y a une condition à l'intérieur qui devient vraie à un moment donné et qui fait un `break` pour sortir de la boucle.

En général, il faut éviter les boucles infinies ou qui en apparence ont l'air infinie parce qu'elles sont plus difficiles à lire et à déboguer. Les vraies boucles infinies sont utiles dans certaines situations très particulières, comme l'implémentation d'un serveur qui doit attendre un nombre indéfini de connexions ou de requêtes, mais ces situations particulières sont rares.

## Boucles sur les tableaux

Voici comment définir un tableau de nombres entiers avec valeurs initiales, et comment faire une boucle sur le tableau en utilisant les index, ou positions, des éléments dans le tableau, pour écrire tous les éléments du tableau sur le `stdout` avec un format particulier. On utilise la fonction `len` pour obtenir la longueur d'un tableau.

On peut également donner un tableau directement à la fonction `Println` pour en écrire le contenu.

```
package main

import "fmt"

func loop4() {
    numbers := [7]int{2, 5, 3, 4, 7, 1, 7}
    for i := 0; i < len(numbers); i++ {
        fmt.Printf("numbers[%d] = %d\n", i, numbers[i])
    }
    fmt.Println(numbers)
}
```

Sortie :

```
numbers[0] = 2
numbers[1] = 5
numbers[2] = 3
numbers[3] = 4
numbers[4] = 7
numbers[5] = 1
numbers[6] = 7
[2 5 3 4 7 1 7]
```

Si on n'initialise pas le tableau immédiatement avec des valeurs spécifiées entre accolades immédiatement après le type [7]int, alors le tableau sera initialisé avec les valeurs zéro du type, dans ce cas-ci 0 à cause du type int. Il est possible de remplacer la longueur 7 par ... pour que le compilateur calcule la longueur automatiquement. Il est également possible de ne rien spécifier entre les [], mais dans ce cas, on n'obtiendra pas un tableau, mais une slice (tranche). Les tableaux et les tranches se ressemblent, mais il y a des différences en termes de leurs propriétés et de la gestion de la mémoire. Plus de détails viendront à ce sujet dans le prochain chapitre.

Il existe une autre façon de parcourir tous les éléments d'un tableau en Go : en utilisant range. C'est la manière la plus pratique et la plus utilisée de parcourir tous les éléments d'un tableau à partir du début.

```
package main

import "fmt"

func loop5() {
    numbers := [...]int{2, 5, 3, 4, 7, 1, 7}
    for i, number := range numbers {
        fmt.Printf("numbers[%d] = %d\n", i, number)
    }
    fmt.Println(numbers)
}
```

La sortie est la même que l'exemple précédent. À chaque tour de boucle, i a l'index courant, et number a la valeur de l'élément courant du tableau. Les avantages de ce type de boucle sont qu'on n'a pas besoin de gérer l'index manuellement, il est incrémenté automatiquement, et la fin de boucle est vérifiée aussi automatiquement. Les désavantages de ce type de boucle sont qu'on ne peut pas modifier le contenu du tableau dans la boucle, et on ne peut pas parcourir les éléments du tableau dans un ordre différent (par exemple, de la fin vers le début).

Si on n'a pas besoin des index, mais seulement des valeurs des éléments dans le tableau, on doit alors faire la boucle de la façon suivante.

```
package main

import "fmt"

func loop6() {
```

```
numbers := [...]int{2, 5, 3, 4, 7, 1, 7}
for _, number := range numbers {
    fmt.Printf("%d ", number)
}
fmt.Println()
```

Sortie :

```
2 5 3 4 7 1 7
```

Dans le langage Go, toutes les variables déclarées doivent être utilisées au moins une fois. Puisque dans cet exemple, on ne veut pas utiliser l'index des éléments du tableau, on utilise la barre de soulignement `_` à la place d'un nom de variable. Si on écrivait `i` à la place de `_` dans cet exemple, on obtiendrait l'erreur `i declared and not used` au moment de la compilation.

# Exercices

- Créez un projet nommé `exercices` dans lequel vous placerez tous vos exercices, de ce chapitre et des autres chapitres.
- Créez un dossier nommé `chap01` à l'intérieur de ce dossier de projet.
- Créez un fichier `main.go` dans ce dossier, qui sera dans le paquet `main`, et qui contiendra la fonction `main`.
- Créez un fichier `exercices.go`, qui sera aussi dans le paquet `main`, qui contiendra les fonctions qui répondent aux questions suivantes.
- Créez le fichier `input.go` qui contiendra les fonctions `ReadLine` et `ReadFloat64`, à utiliser pour vous aider à répondre aux questions.
  - Vous pouvez appeler les fonctions de la manière suivante : `line, _ := ReadLine()` et `x, _ := ReadFloat64()`.
  - Les entrées/sorties seront présentées en détails dans les prochains chapitres. Pour le moment, les erreurs sont ignorées en utilisant `_`.
  - Contenu du fichier `input.go`:

```
package main

import (
    "bufio"
    "io"
    "os"
    "strconv"
)

func ReadLine() (string, error) {
    scanner := bufio.NewScanner(os.Stdin)
    if scanner.Scan() {
        return scanner.Text(), nil
    } else {
        err := scanner.Err()
    }
}
```

```

        if err != nil {
            return "", err
        } else {
            return "", io.EOF
        }
    }

func ReadFloat64() (float64, error) {
    line, err := ReadLine()
    if err != nil {
        return 0, err
    }
    x, err := strconv.ParseFloat(line, 64)
    if err != nil {
        return 0, err
    }
    return x, nil
}

```

- Pour chacune des questions suivantes, écrivez une fonction qui effectuera ce qui est demandé.
- Ajoutez un appel à chacune des fonctions dans la fonction `main` pour tester votre code. Vous pouvez commenter les appels aux fonctions précédentes dans le `main` pour éviter d'exécuter toutes les fonctions précédentes quand vous ajouter une nouvelle fonction.
  - N'effacez pas les appels précédents, gardez-les, mais vous pouvez les commenter.
- Utilisez des nombres réels (type `float64`) pour tous nombres, sauf quand vous devez absolument avoir un nombre entier.
  - Vous pouvez convertir `in float64` en `int` de cette façon :

```

var f float64 = 3.14
var i int = int(f)

```

## Questions

1. Calculer et afficher la valeur absolue d'un nombre entré par l'utilisateur.
2. Déterminer si le nombre *entier* entré par l'utilisateur est pair ou impair.
3. Lire trois nombres et imprimer le plus petit de ces trois nombres.
4. Calculer le salaire total d'un employé. On lit en entrée les données concernant son salaire horaire et le nombre d'heures travaillées. Si l'employé a travaillé plus de 40 heures, les heures supplémentaires sont payées à 1.5 fois le salaire horaire.
5. Lire trois nombres positifs représentant la longueur des côtés d'un triangle. Imprimer :
  - "Scalène" si les trois côtés sont inégaux
  - "Isocèle" si deux des côtés sont égaux
  - "Équilatéral" si les trois côtés sont égaux
6. Un professeur vous fournit trois notes calculées sur 100. Calculer la moyenne et imprimer échec si la note finale est inférieure à 60/100. Dans le cas contraire, imprimer la note obtenue par l'étudiant.
7. Lire en entrée une note finale d'un cours. Si la note est plus petite que 0, ou si la note est plus grande que 100, alors afficher "Cette note est invalide" et terminer la fonction. Si la note est valide, alors vous devez afficher une lettre correspondant à la note selon les conditions suivantes :
  - E: plus petite que 60
  - D: de 60 à moins que 70
  - C: de 70 à moins que 80
  - B: de 80 à moins que 90
  - A: 90 ou plus
8. Vous devez lire un nombre entre 1 et 10 inclusivement.
  - Si le nombre n'est pas dans cet intervalle, alors afficher "*invalide*".
  - Si le nombre est valide, alors afficher "*valide*".
9. Vous devez lire un nombre entre 1 et 10 inclusivement.

- Si le nombre n'est pas dans cet intervalle, alors afficher "*invalide*" et demander le nombre à nouveau. Vous devez vous assurer que le nombre est valide avant de continuer à la prochaine étape. Il n'y aucune limite sur le nombre d'essais incorrects.
- Si le nombre est valide, alors afficher "*valide*".

10. Vous devez lire un nombre entre 1 et 10 inclusivement.

- Si le nombre n'est pas dans cet intervalle, alors afficher "*invalide*" et demander le nombre à nouveau. Vous devez vous assurer que le nombre est valide avant de continuer à la prochaine étape. Il y a une limite de 3 essais incorrects.
- Si le nombre maximal d'essais incorrects a été atteint, alors afficher "*Nombre maximal d'essais atteint.*" et la doit se terminer.
- Si le nombre est valide, alors afficher "*valide*".

11. Vous devez lire un nombre *entier* et l'afficher à l'envers. Par exemple, l'utilisateur saisit 123456 et le programme affiche 654321. Pour cela il faudra utiliser la division et le modulo.

12. Vous devez lire un nombre *entier* et afficher un décompte à partir de ce nombre jusqu'à 0. Lorsque le décompte est terminé, afficher "Terminé !" à la place du nombre 0. Par exemple, si le nombre entré est 5, vous devez afficher

```
5
4
3
2
1
Terminé !
```

13. Pour chacune des questions suivantes, définissez un tableau de nombres entiers de cette manière : numbers := [...]int{2, 5, 3, 4, 7, 1, 7}. La longueur du tableau et les nombres qu'il contient sont à votre choix, mais ils devraient permettre de bien tester les questions. Répondez aux questions dans des fonctions différentes. En utilisant une boucle `for` sur les index du tableau,

1. trouvez la somme et la moyenne de tous les nombres dans le tableau.
2. affichez tous les nombres pairs contenus dans le tableau.

3. déterminez si tous les nombres dans le tableau sont positifs ou non. Si tous les nombres sont positifs, alors affichez *vrai*, sinon affichez *faux*.
14. Répétez la question précédente, mais en utilisant une boucle `for` et un `range` sur le tableau.

# Références

1. Go (<https://go.dev>)
2. GoLand (<https://www.jetbrains.com/go/>)
3. Go cheatsheet (<https://devhints.io/go>)
4. A tour of Go (<https://go.dev/tour/list>)
5. Paquet fmt (<https://pkg.go.dev/fmt>)

# Chapitre 2 : Les pointeurs

Dans le langage Go, il y a 2 types de variables :

- les **variables de type valeur**, et
- les **variables de type pointeur**.

## Variables de type valeur :

En Go, **tous les types sont par défaut des types valeur**. Cela signifie que lorsque vous attribuez une variable à une autre, la "valeur" est copiée. Si vous modifiez la nouvelle variable, l'originale ne sera pas affectée.

```
var a int = 5
b := a // b = 5, copie de la valeur de a
b = 10 // la modification de b n'affecte pas a
```

Dans cet exemple, lorsque vous changez la valeur de `b`, `a` reste le même. Cela est dû au fait que `b` est une copie de `a`.

## Variables de type pointeur :

Un pointeur, d'autre part, pointe vers l'emplacement mémoire d'une autre variable. Lorsque vous modifiez la valeur à laquelle pointe un pointeur, vous modifiez également la valeur originale.

```
var a int = 5
var b *int = &a // b pointe vers la mémoire de a
*b = 10          // nous changeons la valeur de a à travers b
```

Maintenant, si vous modifiez `*b`, `a` change aussi, parce que `b` est un pointeur vers `a`.

Les pointeurs sont utiles lorsque vous voulez partager une variable entre plusieurs fonctions, ou lorsque vous travaillez avec de grands ensembles de données que vous ne voulez pas copier.

Ces exemples sont seulement utilisés en guise d'introduction. Ce chapitre est consacré aux pointeurs et à la gestion de la mémoire. Il y aura donc beaucoup plus d'exemples et d'explications pour clarifier ces concepts dans les sections qui suivent.

La mémoire peut être allouée de 2 façons :

- l'allocation statique de la mémoire, et
- l'allocation dynamique de la mémoire.

### L'allocation statique de la mémoire :

Dans l'allocation statique, la taille et la position de la mémoire à allouer sont déterminées avant l'exécution du programme. Les variables déclarées globalement et localement dans une fonction sont des exemples de cette allocation. En Go, lorsque vous déclarez une variable comme suit :

```
var num int
```

La mémoire nécessaire pour stocker une valeur entière est automatiquement allouée. Cette allocation est appelée statique car la taille de la mémoire allouée est fixe et ne peut pas être modifiée pendant l'exécution.

### L'allocation dynamique de la mémoire :

Au contraire, l'allocation dynamique de la mémoire se produit pendant l'exécution du programme et permet d'allouer la quantité de mémoire nécessaire en temps réel. En Go, cette allocation est réalisée en utilisant le mot-clé `new` ou `make`. Par exemple :

```
ptr := new(int)
```

Dans ce cas, Go alloue la mémoire nécessaire pour stocker un entier et renvoie un pointeur vers cette mémoire. De même, la fonction `make` est utilisée pour allouer de la mémoire pour les *slices*, les *maps* et les *channels*. Cette allocation est dite dynamique parce que la quantité de mémoire peut varier pendant l'exécution.

Il est à noter que contrairement à certains autres langages de programmation, Go gère automatiquement la *libération* (ou "*désallocation*") de la mémoire à l'aide d'un **garbage collector** (*ramasse-miettes* ou *vidangeur*), de sorte que les développeurs n'ont pas à se soucier de la libération de la mémoire une fois qu'elle n'est plus nécessaire.

Dans d'autres langages comme le C et le C++, la libération de la mémoire est manuelle, ce qui est une source importante de problèmes potentiels, notamment les *fuites de mémoire* (*memory leaks*). Il est très difficile dans les applications moindrement complexes de s'assurer de l'absence de fuites de mémoire dans un langage qui utilise la libération manuelle. Il est plus

difficile de tester une application qui utilise un langage avec la libération manuelle de la mémoire.

Par contre, l'utilisation d'un *garbage collector* peut affecter la performance d'un programme dans certaines situations. Mais en général, cet impact est mineur. Dans la grande majorité des cas, les risques qu'amène la libération manuelle de la mémoire sont trop grand et beaucoup plus significatif que les inconvénients de l'utilisation d'un *garbage collector*.

Nous reviendrons sur ce sujet plus tard.

# Section 1 : Les pointeurs

En Go, un pointeur est une variable qui stocke l'adresse mémoire d'une autre variable. Les pointeurs sont une partie très fondamentale de Go et sont extrêmement utiles car ils permettent aux fonctions de modifier les données qui leur sont passées, parce qu'ils ont accès à la même adresse mémoire où réside la variable originale.

Pour créer un pointeur, nous utilisons l'opérateur `&` avant une variable. Cela renvoie l'adresse mémoire de cette variable. Par exemple :

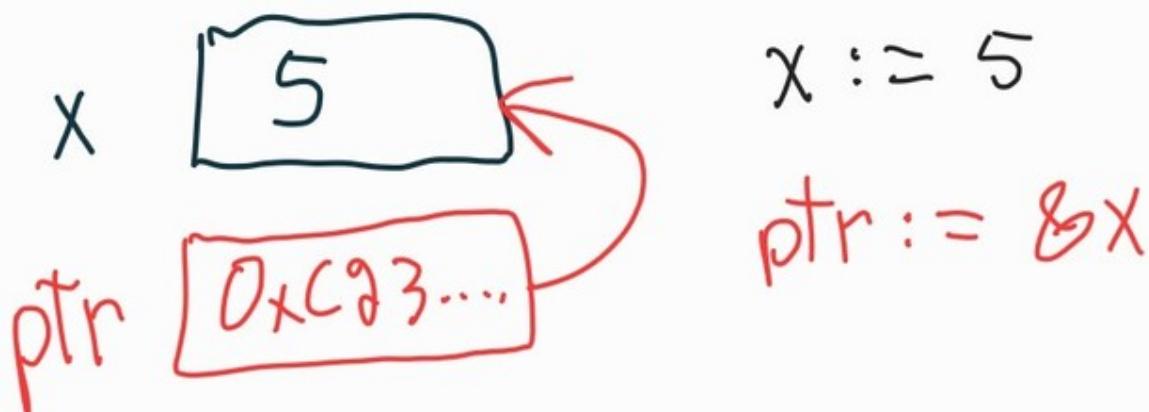
```
var x int = 5  
ptr := &x
```

Dans ce cas, `ptr` est un pointeur vers `x`, car il contient l'adresse mémoire de `x`.

Le type de `ptr` dans le cas ci-dessus est `*int`, car c'est un pointeur vers un entier. Le symbole `*` avant un type est utilisé pour désigner un pointeur vers ce type.

Pour obtenir la valeur stockée à l'adresse mémoire indiquée, ou pour "déréférencer" le pointeur, vous pouvez utiliser l'opérateur `*` avant le pointeur, comme `*ptr`. Par exemple :

```
fmt.Println(*ptr) // Cela imprimera : 5
```



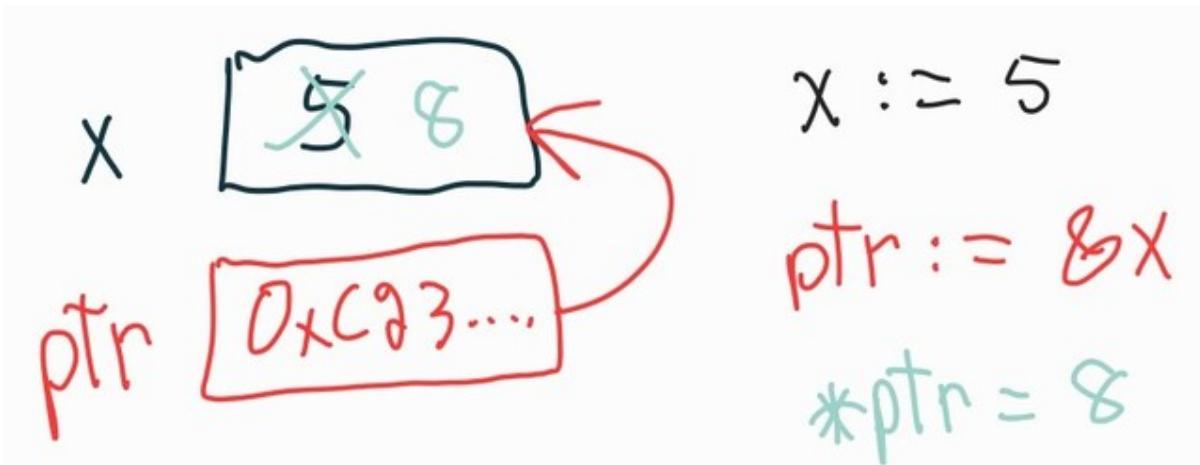
Pointeur vers int

Vous pouvez également utiliser des pointeurs pour changer la valeur stockée à une certaine adresse mémoire :

```
*ptr = 8
```

```
fmt.Println(x) // Cela imprimera : 8
```

Dans ce cas, en modifiant `*ptr`, nous changeons en fait la valeur de `x` puisque `ptr` contient l'adresse de `x`.



Modification de la valeur pointée

On peut également imprimer la valeur contenue dans la variable `ptr`. Puisque c'est un pointeur, nous obtiendrons une adresse mémoire, qui dans ce cas-ci est l'adresse (ou l'emplacement mémoire) où est située la variable `x`.

```
fmt.Println(ptr) // Cela imprimera une valeur semblable à :  
0xc00012e010
```

La valeur imprimée ne sera pas toujours la même. Ça dépendra de l'endroit exact où la variable a été allouée lors de l'exécution du programme, qui varie selon différents facteurs, qui dépendent en partie du système d'exploitation. Nous reviendrons sur les modèles d'allocation de mémoire un peu plus loin.

Faites attention, la valeur zéro d'une variable non initialisée de type pointeur est `nil`, et si vous essayez de déréférencer le pointeur, Go interrompra l'exécution avec une panique au moment de l'exécution. **Assurez-vous toujours que vos pointeurs sont initialisés avant utilisation.**

# Section 2 : Appels de fonctions et types de paramètres

En Go, tous les arguments de fonction sont passés par valeur. Cela signifie que la fonction reçoit une copie de chaque argument, et toute modification n'est pas reflétée dans la variable originale. Les pointeurs peuvent offrir une solution à cette situation, puisqu'en passant l'adresse mémoire de la variable à la fonction, elle peut modifier la variable d'origine.

Reprendons un exemple du chapitre précédent :

```
package main

import "fmt"

func swap2(x int, y int) {
    temp := x
    x = y
    y = temp
}
```

Cette fonction manipule les valeurs des variables `x` et `y`, qui contiennent une copie des valeurs données en paramètre lors de l'appel de la fonction. Si, par exemple, la fonction est appelée de cette façon :

```
a, b := 2, 5
fmt.Println(a, b)
swap2(a, b)
fmt.Println(a, b)
```

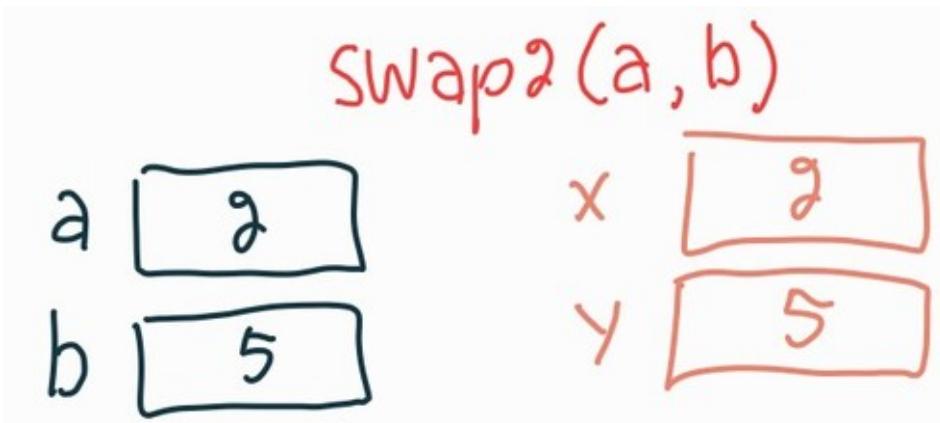


Diagramme de la fonction swap2 - partie 1

Alors **x** contient une copie de la valeur de la variable **a**, et **y** contient une copie de la valeur de la variable **b**. Donc modifier **x** et **y** n'affectent pas **a** et **b** parce qu'elles sont situées à des emplacements mémoire (des adresses) différents. Les variables **x** et **y** sont locales à la fonction **swap2**, tandis que **a** et **b** ne sont pas du tout accessibles à l'intérieur de **swap2**.

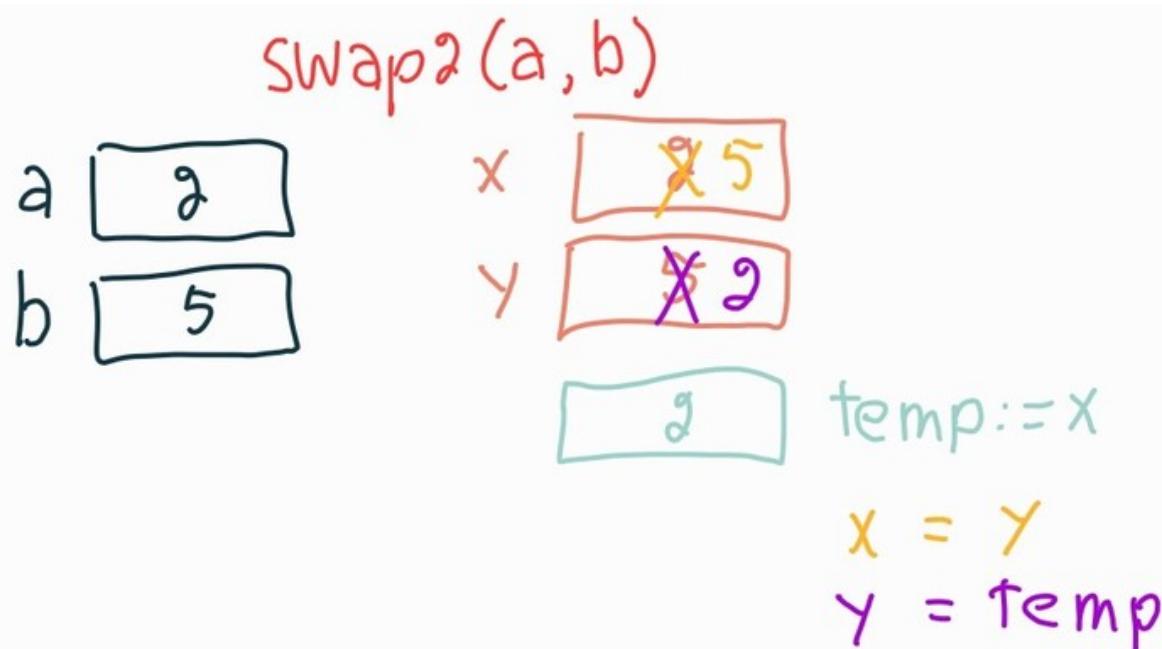


Diagramme de la fonction swap2 - partie 2

C'est ce qu'on appelle le **passage de paramètres par valeurs**. Les variables **a** et **b** sont passées par valeur à la fonction **swap2**.

## Passage par pointeurs

La fonction **swap3** est une modification de **swap2**. À la place de passer les paramètres par valeur, on passe les adresses des variables **a** et **b** à la fonction, donc on passe **a** et **b** par

pointeurs. En réalité, on passe les adresses de `a` et `b` par valeurs à `swap3`, et on utilise ces valeurs (les addresses), pour accéder aux emplacements mémoires qui contiennent `a` et `b`.

```
func swap3(xPtr *int, yPtr *int) {  
    temp := *xPtr  
    *xPtr = *yPtr  
    *yPtr = temp  
}
```

Les paramètres ont été renommés `xPtr` et `yPtr` pour faciliter la compréhension, mais il n'est pas nécessaire de toujours inclure `Ptr` dans le nom des variables qui sont pointeurs.

Si on appelle la fonction comme ceci :

```
a, b := 2, 5  
fmt.Println(a, b)  
swap3(&a, &b)  
fmt.Println(a, b)
```

La sortie sera :

```
2 5  
5 2
```

Dans un premier temps, les variables `xPtr` et `yPtr` sont initialisées pour pointer vers les variables `a` et `b` respectivement. Donc, `xPtr` contient l'adresse de `a`, et `yPtr` contient l'adresse de `b`. Ensuite, la variable `temp` est initialisée avec la valeur de `a`, en déréférençant `xPtr` avec l'astérisque `*`. Donc `temp` a la valeur 2.

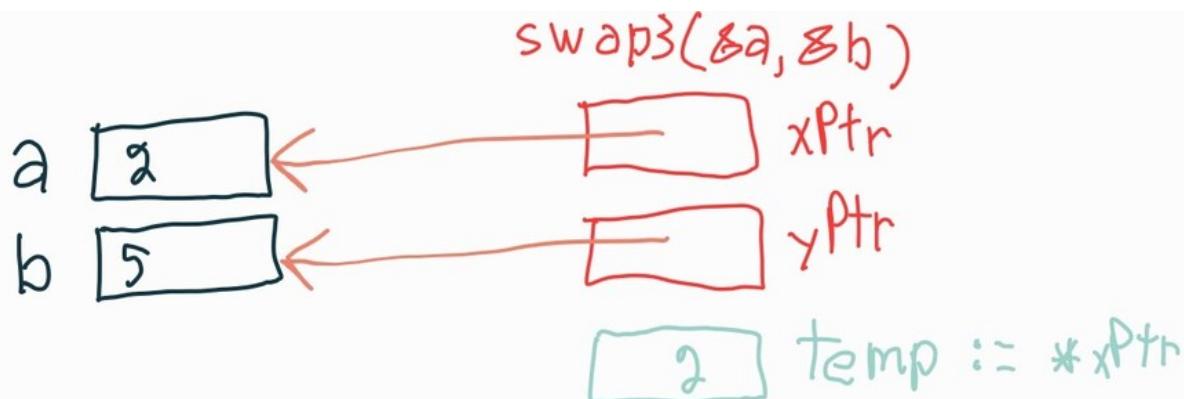


Diagramme fonction `swap3` - partie 1

Ensuite, on prend la valeur de `*yPtr` et on la place dans `*xPtr`, ce qui revient à placer la valeur de `b` dans `a`. La nouvelle valeur de `a` est 5. Ensuite, la valeur de `temp` est placée dans `*yPtr`, ce qui revient à placer 2 dans `b`.

Quand on déréfère `yPtr` avec l'astérisque `*`, c'est comme si on suivait le pointeur à partir de `yPtr` vers `b`. Dans ce cas-ci, on a un accès direct à l'emplacement mémoire qui contient la variable `b`, sans pouvoir accéder directement à cette variable à l'intérieur de la fonction.

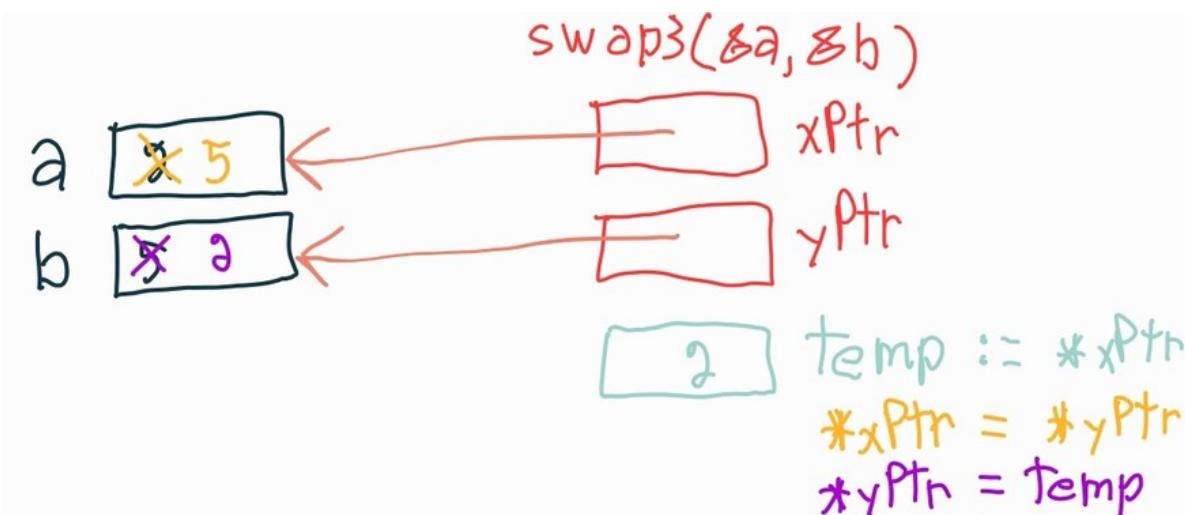


Diagramme fonction swap3 - partie 2

En termes de quand vous devriez utiliser des pointeurs, ils sont utiles pour :

1. Lorsque vous devez modifier l'état d'une variable et que vous avez besoin que ce changement soit visible en dehors de la portée de la fonction actuelle.

- La fonction `Scnf`, présentée dans la prochaine section, demande l'utilisation de pointeurs dans ses paramètres

2. Si vous avez une grande quantité de données et que vous devez la passer à différentes fonctions. Il est plus efficace d'envoyer un pointeur vers ces données plutôt que de copier toutes les données si on les passe par valeur.

- Nous verrons des exemples de cette situation plus loin, lors de l'utilisation des tableaux et des tranches.

# Section 3 : Lire des valeurs avec Scnf

La fonction `Scnf` est la contrepartie à la fonction `Printf`: au lieu d'imprimer, ou d'écrire, des valeurs sur la sortie standard (`stdout`), `Scnf` lit des valeurs sur l'entrée standard (`stdin`). Les fonctions `Sscanf` et `Fscanf` sont semblables à `Scnf`, mais lisent à partir, respectivement, d'une chaîne de caractères et d'un fichier.

Des alternatives à `Scnf` vont être présentées dans la section suivante. `Scnf` est un peu capricieuse, donc il faut faire attention quand on l'utilise. Cette fonction est basée sur la fonction `scanf` du langage C.

## Exemple 1

Voici un programme qui lit le nom de l'utilisateur sur le `stdin`, suivi de son âge.

```
package main

import "fmt"

func ex1() {
    var name string
    var age int

    fmt.Print("Veuillez entrer votre nom : ")
    _, err := fmt.Scanf("%s", &name)

    if err != nil {
        fmt.Println(err)
    }

    fmt.Print("Veuillez entrer votre âge : ")
    _, err = fmt.Scanf("%d", &age)

    if err != nil {
        fmt.Println(err)
    }
}
```

```
    fmt.Printf("Bonjour %s, vous avez %d ans.\n", name, age)  
}
```

Scnf retourne 2 valeurs : un entier qui contient le nombre d'éléments *scannés* correctement, et une erreur, qui pourrait être nil s'il n'y a pas d'erreurs. Dans cet exemple, on ignore la première valeur rentrée en utilisant \_ à la place d'une variable. Nous allons utiliser cette valeur dans d'autres exemples plus tard. Après chaque appel, on vérifie s'il y a des erreurs ou non en comparant err à nil. Si err n'est pas nil, alors il y a une erreur et on écrit cette erreur sur le *stdout*.

Le premier paramètre donné à Scnf est le format, similaire au format donné à la fonction Printf. Le deuxième paramètre est l'endroit où l'on veut que la valeur *scannée* soit placée. Nous devons donner l'adresse d'une variable, ou un pointeur, sinon, comme on a vu plus tôt, les changements apportés à la variable ne seraient pas visibles à l'extérieur de la fonction. C'est pourquoi on donne l'adresse de la variable name dans le premier cas, et de la variable age dans le deuxième, en utilisant l'opérateur &.

## Exécution 1

La sortie dépendra évidemment des valeurs entrées par l'utilisateur. La sortie, incluant les valeurs Denis et 22 entrées sur les 2 premières lignes, ressemblera à ceci :

```
Veuillez entrer votre nom : Denis  
Veuillez entrer votre âge : 22  
Bonjour Denis, vous avez 22 ans.
```

Notez que le caractère \n n'est pas inclus dans la valeur de name même si on doit peser sur enter après avoir entré le nom. La fonction Scnf accepte tous les caractères jusqu'à ce qu'elle rencontre la fin de la ligne. C'est le même processus pour l'âge, sauf que les caractères entrés sont convertis en un nombre entier parce que le format %d a été spécifié.

## Exécution 2

Si on inverse les deux valeurs, on obtient :

```
Veuillez entrer votre nom : 22  
Veuillez entrer votre âge : Denis  
expected integer  
Bonjour 22, vous avez 0 ans.
```

`Scnf` ne peut pas savoir si 22 est un nom approprié ou non, le format `%s` lui demande tout simplement d'accepter une chaîne de caractères, donc la chaîne "22" est acceptée. Il faudra utiliser du code supplémentaire pour valider le contenu de chaînes de caractères (à voir plus tard).

Mais pour l'âge, la chaîne "Denis" ne peut définitivement pas être convertie en nombre entier, donc on obtient l'erreur `expected integer`. Quand une erreur de la sorte arrive, la variable pointée par le paramètre de `Scnf`, dans le cas-ci la variable `age`, ne sera pas modifiée, donc la valeur écrite à la fin sera la valeur que la variable avait avant l'appel à `Scnf`.

### Exécution 3

Qu'arrive-t-il si on n'entre aucun caractère (à part le `enter`) ?

```
Veuillez entrer votre nom :  
unexpected newline  
Veuillez entrer votre âge :  
unexpected newline  
Bonjour , vous avez 0 ans.
```

On obtient l'erreur `unexpected newline`. `Scnf` ne modifiera pas les variables `name` et `age` à travers leurs pointeurs si les valeurs entrées sont vides, donc la sortie inclut la valeur que ces variables avaient les appels à `Scnf`. Puisque ces variables n'avaient pas été explicitement initialisées, elles ont leurs valeurs zéros, c'est-à-dire "" et 0 respectivement.

### Exécution 4

Une particularité de l'interaction avec le `stdin` est qu'il faut entrer un changement de ligne pour que les caractères entrés soient traités par `Scnf`. Cette fonction va alors lire chaque caractère entré et va essayer de les faire correspondre au format donné à la fonction. Quand `Scnf` rencontre un caractère blanc (un *whitespace character*, comme un espace ' ', un changement de ligne '\n', une tabulation '\t'), ça indique la fin d'un format. Par exemple, la chaîne "1 100" représente 2 nombres pour `Scnf` (*un* et *cent*), et non pas le nombre *mille cent*.

Dans notre exemple, on peut entrer les deux valeurs sur la même ligne. Le premier appel à `Scnf` va lire jusqu'à l'espace entre le 's' et le premier '2', et le deuxième appel à `Scnf` va lire le nombre 22 jusqu'au changement de ligne. Ça va fonctionner, mais la sortie sera moins belle.

```
Veuillez entrer votre nom : Denis 22  
Veuillez entrer votre âge : Bonjour Denis, vous avez 22 ans.
```

Des exemples avec une meilleure validation des entrées seront présentés un peu plus loin dans la prochaine section.

## Exécution 5

Quand `Scanf` essaie de lire un nombre entier, la fonction va essayer de lire des chiffres, et quand un caractère qui n'est pas un chiffre est rencontré, ça va terminer la lecture du nombre entier. En général, `Scanf` va essayer de lire autant de caractères possible pour les faire correspondre au format.

```
Veuillez entrer votre nom : Denis  
Veuillez entrer votre âge : 2.2  
Bonjour Denis, vous avez 2 ans.
```

Dans ce cas-ci, `Scanf` arrête au point, donc l'âge est 2. `Scanf` n'est pas parfait, mais elle a basée sur la fonction `scanf` du langage de programmation C. Le but des développeurs du langage Go était de copier le comportement du langage C dans ce cas-ci, pour être compatible.

## Exemple 2

Cet exemple est semblable au précédent, mais le nom et l'âge doivent être entrés sur la même ligne, séparés par un espace. Il n'y a qu'un seul appel à `Scanf` ici parce que le format est "%s %d", qui demande d'avoir une chaîne de caractères suivie d'un nombre entier sur la même ligne.

```
func ex2() {  
    var name string  
    var age int  
  
    fmt.Println("Veuillez entrer votre nom suivi de votre âge, séparés  
    par un espace : ")  
    n, err := fmt.Scanf("%s %d", &name, &age)  
  
    if err != nil {  
        fmt.Println(err)  
    }  
  
    if n != 2 {  
        fmt.Println("Vous devez entrer votre nom suivi de votre,
```

```
    séparés par un espace")
    return
}

fmt.Printf("Bonjour %s, vous avez %d ans.\n", name, age)
}
```

Une autre différence dans cet exemple est la vérification du nombre d'éléments lus correctement, ici placé dans la variable `n`. Dans cet exemple, `n` devrait être 2. L'utilisation de `n` donne une façon différente de détecter les erreurs. En pratique, on utilise soit `err`, soit `n`, selon les besoins, mais pas nécessairement les deux à la fois.

## Exécution 1

Entrée : Denis 22\n

```
Veuillez entrer votre nom suivi de votre âge, séparés par un espace :
Denis 22
Bonjour Denis, vous avez 22 ans.
```

## Exécution 2

Entrée : Denis\n

```
Veuillez entrer votre nom suivi de votre âge, séparés par un espace :
Denis
newline in input does not match format
Vous devez entrer votre nom suivi de votre, séparés par un espace
```

## Exécution 3

Entrée : Denis abc\n

```
Veuillez entrer votre nom suivi de votre âge, séparés par un espace :
Denis abc
expected integer
Vous devez entrer votre nom suivi de votre, séparés par un espace
```

# Section 4 : Validation des entrées

Étant donné que `Scanf` est un peu capricieuse, plusieurs personnes préfèrent utiliser des alternatives pour l'entrée de valeurs. Les exemples de la section précédente utilisaient le `stdin`, mais les exemples seraient presque identiques si on utilisait `Sscanf` ou `Fscanf`. Cette section inclut quelques alternatives. Le chapitre sur la gestion des fichiers contiendra plus d'exemples.

## Lire une ligne complète jusqu'à la fin

Il est préférable d'utiliser un `Reader` sur le `stdin` que d'utiliser `Scanf` pour lire une ligne complète, comme démontré dans l'exemple suivant.

On commence par définir une fonction nommée `ReadLine`, qui lit une ligne de texte sur le `stdin` et la retourne sans inclure le `\n` à la fin de la ligne. La fonction retourne aussi une erreur s'il y en a une, ou sinon `nil` s'il n'y en a pas. La fonction `section4a` plus loin démontre comment utiliser `ReadLine` en copiant l'exemple de la section précédente.

```
func ReadLine() (string, error) {
    scanner := bufio.NewScanner(os.Stdin)
    if scanner.Scan() {
        return scanner.Text(), nil
    } else {
        err := scanner.Err()
        if err != nil {
            return "", err
        } else {
            return "", io.EOF
        }
    }
}
```

La fonction `ReadLine` est une fonction utilitaire qui lit une ligne d'entrée de l'utilisateur depuis le terminal. La fonction `bufio.NewReader(os.Stdin)` initialise un nouveau scanner sur le `os.Stdin`, le flot d'entrée standard. `scanner.Scan()` avance le scanner jusqu'à la prochaine ligne. Si `Scan` renvoie `false`, il a été impossible de lire une ligne de texte, soit parce qu'il y a eu une erreur, soit parce qu'on a atteint la fin du fichier. Nous vérifions alors si une erreur est survenue. Si c'est le cas, nous renvoyons l'erreur. Sinon, nous renvoyons le symbole

représentant la fin du fichier `io.EOF` (*End-Of-File*). Si tout s'est bien passé, `Scan` retourne `true`, et on retourne la ligne de texte qui a été lue avec `scanner.Text()`.

`os.Stdin` est de type `*os.File`, donc `os.Stdin` est un pointeur vers un fichier. Un fichier très spécial, en lecture seule, qui correspond à ce que l'utilisateur tape sur le terminal, mais du point de vue du système d'exploitation, c'est tout de même un fichier. Si jamais le `os.Stdin` est fermé, ce qui ne devrait normalement pas arriver, le scanner recevra un `io.EOF`, et il faut le gérer correctement. Le chapitre 4 sur la gestion des fichiers décrira le `io.EOF` plus en détails.

Ensuite, la fonction `section4a`:

```
func section4a() {
    fmt.Print("Veuillez entrer votre nom : ")
    name, err := ReadLine()

    if err != nil {
        fmt.Println(err)
    }

    fmt.Print("Veuillez entrer votre âge : ")
    ageStr, err := ReadLine()

    if err != nil {
        fmt.Println(err)
    }

    age, err := strconv.Atoi(ageStr)

    if err != nil {
        fmt.Println(err)
    }

    fmt.Printf("Bonjour %s, vous avez %d ans.\n", name, age)
}
```

Cette fonction fait les choses suivantes :

1. Demande à l'utilisateur de saisir son nom en utilisant `fmt.Print`.
2. Lit le nom entré en utilisant `ReadLine`.

3. Si une erreur se produit lors de la lecture, imprime l'erreur.
4. Demande à l'utilisateur d'entrer son âge.
5. Lit l'âge entré, à nouveau en utilisant `ReadLine`.
6. Si une erreur se produit pendant la lecture, imprime l'erreur.
7. Convertit l'âge entré (qui est une chaîne de caractères) en un nombre entier à l'aide de `strconv.Atoi`.
8. Si la conversion provoque une erreur, imprime cette erreur.
9. Enfin, salue l'utilisateur en utilisant `fmt.Printf`, en insérant le nom et l'âge qu'ils ont entrés dans la chaîne.

**Note:** Gérez toujours correctement les erreurs dans votre code. Dans ce cas, après une erreur, la fonction continue l'exécution même si les données sont peut-être incorrectes ou absentes. Il est généralement conseillé de renvoyer ou de traiter les erreurs de manière à ce qu'elles ne causent pas de problèmes imprévus plus tard dans l'exécution du programme.

Une façon différente de gérer les erreurs lors de l'entrée d'un nombre entier suit après les exemples d'exécution.

## Exécution 1

Avec des entrées sans erreurs, la sortie sera la même.

```
Veuillez entrer votre nom : Denis
Veuillez entrer votre âge : 22
Bonjour Denis, vous avez 22 ans.
```

## Exécution 2

Avec des entrées vides, il n'y aura pas d'erreur pour le nom parce que la chaîne de caractères vide est une chaîne valide. Par contre, une chaîne vide ne peut pas être convertie en nombre entier, donc il y a une erreur pour l'âge.

```
Veuillez entrer votre nom :
Veuillez entrer votre âge :
strconv.Atoi: parsing "": invalid syntax
Bonjour , vous avez 0 ans.
```

On pourrait définir une autre version de `ReadLine`, qui n'accepterait pas les chaînes vides, qu'on pourrait nommer par exemple `ReadLineNonEmpty` ou `ReadNonEmptyLine`.

## Lire un entier

La fonction `ReadInt` utilise la fonction `ReadLine` de la section précédente et s'assure que l'entrée est vraiment un nombre entier. Elle retourne une erreur seulement si le nombre d'essais maximum a été atteint. Il serait possible de faire à peu près la même chose, mais en utilisant `Scnf` à la place. Le comportement ne serait pas nécessairement le même dans tous les cas parce que `Scnf` ne lit pas automatiquement jusqu'à la fin de la ligne, comme `ReadLine` le fait.

```
func ReadInt(nAttempts int) (int, error) {
    for i := 0; i < nAttempts; i++ {
        inputStr, err := ReadLine()
        if err != nil {
            fmt.Println("Erreur lors de la lecture. Essayez de nouveau.")
        } else {
            inputNum, err := strconv.Atoi(inputStr)
            if err != nil {
                fmt.Println("Ceci n'est pas un nombre entier. Essayez de nouveau.")
            } else {
                return inputNum, nil
            }
        }
    }
    return 0, fmt.Errorf("échec : nombre limite de %d essai(s) atteint", nAttempts)
}
```

Cette fonction tente de lire un entier depuis l'entrée standard (`stdin`). Le nombre de tentatives permises pour que l'utilisateur fournit un nombre entier valide est défini par le paramètre `nAttempts`.

Voici une interprétation pas à pas :

1. La fonction initialise une boucle `for` qui continue pendant le nombre spécifié de tentatives

(nAttempts).

- À l'intérieur de la boucle, elle appelle la fonction `ReadLine()` pour lire une ligne à partir de l'entrée standard.

```
inputStr, err := ReadLine()
```

`ReadLine` est la même fonction que dans l'exemple précédent.

- Si une erreur s'est produite lors de la lecture de la ligne (par exemple, si une erreur d'E/S s'est produite), la fonction affiche un message d'erreur et passe à la prochaine itération.

```
fmt.Println("Erreur lors de la lecture. Essayez de nouveau.")
```

- Si aucune erreur ne s'est produite lors de la lecture, elle tente de convertir la ligne en entier à l'aide de `strconv.Atoi`.

```
inputNum, err := strconv.Atoi(inputStr)
```

- Si une erreur s'est produite lors de la conversion (par exemple, si la ligne n'est pas un entier valide), la fonction imprime un message d'erreur informatif et passe à la prochaine itération.

```
fmt.Println("Ceci n'est pas un nombre entier. Essayez de nouveau.")
```

- Si la conversion a réussi, la fonction retourne l'entier et une erreur `nil`.

```
return inputNum, nil
```

- Si la fonction n'a pas pu lire un entier valide après `nAttempts` tentatives, elle renvoie `0` et un message d'erreur indiquant que la limite des tentatives a été atteinte.

```
return 0, fmt.Errorf("échec : nombre limite de %d essai(s) atteint", nAttempts)
```

Donc, cette fonction essaie de lire un ensemble de caractères depuis l'entrée standard, convertit cet ensemble de caractères en un entier, et le renvoie. Si elle n'est pas en mesure de lire l'entrée correctement, ou si les caractères lus ne représentent pas un entier, elle renvoie une erreur dès que le nombre limite d'essais a été atteint.

# Exercices

1. Écrire et tester la fonction `ReadNonEmptyLine`, qui, tant que l'entrée est la chaîne vide ou qu'il y a une erreur lors de la lecture de la ligne, refuse cette entrée et redemande d'entrer une chaîne. Vous pouvez vous baser sur la fonction `ReadInt`, présentée dans la prochaine section.
2. Créez une copie de la fonction `section4a` nommée `section4b` qui a fait la même chose, mais qui utilise les fonctions `ReadNonEmptyLine` et `ReadInt` pour valider les entrées.
3. Faites une version de `ReadInt` qui utilise `Scarf` à la place de `.ReadLine`.
4. Faites une autre version de `ReadInt` (à partir de l'originale ou de celle de l'exercice précédent), qui, lorsque `nAttempts` est plus petit que 1, va toujours redemander une entrée, donc il n'y aura pas de limite au nombre d'essais. Lorsque `nAttempts` est au moins 1, la fonction doit faire la même chose qu'avant.
5. Faites une version de `ReadNonEmptyLine` qui accepte en paramètre un nombre d'essais, comme pour `ReadInt`. Gérez le nombre d'essais de la même façon que l'exemple précédent.

# Chapitre 3 : Tableaux et tranches

En Go, un tableau est une collection de valeurs de même type qui sont stockées de manière contiguë en mémoire. La taille d'un tableau est fixe, c'est-à-dire que vous devez définir la taille d'un tableau au moment de sa déclaration.

Voici comment nous définissons un tableau en Go :

```
var monTableau [10] int
```

Dans l'exemple ci-dessus, `monTableau` est un tableau de dix entiers. `var` est le mot-clé utilisé pour déclarer une variable. `monTableau` est le nom du tableau, `[10]` indique la taille du tableau et `int` est le type de données que le tableau peut stocker.

Pour initialiser un tableau au moment de la déclaration, vous pouvez faire comme suit :

```
monTableau := [5] int{1, 2, 3, 4, 5}
```

Dans cette déclaration, `monTableau` est un tableau de 5 entiers. Le tableau est initialisé avec les valeurs 1, 2, 3, 4, 5.

Vous pouvez également créer un tableau sans spécifier sa taille et laisser Go déterminer la taille en fonction du nombre d'éléments que vous fournissez :

```
monTableau := [...] int{1, 2, 3, 4, 5}
```

Ici, `monTableau` est aussi un tableau de 5 entiers, tout comme dans l'exemple précédent. Cependant, nous avons utilisé `...` au lieu de spécifier la taille du tableau, et Go a automatiquement déduit la taille à partir du nombre d'éléments fournis.

# Section 1 : Valeur ou pointeur ?

En Go, les tableaux sont des types valeur, pas des pointeurs, contrairement à d'autres langages de programmation.

Cela signifie que lorsqu'un tableau est assigné à une nouvelle variable, une copie du tableau original est créée. De la même manière, passer un tableau à une fonction entraînera la création d'une copie de ce tableau dans la fonction. Les modifications apportées à la copie ne modifieront pas le tableau original.

Voici un exemple pour illustrer ceci :

```
package main

import "fmt"

// La fonction modifierTentative modifie le tableau qu'elle reçoit.
func modifierTentative(tab [3]int) {
    tab[0] = 100
}

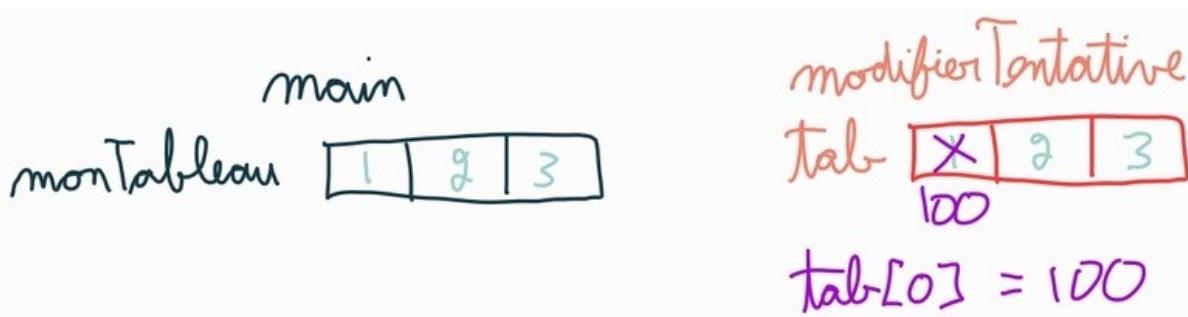
func main() {
    monTableau := [3]int{1, 2, 3}

    modifierTentative(monTableau)

    fmt.Println(monTableau) // Sortie : [1 2 3]

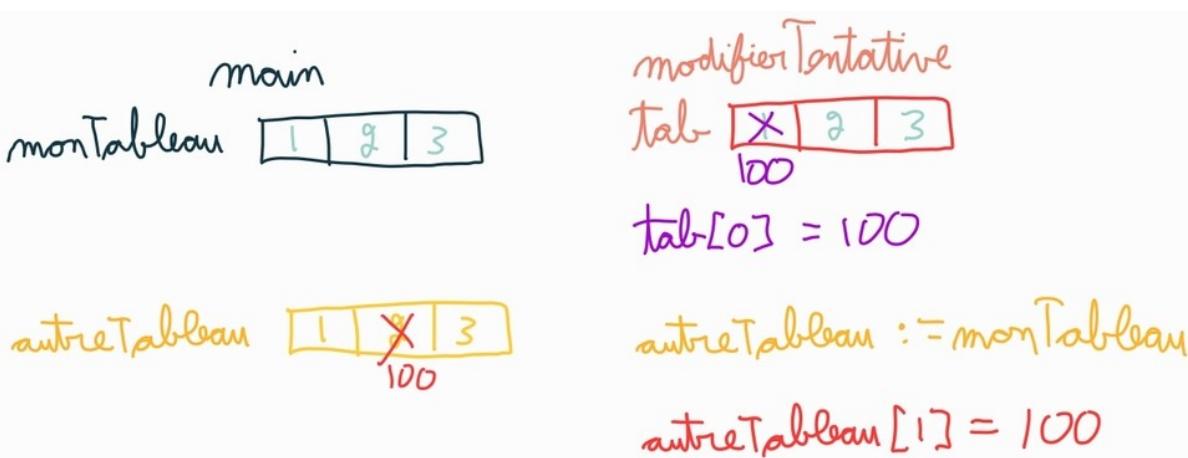
    autreTableau := monTableau
    autreTableau[1] = 100

    fmt.Println(monTableau) // Sortie : [1 2 3]
}
```



ch03s01-Tableau 1a.jpg

Dans l'exemple ci-dessus, même si la fonction `modifierTentative` a tenté de modifier le premier élément du tableau `tab`, cela n'a pas eu d'impact sur `monTableau` dans la fonction `main`. C'est pourquoi lorsque nous imprimons `monTableau` après l'appel à `modifierTentative`, les valeurs restent inchangées.



ch03s01-Tableau-1b.jpg

Le même principe s'applique si on crée un autre tableau à partir de `monTableau`. Quand on fait `autreTableau := monTableau`, on crée un autre espace mémoire pour `autreTableau` du même type et de même longueur que `monTableau`, et le contenu de `monTableau` est copié dans `autreTableau`.

Si vous voulez que les modifications se répercutent sur le tableau d'origine, vous devriez utiliser des tranches (slice) ou passer un pointeur vers le tableau d'origine. Les tranches seront présentées un peu plus loin. Un exemple avec un pointeur vers un tableau suit.

## La longueur du tableau fait partie du type

Le fait qu'en Go, un tableau est de type valeur, implique que, en plus du type des éléments du tableau, la longueur du tableau fait partie du type du tableau. Il n'est pas possible, par

exemple, de passer en argument un tableau de type [4]int à une fonction qui demande un tableau de type [3]int.

Modifiez l'exemple précédent en modifiant soit la taille du tableau dans le main, soit la longueur du tableau dans la déclaration du paramètre de la fonction, et vous obtiendrez une erreur du genre cannot use monTableau (variable of type [4]int) as [3]int value in argument to modifierTentative. Le programme ne compilera pas parce que les types ne correspondent pas. Puisque les valeurs du tableau donné en argument doivent être copiées dans le tableau local à la fonction, non seulement les valeurs doivent être du même type, mais il doit y avoir le même nombre de valeurs (c.-à-d. la même longueur de tableau) dans les deux tableaux.

Pour passer un tableau à une fonction, on utilise soit un pointeur vers un tableau, ou plus couramment une tranche, qui contient un pointeur vers un tableau.

## Pointeur vers un tableau

L'exemple suivant est une modification de l'exemple précédent, sauf que la fonction accepte un pointeur vers le tableau plutôt qu'une copie du tableau.

```
package main

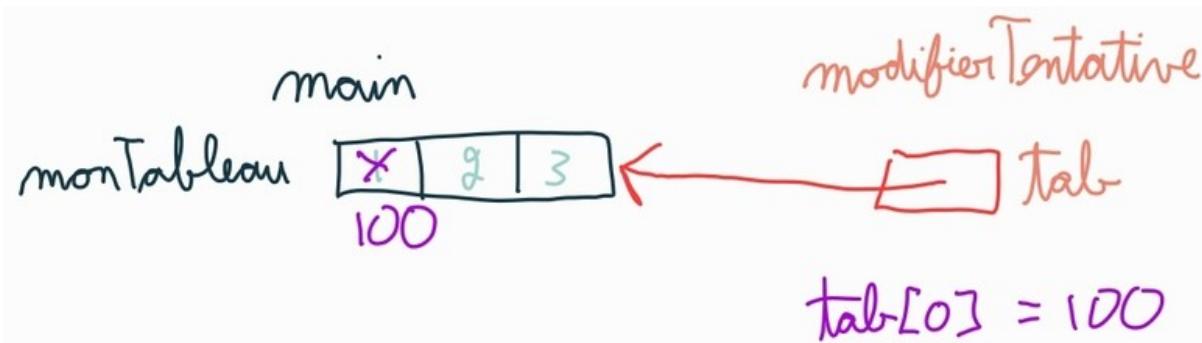
import "fmt"

// La fonction modifierTentative modifie le tableau qu'elle reçoit.
func modifierTentative(tab *[3]int) {
    (*tab)[0] = 100
}

func main() {
    monTableau := [3]int{1, 2, 3}

    modifierTentative(&monTableau)

    fmt.Println(monTableau) // Output : [100 2 3]
}
```



ch03s01-Tableau-2.jpg

Dans cet exemple, `modifierTentative` accepte un pointeur vers un tableau de 3 entiers (type `*[3]int`). À l'intérieur de la fonction, nous utilisons `(*tab)[0]` pour accéder au premier élément du tableau pointé par `tab` et pour le modifier. Les parenthèses sont nécessaires ici parce que les crochets `[]` ont préséance sur l'astérisque `*`, donc il faut déréférencer le pointeur vers un tableau avant d'essayer d'accéder à ses éléments.

Dans la fonction `main`, nous passons l'adresse du tableau (`&monTableau`) à la fonction `modifierTentative`.

Maintenant, les modifications apportées à l'intérieur de `modifierTentative` sont répercutées sur le tableau `monTableau` dans `main`. Ainsi, lorsque nous imprimons `monTableau` après l'appel à `modifierTentative`, il affiche le tableau modifié :

`[100 2 3]`

# Section 2 : Tranches

En Go, lorsque vous utilisez un tableau statique et passez un pointeur de ce tableau à une fonction, vous devez spécifier la taille du tableau. C'est parce que les tableaux de différentes tailles sont considérés comme des types différents en Go. En d'autres mots, la taille du tableau fait partie de son type.

Cependant, si vous voulez une fonction qui puisse accepter une collection d'entiers de n'importe quelle taille, vous devriez utiliser une **tranche** (*slice*) au lieu d'un tableau. Les tranches sont des abstractions construites autour des tableaux et sont plus flexibles. Les tranches sont une structure de données de la catégorie des **tableaux dynamiques**, qui ressemblent aux `ArrayList` en Java.

Voici une modification de l'exemple précédent, mais avec des tranches :

```
package main

import "fmt"

// La fonction modifierTentative modifie la tranche qu'elle reçoit.
func modifierTentative(s []int) {
    s[0] = 100
}

func main() {
    maTranche := []int{1, 2, 3}

    modifierTentative(maTranche)

    fmt.Println(maTranche) // Output : [100 2 3]
}
```

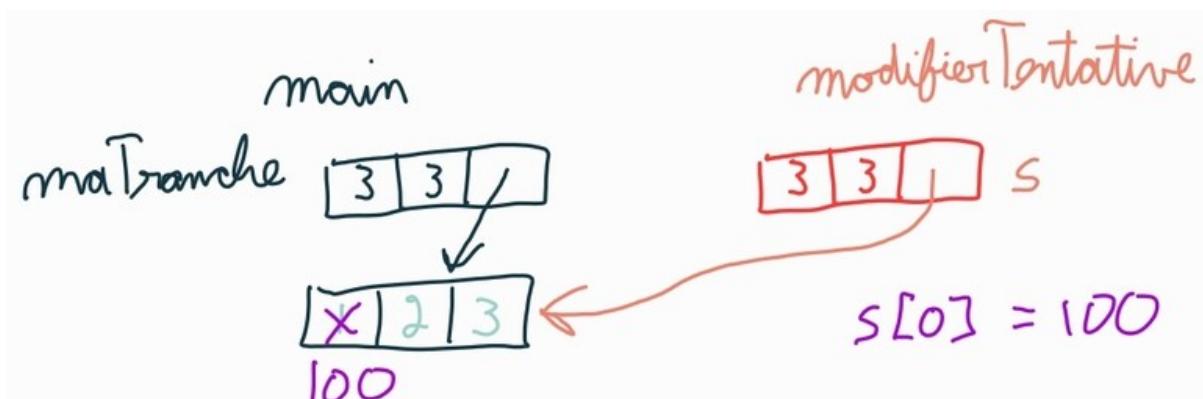
Dans cet exemple, `modifierTentative` accepte une tranche d'entiers (`[]int`). À l'intérieur de la fonction, nous utilisons `s[0]` pour accéder au premier élément de la tranche `s` et pour la modifier.

Et ici, contrairement aux tableaux, le passage d'une tranche à une fonction ne crée pas de nouvelle copie des données du tableau. Au lieu de cela, la fonction modifie la tranche

originale. C'est pourquoi lorsque nous imprimons `maTranche` après l'appel à `modifierTentative`, ça affiche la tranche modifiée : [100 2 3].

Non seulement la taille d'une tranche est dynamique (c'est pour ça qu'on appelle souvent les structures de ce type des tableaux dynamiques), mais on peut également modifier leur contenu sans avoir besoin d'utiliser un pointeur vers la tranche, comme dans l'exemple précédent.

Voici la représentation mémoire des variables de l'exemple, mais pour comprendre les détails, il va falloir comprendre ce qu'est une structure. Plus de détails sur les tranches seront présentés dans un autre chapitre, après la présentation des structures (`struct`) et des principes de la gestion de la mémoire. Mais en attendant, on peut voir une structure comme étant un regroupement de plusieurs variables ensemble, un peu comme une classe avec ses attributs dans un langage orienté-objet.



ch03s02-Tranche.jpg

Dans une structure représentant une tranche, on a essentiellement 2 nombres entiers, la longueur et la capacité de la tranche, suivie d'un pointeur vers un tableau. La capacité est la longueur du tableau, égale à 3 dans l'exemple, et la longueur est le nombre d'emplacements utilisés dans le tableau, aussi égale à 3 dans l'exemple. Comme mentionné précédemment, les tranches vont être expliquées plus en détails dans un autre chapitre. Le plus important pour l'instant, c'est qu'une tranche contient un pointeur vers un tableau, et lorsque l'on passe une tranche en paramètre à une fonction, on copie la structure, incluant le pointeur vers le tableau, mais on ne copie le tableau lui-même. C'est un peu comme si on passait un pointeur à un tableau comme dans un exemple précédent, mais accompagné de 2 valeurs supplémentaires. Donc quand on modifie le contenu de la tranche avec `s[0] = 100`, on modifie réellement le tableau caché derrière la tranche, qui lui n'est pas copié, mais partagé entre `maTranche` et `s`.

## Taille dynamique

## Avec append

Pour ajouter des éléments à une tranche, on doit utiliser la fonction `append`. Cette fonction accepte une tranche suivie d'un ou plusieurs éléments à ajouter, et retourne la tranche avec les éléments qui ont été ajoutés. Il ne faut surtout pas ignorer la valeur de retour, sinon les changements seront perdus.

```
s := []int{1, 2, 3}
fmt.Println(s)          // sortie : [1 2 3]
s = append(s, 7, 4)
fmt.Println(s)          // sortie : [1 2 3 7 4]
```

## Modification à l'intérieur d'une fonction : exemple 1

Cet exemple sert de référence. C'est un exemple de modification comme déjà fait précédemment. Un élément de la tranche est modifié à l'intérieur de la fonction, et le changement est visible à l'extérieur.

```
func modSlice1(s []int) {
    s[0] = 100
}

func section2a() {
    s := []int{1, 2, 3}
    fmt.Println(s)
    modSlice1(s)
    fmt.Println(s)
    fmt.Println()
}
```

Sortie :

```
[1 2 3]
[100 2 3]
```

## Modification à l'intérieur d'une fonction : exemple 2

Si on modifie une tranche à l'aide de `append` à l'intérieur d'une fonction, les changements ne seront pas visibles à l'extérieur parce que la longueur et la capacité sont des copies locales, donc les changements ne seront pas visibles à l'extérieur de la fonction.

```

func modSlice2(s []int) {
    s = append(s, 100)
}

func section2b() {
    s := []int{1, 2, 3}
    fmt.Println(s)
    modSlice2(s)
    fmt.Println(s)
    fmt.Println()
}

```

Sortie :

```
[1 2 3]
[1 2 3]
```

### Modification à l'intérieur d'une fonction : exemple 3

Pour voir l'effet de `append` à l'extérieur de la fonction, il faut passer un pointeur vers la tranche pour être capable de modifier la structure qui contient la longueur et la capacité de la tranche directement, et non une copie de ces valeurs.

```

func modSlice3(s *[]int) {
    *s = append(*s, 100)
}

func section2c() {
    s := []int{1, 2, 3}
    fmt.Println(s)
    modSlice3(&s)
    fmt.Println(s)
    fmt.Println()
}

```

Sortie :

```
[1 2 3]
[1 2 3 100]
```

## Créer une tranche avec la fonction make

Il est possible d'utiliser la fonction `make` pour créer une tranche de longueur et de capacité variables. La fonction accepte le type à créer, suivi de la longueur et de la capacité : `make([]int, length, capacity)`.

### Modification à l'intérieur d'une fonction : exemple 4

Voici un exemple semblable aux précédents, mais avec la fonction `make` pour la création de la tranche. Dans cet exemple, après l'appel à `make`, la tranche contient 3 éléments, tous égaux à 0, la valeur par défaut pour un `int`. Mais le tableau a de l'espace pour 2 éléments de plus parce que la capacité est 5. Plus de détails sur les algorithmes utilisés pour la gestion de la mémoire par une tranche seront présentés dans un autre chapitre.

```
func modSlice4(s []int) {
    s = append(s, 100)
}

func section2d() {
    s := make([]int, 3, 5)
    s[0] = 4
    s[1] = 1
    s[2] = -5
    fmt.Printf("%d %d %v\n", len(s), cap(s), s)
    modSlice4(s)
    fmt.Printf("%d %d %v\n\n", len(s), cap(s), s)
}
```

Sortie :

```
3 5 [4 1 -5]
3 5 [4 1 -5]
```

### Modification à l'intérieur d'une fonction : exemple 5

Un autre exemple, semblable au précédent, mais utilisant un pointeur vers une tranche comme type du paramètre de la fonction. Notez que la capacité de la tranche augmente automatiquement si nécessaire.

```
func modSlice5(s *[]int) {
    *s = append(*s, 100)
```

```

}

func section2e() {
    s := make([]int, 3, 5)
    s[0] = 4
    s[1] = 1
    s[2] = -5
    fmt.Printf("%d %d %v\n", len(s), cap(s), s)
    modSlice5(&s)
    fmt.Printf("%d %d %v\n", len(s), cap(s), s)
    modSlice5(&s)
    modSlice5(&s)
    fmt.Printf("%d %d %v\n\n", len(s), cap(s), s)
}

```

Sortie :

```

3 5 [4 1 -5]
4 5 [4 1 -5 100]
6 10 [4 1 -5 100 100 100]

```

## Obtenir une sous-tranche d'une tranche

Une sous-tranche est une tranche dérivée d'une autre tranche. On doit spécifier un intervalle avec 2 index, l'index de départ et celui d'arrivée. Voici quelques exemples :

```

func section2f() {
    numbers := []int{0, 1, 2, 3, 4, 5}
    fmt.Println(numbers)
    fmt.Println(numbers[1:6])
    fmt.Println(numbers[1:5])
    sub := numbers[1:]
    fmt.Println(sub)
    numbers[2] = -1
    fmt.Println(numbers)
    fmt.Println(sub)
    fmt.Println(sub[:len(sub)-1])
    fmt.Println(sub[1:2])
}

```

```
    fmt.Println(sub[1])  
}
```

Sortie :

```
[0 1 2 3 4 5]  
[1 2 3 4 5]  
[1 2 3 4]  
[1 2 3 4 5]  
[0 1 -1 3 4 5]  
[1 -1 3 4 5]  
[1 -1 3 4]  
[-1]  
-1
```

Il est important de noter que les sous-tranches partagent le même tableau sous-jacent avec la tranche originale, toute modification de la sous-tranche affectera la tranche originale. Ce comportement est dû à la façon dont les tranches en Go sont conçues, où une tranche se compose d'un pointeur vers le tableau, la longueur de la tranche, et sa capacité.

Lorsque vous traitez de grands ensembles de données, la création de sous-tranches peut économiser de la mémoire, car vous ne dupliquez pas l'ensemble des données. Gardez à l'esprit la caractéristique de partage de tableau des tranches lors de la modification de celles-ci, surtout lors de l'exécution concurrente.

N'oubliez pas de vérifier les limites du tableau lors du découpage, pour éviter une panique d'exécution. Les limites d'une tranche sont inclusives pour l'indice de début et exclusives pour l'indice de fin, c'est-à-dire `slice[start:end]`.

Notez aussi, dans l'exemple précédent, la différence entre les 2 dernières sorties. La dernière sortie est créée en spécifiant seulement une valeur entre les crochets `[]`, donc aucune sous-tranche n'est créée ici, on obtient uniquement l'élément à l'index donné. Tandis que pour l'avant-dernière sortie, on utilise 2 index séparés par `:`, donc on obtient une tranche avec uniquement un élément.

# Section 3 : Chaînes de caractères

En Go, une chaîne de caractères est essentiellement une tranche d'octets en lecture seule. Cela permet aux chaînes Go d'avoir des tailles variables, comme les tranches de n'importe quel type. Si les chaînes étaient construites directement avec des tableaux, on ne pourrait pas mélanger les chaînes de différentes longueurs.

Les chaînes de caractères sont **immuables**. Cela signifie qu'une fois qu'une chaîne est créée, elle ne peut pas être modifiée. Si vous essayez de modifier une chaîne, Go créera une nouvelle chaîne avec les modifications, au lieu de modifier la chaîne originale. Ou si vous essayez de modifier directement un caractère spécifique d'une chaîne, il y aura une erreur.

Voici un exemple démontrant l'immutabilité des chaînes :

```
package main

import "fmt"

func main() {
    s := "bonjour"
    s += " le monde" // Crée une nouvelle chaîne
    fmt.Println(s) // Affiche : "bonjour le monde"

    s[0] = 'B' // ne compilera pas
}
```

Dans l'exemple ci-dessus, `s += " le monde"` n'ajoute pas "*le monde*" à la chaîne originale "*bonjour*". Au lieu de cela, il crée une nouvelle chaîne "*bonjour le monde*" et l'assigne à `s`.

Si vous avez besoin d'un type semblable à une chaîne que vous pouvez modifier, envisagez d'utiliser une tranche d'octets (`[]byte`) ou un `strings.Builder` (voir l'exemple suivant).

Les chaînes Go sont encodées en UTF-8, un seul caractère dans une chaîne Go pourrait être représenté par un ou plusieurs octets. Donc le nombre de caractères dans une chaîne n'est pas égal, en général, au nombre d'octets dans une tranche représentant la même chaîne. C'est en partie pourquoi les chaînes sont immuables en Go.

## Utilisation d'un `strings.Builder`

Voici un exemple très simple d'utilisation d'un `strings.Builder`.

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var builder strings.Builder

    builder.WriteString("Hello")
    builder.WriteString(" ")
    builder.WriteString("World")
    builder.WriteString("!")

    result := builder.String()

    fmt.Println(result)
}
```

Dans cet exemple, `strings.Builder` est utilisé pour faire une concaténation efficace de plusieurs chaînes de caractères. C'est très utile lorsque vous avez besoin de concaténer un grand nombre de chaînes, car il est plus efficace en termes de performance que l'opérateur `+` ou que `fmt.Sprintf`.

Plus d'exemples sur l'utilisation des `strings.Builder` seront présentés plus loin.

# Exercices

Start typing here...

# Chapitre 4 : La gestion des fichiers

Lire ou écrire dans un fichier est très semblable à la lecture à partir du `stdin` et à l'écriture sur le `stdout`. La différence principale est qu'il faut ouvrir et fermer les fichiers correctement avant et après la lecture ou l'écriture des données dans les fichiers. Du point de vue du code, il n'y a pratiquement pas de différences entre lire des données du `stdin` et lire des données d'un fichier ouvert en mode lecture, et réciproquement, écrire des données sur le `stdout` ou dans un fichier ouvert en mode écriture sont presque identiques en termes de code. `os.Stdin` et `os.Stdout` sont de type `*os.File`, donc ils sont des pointeurs vers des fichiers, des fichiers spéciaux du système d'exploitation qui correspondent à l'entrée et à la sortie associées à un terminal.

# Section 1 : Lire le contenu d'un fichier

La façon la plus simple, mais pas nécessairement la plus efficace, de lire le contenu d'un fichier est d'utiliser la fonction `os.ReadFile`. Cette fonction ouvre le fichier, lit son contenu dans une chaîne de caractères, ferme le fichier, et retourne le contenu. Si le fichier est gros, les performances ne seront pas très bonnes parce que tout le fichier sera lu et placé dans une seule chaîne.

## Exemple simple avec `os.ReadFile`

```
package main

import (
    "fmt"
    "os"
    "log"
)

func section1a() {
    content, err := os.ReadFile("allo.txt") // Lecture du contenu du
fichier
    if err != nil {
        log.Fatalf("lecture du contenu du fichier impossible : %s",
err)
    }

    fmt.Println("Contenu du fichier :")
    fmt.Println(string(content))
}
```

Si le fichier existe, la sortie sera `Contenu du fichier :` sur la première ligne, suivi du contenu du fichier sur les lignes suivantes.

Si le fichier n'existe pas, alors il y aura une erreur, `err` ne sera pas `nil`, et le message d'erreur `lecture du contenu du fichier impossible :` sera affiché précédé de la date et l'heure, et suivi de l'erreur elle-même. La date et l'heure viennent de la fonction `log.Fatalf`, qui ajoute ces informations automatiquement. Aussi, cette fonction mettra un terme à l'exécution du programme parce qu'on utilise le niveau de `log Fatal`, qui veut dire qu'on ne peut pas

continuer. Le `f` à la fin de la fonction est similaire au `f` à la fin de `fmt.Printf`, ce qui veut dire que la fonction accepte les formats pour la sortie.

Voir la doc officielle (<https://pkg.go.dev/log#Fatalf>) pour plus de détails sur le paquet `log`.

L'erreur la plus commune sera une erreur d'ouverture du fichier, similaire à `open allo.txt: no such file or directory`. Il faut en général que le fichier soit dans le même dossier que le programme compilé en fichier exécutable (extension `.exe` sous Windows) si on spécifie seulement le nom du fichier à ouvrir sans spécifier de chemin. On peut également spécifier un chemin relatif ou absolu avec un ou plusieurs dossiers séparés par des barres obliques. Le chemin relatif sera à partir du dossier où on exécute le programme.

Si on exécute le programme à travers l'interface graphique de GoLand, ça pourrait être différent, dépendamment de sa configuration. Normalement, le fichier `.go` sera compilé dans un dossier temporaire, et exécuté à partir du dossier du principal du projet. Donc le programme essaiera d'ouvrir un fichier à partir de ce dossier (à moins d'avoir spécifié un chemin absolu).

À partir d'un terminal, on peut compiler le programme à partir du dossier contenant le fichier `.go` avec `go build`, et exécuter le programme dans le terminal avec une commande du genre `./main.exe`. Dans ce cas, l'ouverture d'un fichier se fera relativement au dossier courant.

## Même exemple, mais avec ouverture et fermeture de fichier explicites

```
func section1b() {
    file, err := os.Open("allo.txt") // Ouverture du fichier en mode
                                    // lecture
    if err != nil {
        log.Fatalf("ouverture du fichier impossible : %s", err)
    }

    content, err := io.ReadAll(file) // Lecture du contenu du fichier
    if err != nil {
        log.Fatalf("lecture du contenu du fichier impossible : %s",
                  err)
    }

    fmt.Println("Contenu du fichier :")
    fmt.Println(string(content))
```

```

        err = file.Close()
        if err != nil {
            log.Printf("erreur lors de la fermeture du fichier : %s", err)
        }
    }
}

```

Cet exemple a le même problème que le précédent si le fichier lu est gros. Le but ici est démontrer comment on peut ouvrir et fermer un fichier explicitement. Il existe une autre façon de fermer un fichier en Go, avec l'utilisation de `defer`, mais pour l'instant, nous le faisons de la manière longue.

1. La fonction commence par ouvrir le fichier `allo.txt` en mode lecture avec `os.Open`. Si une erreur se produit (par exemple, si le fichier ne peut pas être trouvé), un message d'erreur sera affiché et le programme s'arrêtera avec `log.Fatalf`.
2. Ensuite, la fonction lit le contenu du fichier avec `io.ReadAll`. Encore une fois, si une erreur se produit pendant la lecture (par exemple, si les données ne peuvent pas être interprétées correctement), un message d'erreur sera affiché et le programme s'arrêtera.
3. Le contenu du fichier (qui est maintenant en mémoire sous forme de tableau de bytes) est converti en chaîne de caractères avec `string(content)` et est ensuite affiché dans la console avec `fmt.Println`.
4. Enfin, la fonction essaie de fermer le fichier avec `file.Close()`. Si une erreur se produit lors de la fermeture du fichier (ce qui est rare, mais peut arriver si par exemple l'espace disque est plein et que le système ne peut pas mettre à jour les métadonnées du fichier), un message d'erreur sera affiché, mais contrairement aux erreurs précédentes, le programme ne s'arrêtera pas dans ce cas.

## Exemple : Compter le nombre de lignes non-vides dans un fichier

Voici une fonction qui accepte le nom d'un fichier en paramètre, et qui compte le nombre de lignes non-vides dans ce fichier. Le nom de fichier peut inclure un chemin relatif ou absolu.

```

func countNonEmptyLines(filename string) {
    file, err := os.Open(filename) // Ouverture du fichier en mode
    lecture
    if err != nil {
        log.Fatalf("ouverture du fichier impossible : %s", err)
    }
    defer file.Close()
    var count int
    for _, line := range content {
        if len(line) > 0 {
            count++
        }
    }
    return count
}

```

```

}

count := 0
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    if len(scanner.Text()) > 0 {
        count++
    }
}

err = scanner.Err()
if err != nil {
    log.Printf("erreur lors de la lecture du fichier : %s", err)
}

fmt.Println("Nombre de ligne non-vide :", count)

err = file.Close()
if err != nil {
    log.Printf("erreur lors de la fermeture du fichier : %s", err)
}
}

```

1. La fonction `countNonEmptyLines` est définie, qui prend un nom de fichier en paramètre.
2. La fonction essaie d'abord d'ouvrir le fichier en lecture. S'il y a une erreur (comme le fichier n'existe pas ou manque les permissions nécessaires), l'erreur est enregistrée et l'exécution du programme se termine.
3. Une instance de `bufio.Scanner`, `scanner` pour le fichier ouvert est créée pour lire le fichier.
4. Le fichier est lu ligne par ligne avec une boucle `for`. Si une ligne n'est pas vide (elle contient des caractères), le compteur est incrémenté.
5. Après que les lignes ont été analysées, il vérifie s'il y a des erreurs qui se sont produites lors de la lecture du fichier. Si une erreur s'est produite, elle enregistre le message d'erreur.
6. Le nombre de lignes non vides trouvées dans le fichier est imprimé.
7. Enfin, le fichier est fermé et s'il y a une erreur lors de la fermeture du fichier, il enregistre le

message d'erreur.

# Section 2 : Écrire dans un fichier

La façon la plus simple de d'ouvrir un fichier pour l'écriture est avec la fonction `os.Create`. Elle s'appelle de cette façon parce que si le fichier n'existe pas, alors le fichier va être créé avant de pouvoir y écrire. Si le fichier existe déjà, il sera *tronqué*, c'est-à-dire que son contenu sera effacé et pour ensuite y écrire à partir du début. C'est comme si on écrivait par-dessus le fichier existant.

## Exemple simple d'écriture dans un fichier

```
func section2a(filename string) {
    // Créer un nouveau fichier, ou le tronquer s'il existe
    file, err := os.Create(filename)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    // Écrire des bytes dans le fichier
    bytesWritten, err := file.WriteString("Bonjour tout le monde !")
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("Écrit %d bytes.\n", bytesWritten)
}
```

1. La fonction a comme paramètre le nom du fichier `filename` dans lequel on doit écrire.
2. On commence par création du nouveau fichier en utilisant la fonction `os.Create` avec le nom de fichier fourni en argument. Cette fonction crée un nouveau fichier et si le fichier existe déjà, elle tronque le contenu du fichier. Le résultat de la fonction `os.Create` comprend deux variables. La première est un pointeur de fichier `file` qui est utilisée pour les opérations de fichier ultérieures et la seconde est une erreur `err` qui aurait pu se produire.
3. On vérifie ensuite s'il y a eu une erreur lors de la création du fichier. S'il y a eu une erreur (`err` n'est pas `nil`), on logue l'erreur et termine immédiatement le programme en utilisant `log.Fatal(err)`.

4. L'instruction `defer file.Close()` fait en sorte que `file.Close()` soit appelée juste avant que la fonction `section2a` ne se termine, soit normalement ou via une erreur non traitée, afin de libérer les ressources du système (voir les détails plus bas).
5. Ensuite, on utilise la fonction `file.WriteString` pour écrire une chaîne de caractères dans le fichier. Le comptage en octets du contenu écrit est stocké dans `bytesWritten`, et toute erreur pendant le processus est stockée dans `err`.
6. Encore une fois, toute erreur lors de l'écriture dans le fichier est vérifiée et s'il y a eu une erreur, le programme enregistre l'erreur et se termine immédiatement.
7. Si l'écriture des bytes est réussie, le nombre de bytes écrits est enregistré dans le journal (`log`).

Si on n'utilisait pas `defer` ici, et qu'on appelait `Close` à la fin de la fonction comme dans les exemples précédents, alors ça pourrait créer des problèmes. Quand on ouvre un fichier (ou toute autre resource), il faut s'assurer de la fermer, ou libérer, avant de quitter la fonction. Si on a un `return` avant de se rendre à la fin de la fonction, ou si on quitte à cause d'une erreur, comme avec `log.Fatal` ou avec une panique (une erreur grave au moment de l'exécution), alors le fichier, ou en général la resource, ne sera pas fermée, et ça pourrait causer des problèmes, comme des fuites de mémoires ou, dans notre cas, des données qui ne seraient pas enregistrées correctement dans le fichier ouvert. Si on n'utilise pas `defer`, il faudrait appeler `file.Close()` possiblement à plusieurs endroits, à chaque endroit où on pourrait potentiellement quitter la fonction. Dans certains cas, comme pour les paniques, elles pourraient arriver à des endroits imprévus, donc la resource pourrait ne jamais être correctement fermée ou libérée.

L'utilisation de `defer file.Close()` nous assure que le fichier sera fermé, peu importe la façon dont nous quittons la fonction. Dans les exemples précédents, puisque les fichiers étaient ouverts en lecture seule, le fichier n'était pas modifié, donc on ne pouvait pas perdre de données si le fichier n'était pas fermé correctement. Mais il est tout de même préférable d'utiliser `defer` dans tous les cas pour s'assurer d'une bonne gestion de la mémoire, pour ne pas conserver les contenu des fichiers en mémoire pour rien.

## Exemple : écrire des données entrées par l'utilisateur dans un fichier

La fonction suivante demande des entrées à l'utilisateur et écrit ces entrées dans un fichier CSV en utilisant les opérations d'E/S de fichiers de la bibliothèque standard.

```

func section2b() {
    nAttempts := 3
    // demander le nom du fichier
    fmt.Println("Nom du fichier dans lequel enregistrer les données : ")
    filename, err := ReadNonEmptyLine(nAttempts)
    if err != nil {
        log.Fatalf("Impossible de lire le nom du fichier : %s", err)
    }

    // ouvrir le fichier
    file, err := os.Create(filename)
    if err != nil {
        log.Fatalf("Impossible d'ouvrir le fichier : %s", err)
    }
    defer file.Close()

    done := false
    for !done {
        // lire le nom
        fmt.Println("Veuillez entrer votre nom : ")
        name, err := ReadNonEmptyLine(nAttempts)
        if err != nil {
            log.Printf("Impossible de lire le nom : %s", err)
        } else {
            // s'il n'y a pas d'erreur, lire l'âge
            fmt.Println("Veuillez entrer votre âge : ")
            age, err := ReadInt(nAttempts)
            if err != nil {
                log.Printf("Impossible de lire l'âge : %s", err)
            } else {
                // s'il n'y a pas d'erreur
                line := fmt.Sprintf("%s,%d\n", name, age)
                _, err = file.WriteString(line)
                if err != nil {
                    log.Fatalf("Impossible d'écrire dans le fichier :
%s", err)
                }
            }
        }
    }
}

```

```
// continuer ou non ?
fmt.Println("Voulez-vous entrer continuer (O/N) ?")
continueInput, err := ReadNonEmptyLine(nAttempts)
if err != nil {
    log.Printf("Impossible de lire la réponse : %s", err)
} else {
    if strings.ToUpper(continueInput) == "N" {
        done = true
    }
}
}
```

Voici un pas à pas de la fonction section2b:

1. La fonction `section2b` commence par déclarer une variable `nAttempts` qui définit une limite au nombre de tentatives de lecture des entrées.
  2. Elle demande à l'utilisateur de saisir le nom du fichier dans lequel les données doivent être enregistrées. Ceci est fait via l'appel à `ReadNonEmptyLine(nAttempts)`. Si le nom du fichier est lu avec succès (non vide et pas d'erreur de lecture), il continue, sinon il quitte le programme en enregistrant l'erreur fatale.
  3. Ensuite, elle tente de créer le fichier spécifié à l'aide de `os.Create(filename)`. Si une erreur se produit pendant la création du fichier, une erreur fatale est enregistrée et le programme quitte. La fermeture du fichier est reporté après que toutes les opérations suivantes ont été terminées.
  4. Elle entre ensuite dans une boucle qui continue jusqu'à ce que `done` devienne vraie.
  5. Dans la boucle, elle demande d'abord à l'utilisateur d'entrer son nom, en utilisant une vérification des erreurs similaire à l'entrée précédente. Si le nom est lu avec succès, elle demande ensuite l'âge à l'aide de la fonction `ReadInt(nAttempts)`. Sinon, elle enregistre l'erreur et répète la boucle.
  6. S'il n'y a pas d'erreur lors de la lecture de l'âge, elle formate le nom et l'âge en une chaîne avec `fmt.Sprintf`, puis tente d'écrire cette chaîne dans le fichier ouvert à l'aide de

`file.WriteString(line)`. Elle vérifie les erreurs lors de l'opération d'écriture, enregistre l'erreur fatale et quitte si nécessaire.

7. Ensuite, elle demande à l'utilisateur s'il souhaite continuer à entrer plus de données. Si l'utilisateur saisit 'N' ou 'n', elle change `done` à `true` ce qui brisera la boucle, sinon elle retourne à la demande du nom et de l'âge.

**Note:** le paquet `encoding/csv` offre une meilleure interface pour lire et écrire des fichiers CSV. L'exemple précédent écrit du texte directement dans le fichier. `encoding/csv` permet de lire et d'écrire plus efficacement. Deux fonctions qui utilisent `encoding/csv` suivent. La première lit un fichier CSV, et l'autre écrit dans un fichier CSV.

## Lire et écrire des fichiers CSV

```
func readCSVFile() {
    f, err := os.Open("abc.csv")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    reader := csv.NewReader(f)
    records, err := reader.ReadAll()
    if err != nil {
        log.Fatal(err)
    }

    for _, record := range records {
        fmt.Println(record) // record est une tranche de chaînes de
                            // caractères représentant les lignes du
fichier
    }
}

func writeCSVFile() {
    f, err := os.Create("def.csv")
    if err != nil {
        log.Fatal(err)
    }
```

```
defer f.Close()

writer := csv.NewWriter(f)
data := []string{"Col1", "Col2", "Col3"}

if err := writer.Write(data); err != nil {
    log.Fatal(err)
}
writer.Flush()

if err := writer.Error(); err != nil {
    log.Fatal(err)
}
```

# Section 3 : Copier un fichier en ajoutant des numéros de ligne

La fonction `CopyFileWithLineNumbers(srcFile string, dstFile string)` fait exactement comme son nom le suggère : Elle copie le contenu du fichier source (spécifié par `srcFile`) vers le fichier de destination (spécifié par `dstFile`), et insère des numéros de ligne pendant l'opération de copie.

```
func CopyFileWithLineNumbers(srcFilename string, dstFilename string) error {
    // Ouvrez le fichier source en lecture seule
    src, err := os.Open(srcFilename)
    if err != nil {
        return err
    }
    defer src.Close()

    // Ouvrez le fichier de destination en mode écriture
    dst, err := os.Create(dstFilename)
    if err != nil {
        return err
    }
    defer dst.Close()

    scanner := bufio.NewScanner(src)
    writer := bufio.NewWriter(dst)
    lineNumber := 1

    for scanner.Scan() {
        lineStr := fmt.Sprintf("%d\t%s\n", lineNumber, scanner.Text())
        if _, err := writer.WriteString(lineStr); err != nil {
            return err
        }
        lineNumber++
    }

    if err := scanner.Err(); err != nil {
```

```

        return err
    }

    if err := writer.Flush(); err != nil {
        return err
    }

    return nil
}

```

Voici ce que fait la fonction, étape par étape :

- Ouvre le fichier source en mode lecture seule, en utilisant `os.Open()`. S'il y a une erreur (par exemple, le fichier n'existe pas), elle retourne l'erreur.
- Ouvre le fichier de destination en mode écriture (le créant s'il n'existe pas), en utilisant `os.Create()`. Là encore, s'il y a une erreur, elle la retourne.
- Crée un `bufio.Scanner` pour lire le fichier source et un `bufio.Writer` pour écrire dans le fichier de destination.
- Démarre une boucle qui lit chaque ligne du fichier source.
  - Pour chaque ligne, elle formate une chaîne qui se compose du numéro de ligne (suivi d'un caractère de tabulation), du texte de la ligne du fichier source et d'un caractère de nouvelle ligne.
  - Écrit cette chaîne dans le fichier de destination. S'il y a une erreur, elle la retourne.
  - Incrémente le numéro de ligne pour la prochaine itération.
- Après la boucle (quand toutes les lignes ont été traitées), elle vérifie s'il y a eu des erreurs pendant le balayage. Si c'est le cas, elle retourne l'erreur.
- Appelle `writer.Flush()` pour s'assurer que toutes les opérations en mémoire tampon (s'il y en a) ont été appliquées au fichier de destination. S'il y a une erreur, elle la retourne.
- Si toutes les étapes ci-dessus se sont déroulées sans erreur, elle retourne `nil` indiquant que l'opération a réussi.

Ensuite, nous avons la fonction `section3()`, qui appelle la fonction `CopyFileWithLineNumbers`:

```
func section3() {  
    err := CopyFileWithLineNumbers("allo.txt", "allo2.txt")  
    if err != nil {  
        fmt.Println("Erreur lors de la copie du fichier:", err)  
    } else {  
        fmt.Println("Le fichier a été copié avec succès avec des  
numéros de ligne.")  
    }  
}
```

- Elle appelle `CopyFileWithLineNumbers()` pour copier le contenu de "allo.txt" à "allo2.txt".
- S'il y a une erreur, elle imprime un message indiquant que la copie du fichier a échoué, et elle donne l'erreur.
- S'il n'y a pas d'erreur, elle imprime un message indiquant le succès.

# Section 4 : Autres façons d'ouvrir un fichier

La fonction `os.OpenFile` est un outil très puissant qui vous permet d'ouvrir un fichier avec différents niveaux d'accès et permissions. La signature de la fonction est :

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

Où :

- `name` est le chemin vers le fichier.
- `flag` est une valeur `int` qui représente les modes d'accès aux fichiers (lecture, écriture, ajout, etc.).
- `perm` sont les permissions de fichiers de style Unix que vous souhaitez définir pour le fichier.

## flag

Dans la documentation du paquet `os` (<https://pkg.go.dev/os#pkg-constants>), on a les définitions suivantes :

```
const (
    // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
    O_RDONLY int = syscall.O_RDONLY // open the file read-only.
    O_WRONLY int = syscall.O_WRONLY // open the file write-only.
    O_RDWR   int = syscall.O_RDWR   // open the file read-write.
    // The remaining values may be or'ed in to control behavior.
    O_APPEND int = syscall.O_APPEND // append data to the file when
writing.
    O_CREATE int = syscall.O_CREAT // create a new file if none
exists.
    O_EXCL   int = syscall.O_EXCL  // used with O_CREATE, file must
not exist.
    O_SYNC    int = syscall.O_SYNC  // open for synchronous I/O.
    O_TRUNC   int = syscall.O_TRUNC // truncate regular writable file
```

```
when opened.
```

```
)
```

Le paramètre `flag` spécifie le mode d'accès au fichier. Ce paramètre est généralement défini en effectuant un *OU* binaire entre deux ou plusieurs constantes du package `os`. Voici un résumé de ce que fait chaque flag :

1. **`os.O_RDONLY`**: Ce flag ouvre le fichier en mode lecture seule. Sa valeur est `0`.
2. **`os.O_WRONLY`**: Ce flag ouvre le fichier en mode écriture seule. Si le fichier existe déjà, et que ce flag est utilisé seul, le fichier ne sera pas tronqué.
3. **`os.O_RDWR`**: Ce flag ouvre le fichier en mode lecture-écriture.
4. **`os.O_APPEND`**: Ce flag ouvre le fichier en mode ajout, de sorte que les données sont toujours écrites à la fin du fichier.
5. **`os.O_CREATE`**: Ce flag crée le fichier s'il n'existe pas.
6. **`os.O_TRUNC`**: Ce flag vide le fichier (le tronque) s'il existe.
7. **`os.O_EXCL`**: Ce flag, utilisé uniquement avec `O_CREATE`, renvoie une erreur si le fichier existe déjà.
8. **`os.O_SYNC`**: Ce flag ouvre le fichier pour l'I/O synchrone, rendant les opérations d'E/S immédiates et s'assurant que les données sont écrites sur le matériel sous-jacent avant de considérer l'opération d'écriture comme terminée.

La combinaison de ces flags vous permet de configurer le comportement de `os.OpenFile` pour répondre à vos besoins spécifiques. Ces flags sont indépendants de la plateforme, vous pouvez donc les utiliser de manière fiable sur tous les systèmes d'exploitation pris en charge.

## perm

Le troisième argument de la fonction `os.OpenFile`, `perm`, est utilisé pour définir les permissions du fichier dans le style Unix. L'argument `perm` signifie "permission".

Dans les systèmes basés sur Unix, cette permission contrôle qui peut lire, écrire ou exécuter le fichier. Ces permissions sont représentées par un ensemble de 9 bits. Elles sont généralement représentées en octal (base 8) pour plus de commodité.

Décortiquons cela:

- Le premier chiffre représente les permissions du propriétaire.
- Le deuxième chiffre représente les permissions du groupe.
- Le troisième chiffre représente les permissions de tous les autres.

Chaque chiffre est la somme de :

- 4 pour lire (r),
- 2 pour écrire (w),
- 1 pour exécuter (x).

Par exemple, prenons une permission octale : 0644.

- Le 0 au début est là parce que c'est un nombre octal.
- 6 est 4+2, ce qui signifie des permissions de lecture et d'écriture pour le propriétaire du fichier.
- 4 signifie seulement la permission de lecture pour les membres du groupe du fichier.
- Le dernier 4 signifie seulement la permission de lecture pour tous les autres.

Voici un autre exemple : 0755.

- 7 est 4+2+1, ce qui signifie des permissions de lecture, d'écriture et d'exécution pour le propriétaire.
- Le premier 5 est 4+0+1, ce qui signifie des permissions de lecture et d'exécution pour le groupe.
- Le dernier 5 signifie des permissions de lecture et d'exécution pour tous les autres.

En conclusion, le paramètre `perm` de la fonction `os.OpenFile` vous permet de contrôler l'accès à votre fichier lorsque vous le créez. Par exemple :

```
file, err := os.OpenFile("filename", os.O_RDWR|os.O_CREATE, 0644)
```

Cette ligne créerait un fichier avec des permissions de lecture/écriture pour le propriétaire, mais uniquement avec des permissions de lecture pour tous les autres.

# Exemples

Voici quelques façons différentes d'utiliser la fonction `os.OpenFile`:

1. Ouvrir un fichier en mode lecture seule :

```
file, err := os.OpenFile("file.txt", os.O_RDONLY, 0644)
```

2. Ouvrir un fichier en mode écriture seule (le contenu du fichier existant sera supprimé) :

```
file, err := os.OpenFile("file.txt", os.O_WRONLY|os.O_TRUNC, 0644)
```

3. Ouvrir un fichier en mode écriture seule et créer le fichier s'il n'existe pas :

```
file, err := os.OpenFile("file.txt", os.O_WRONLY|os.O_CREATE, 0644)
```

4. Ouvrir un fichier en mode ajout pour l'écriture (ajoute au contenu de fichier existant) :

```
file, err := os.OpenFile("file.txt", os.O_APPEND|os.O_WRONLY, 0644)
```

5. Ouvrir un fichier en mode lecture-écriture (le contenu du fichier existant sera supprimé) :

```
file, err := os.OpenFile("file.txt", os.O_RDWR|os.O_TRUNC, 0644)
```

6. Ouvrir un fichier en mode lecture-écriture et créer le fichier s'il n'existe pas :

```
file, err := os.OpenFile("file.txt", os.O_RDWR|os.O_CREATE, 0644)
```

Dans tous ces exemples, vous devriez toujours vérifier l'erreur renvoyée (`err`) pour s'assurer que le fichier a été ouvert avec succès :

```
if err != nil {
    log.Fatal(err)
```

```
}
```

N'oubliez pas non plus de fermer le fichier lorsque vous avez terminé de l'utiliser. Vous pouvez utiliser l'instruction `defer` pour vous assurer que `file.Close()` sera appelé à la fin de la fonction conteneur.

```
defer file.Close()
```

# Chapitre 5 : Les structures

## Exemple de base

En Go, une `struct` (structure) est une collection personnalisée de types de données. Elle permet de combiner des éléments de différents types dans une seule structure de données. L'exemple qui suit démontre comment les structs peuvent être utilisées pour créer et gérer des instances d'une personne.

Voici un exemple minimal d'utilisation d'une `struct` appelée `Person` qui contient les champs `Name` et `Age`:

```
package main

import "fmt"

type Person struct {
    Name string
    Age int
}

func main() {
    // Initialiser une nouvelle instance de Person
    person1 := Person{Name: "John Doe", Age: 30}

    // Accéder aux champs de la struct
    fmt.Println(person1.Name) // John Doe
    fmt.Println(person1.Age) // 30

    // Modifier les champs de la struct
    person1.Name = "Jane Doe"
    person1.Age = 28

    fmt.Println(person1.Name) // Jane Doe
    fmt.Println(person1.Age) // 28

    fmt.Println(person1) // {Jane Doe 28}
}
```

1. Nous définissons un nouveau type appelé Person qui est une struct contenant deux champs : Name de type string et Age de type int.
2. Dans la fonction main, nous créons une nouvelle instance de Person et l'initialisons avec des valeurs pour Name et Age.
3. Nous utilisons l'opérateur . pour accéder aux champs de notre instance de Person, puis nous imprimons ces valeurs avec fmt.Println.
4. Enfin, nous utilisons l'opérateur '' pour modifier les valeurs des champs Name et Age dans notre instance de Person.
5. Nous pouvons aussi imprimer une structure directement dans un fmt.Println, mais la sortie peut être ambiguë, comme le montre la dernière sortie dans l'exemple : à partir d'uniquement du résultat de la sortie, il n'est pas clair s'il y a 2 ou 3 champs dans la structure.

Puisque les champs Name et Age commencent par des majuscules, ils sont publics. S'ils commençaient par une minuscule, ils seraient privés, donc accessibles seulement au code situé dans le même paquet (package) que sa déclaration. En Go, l'accessibilité (public vs. privé) est toujours définie au niveau du paquet.

# Section 1 : Valeur vs. pointeur

Voici un exemple de structure pour représenter un point en 2 dimensions, chaque dimension étant un nombre réel (type float64).

```
type Point struct {
    X float64
    Y float64
}

func NewPoint(x float64, y float64) *Point {
    return &Point{X: x, Y: y}
}

func section1a() {
    p := Point{2, 5}
    fmt.Println(p)

    pPtr := &p
    fmt.Printf("%p %v\n", pPtr, *pPtr)

    pPtr = &Point{4, 3}
    fmt.Printf("%p %v\n", pPtr, *pPtr)

    pPtr = new(Point)
    fmt.Printf("%p %v\n", pPtr, *pPtr)
    pPtr.X = 1
    pPtr.Y = 6
    fmt.Printf("%p %v\n", pPtr, *pPtr)

    pPtr = NewPoint(-1, 5.5)
    fmt.Printf("%p %v\n", pPtr, *pPtr)
}
```

Sortie :

```
{2 5}
0xc0000a6020 {2 5}
```

```

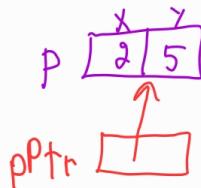
0xc0000a6050 { 4 3 }
0xc0000a6070 { 0 0 }
0xc0000a6070 { 1 6 }
0xc0000a60a0 {-1 5.5}

```

La variable `p` est de type valeur, elle contient un point, qui est composé de 2 `float64`. L'espace utilisé par cette variable est la somme de l'espace utilisé par ses composantes, donc 128 bits dans ce cas-ci. En général, l'espace utilisé pourrait être plus grand que la somme de l'espace utilisé par ses composantes si les composantes sont de différentes tailles (voir les détails plus loin). La variable `pPtr` est un pointeur vers un `Point`.

`p := Point{2,5}`

`pPtr := &p`



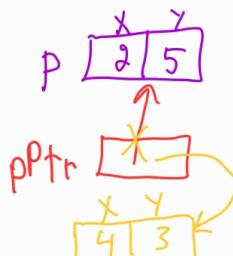
struct valeur

On peut créer un nouveau point de cette façon : `pPtr = &Point{4, 3}`. Notez l'utilisation du `&` devant `Point{4, 3}`. On crée un nouveau point, et on place l'adresse du nouveau point dans `pPtr`. On ne touche pas au premier point ici, on remplace seulement l'adresse mémoire contenu dans la variable `pPtr` par une nouvelle adresse. On a donc 2 points différents en mémoire à ce moment. Vérifiez dans la sortie, les adresses sont différentes. Les adresses vont être fort probablement différentes des adresses données plus haut.

`p := Point{2,5}`

`pPtr := &p`

`pPtr = &Point{4,3}`



struct pointeur

On peut également créer un nouveau point avec `new`. Le problème avec `new`, c'est qu'on ne peut pas spécifier les valeurs des différents champs de cette façon. On doit donc modifier les valeurs par défaut des champs après avoir créé la structure avec `new`. En Go, lorsqu'une variable est de *type pointeur vers une structure*, il n'est pas nécessaire de déréférencer le pointeur avec l'astérisque `*` avant d'accéder à ces champs. Si on devait le faire, on utiliserait `(*pPtr).X = 1` à la place de `pPtr.X = 1`. Il est très commun de manipuler des pointeurs vers des

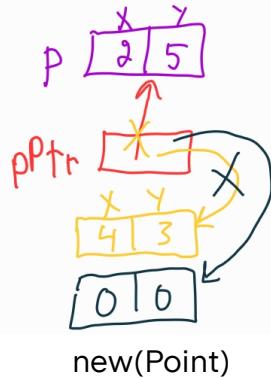
structures, donc les créateurs de Go ont décidé d'introduire la notation avec le `.` non seulement directement sur les structures, mais aussi sur les pointeurs vers des structures.

`p := Point{2,5}`

`pPtr := &p`

`pPtr = &Point{4,3}`

`pPtr = new(Point)`



Il est également possible de déclarer une fonction pour créer un nouveau point, similaire à un constructeur dans un langage orienté-objet. Dans GoLand, il est possible de générer un constructeur automatiquement avec le raccourci `alt-ins`. Cette fonction démontre également qu'il est possible de spécifier la valeur des champs d'une structure par leurs noms, suivis d'un `:` et des valeurs de champ. Si on ne spécifie pas les noms des champs dans l'initialisation, on doit placer les valeurs des champs dans le même ordre que leurs déclarations.

Il faut noter que dans cet exemple, après avoir créé un nouveau point avec `new` et avoir placé son adresse dans `pPtr`, nous perdons l'adresse du point `{4 3}`, qui ne sera plus accessible du tout par le programme. Similairement, après avoir créé un nouveau point avec le constructeur, nous allons perdre le point `{1 6}` parce que nous perdons son adresse. En général, il serait préférable de réutiliser les points déjà créés pour éviter d'allouer de la mémoire pour rien. En Go, comme dans d'autres langages tels que Java et C#, il y a un vidangeur (*garbage collector*) qui va récupérer cet espace mémoire, mais ce procédé n'est pas gratuit, il pourrait y avoir un impact sur la performance dans certains cas. Dans la plupart des cas, l'impact sera faible, mais de bonnes pratiques en termes de gestion de la mémoire sont préférables. Les langages qui n'ont pas de vidangeur, comme le C et le C++, ne pourront pas récupérer la mémoire perdue, ce qui créera une fuite de mémoire (*memory leak*).

# Section 2 : Méthodes

Voici le contenu du fichier `point.go`, qui contient la même définition de la structure `Point` et le même constructeur que dans la section précédente, mais qui contient en plus quelques méthodes définies sur `Point`.

`point.go` sur Github (<https://github.com/profdenis/native1/tree/master/chap5/point.go>)

```
package main

import (
    "fmt"
    "math"
)

type Point struct {
    X float64
    Y float64
}

func NewPoint(x float64, y float64) *Point {
    return &Point{X: x, Y: y}
}

// String convertit le Point en String pour implémenter l'interface
Stringer
func (p *Point) String() string {
    return fmt.Sprintf("Point(%f, %f)", p.X, p.Y)
}

// Add additionne 2 points et retourne le résultat dans un 3e point
func (p *Point) Add(other *Point) *Point {
    return &Point{p.X + other.X, p.Y + other.Y}
}

// Subtract soustrait 2 point et retourne le résultat dans un 3e point
func (p *Point) Subtract(other *Point) *Point {
    return &Point{p.X - other.X, p.Y - other.Y}
}
```

```

// Distance calcule et retourne la distance Euclidienne entre 2 point
func (p *Point) Distance(other *Point) float64 {
    return math.Sqrt(math.Pow(other.X-p.X, 2) + math.Pow(other.Y-p.Y,
2))
}

// Scale mise à l'échelle d'un Point par un facteur et retourne le
résultat dans un nouveau point
func (p *Point) Scale(factor float64) *Point {
    return &Point{p.X * factor, p.Y * factor}
}

```

Les méthodes sont un type de fonctions particulières : ce sont des fonctions définies sur des structures. C'est le même concept que les méthodes dans d'autres langages comme Java et C#, mais définies sur des structures à la place des classes.

## Est-ce que Go est orienté-objet ?

Go n'est pas un langage orienté-objet dans le même sens que Java et C# le sont parce que Go ne supporte pas l'héritage entre classes (ou structures), mais à part pour cette différence, les structures et les classes sont des concepts très similaires. Go supporte les interfaces et la composition de structures et d'interfaces en plus des méthodes, mais pas l'héritage entre classes et le polymorphisme, donc c'est pourquoi Go n'est pas considéré comme étant orienté-objet par certains qui considèrent l'héritage entre classes et le polymorphisme comme étant obligatoire pour qu'un langage soit qualifié d'orienté-objet. Il n'existe aucune définition exacte de ce qu'est un langage orienté-objet, donc dépendamment de la définition utilisée, Go peut être considéré comme orienté-objet ou non.

## Comment définir une méthode

Comme présenté dans l'exemple plus haut, les méthodes sont définies en dehors de la définition de la structure. Une méthode est une fonction avec un élément de plus : un *receveur* est ajouté entre le mot `func` et le nom de la fonction. Le receveur défini sur quel type de structure la méthode peut être appelée. Le receveur peut être de type valeur ou pointeur. Le receveur est comme un paramètre supplémentaire sur une fonction, donc les mêmes règles s'appliquent au receveur qu'aux autres paramètres d'une fonction :

- si le receveur est de type valeur, alors le contenu de la structure sur laquelle la méthode est appelée sera copiée dans une variable locale temporaire pour l'exécution de la méthode.
- si le receveur est un pointeur, alors le contenu de la structure ne sera pas copié parce que l'adresse de la structure sera donnée à la méthode, donc les modifications apportées à la structure dans la méthode seront visibles à l'extérieur de la méthode.

## Comment appeler une méthode

Dans cet exemple, on commence par créer 2 points avec le constructeur, et ensuite, on appelle la méthode `Add` sur le premier point, en donnant le 2e point en argument. Pour appeler une méthode sur une structure, il faut utiliser le `.` directement précédé d'une structure ou d'un pointeur vers une structure, suivi du nom de la méthode et de ses arguments.

```
p1 := NewPoint(1, 2)
```

```
p2 := NewPoint(3, 4)
```

```
fmt.Println(p1)
```

```
fmt.Println(p2)
```

```
p3 := p1.Add(p2)
```

```
fmt.Println(p3)
```

Sortie :

```
Point(1.000000, 2.000000)
```

```
Point(3.000000, 4.000000)
```

```
Point(4.000000, 6.000000)
```

## Conversion pour l'impression

La méthode `String` définie sur la structure `Point` est spéciale car elle permet la conversion de la structure en chaîne de caractères pour la sortie, sur le terminal dans cet exemple-ci. Quand on appelle `fmt.Println` (ou d'autres fonctions d'impression) sur un point, si la méthode `String()` qui retourne une `string` est définie, alors elle sera utilisée pour la conversion du point en chaîne de caractères. Cette fonction est définie dans l'interface `Stringer`. En Java, la méthode `toString` joue le même rôle.

Si cette méthode n'est pas définie, alors la sortie sera :

```
&{1 2}  
&{3 4}  
&{4 6}
```

Par défaut, si la méthode String n'est pas définie, alors la sortie commencera par & si on a un pointeur vers une structure, et entre accolades {}, on aura la valeur de tous les champs séparés par des espaces.

# Section 3 : Sérialisation des structures

La sérialisation est le processus de conversion des structures de données ou des objets en un format qui peut être stocké (par exemple, dans un fichier ou une mémoire tampon), ou transmis (par exemple, à travers un réseau) et reconstruit plus tard (peut-être dans un environnement différent).

Pour une structure Go, la sérialisation signifie la conversion de l'instance de la struct en un format qui peut être facilement stocké ou transmis. Le format cible peut être JSON, XML, texte brut ou tout autre format de données utilisable.

Reprendons l'exemple de la structure Person:

```
type Person struct{
    Name string
    Age int
}
```

Et une instance de cette struct :

```
p := Person{Name: "Alice", Age:25}
```

La sérialisation de cette instance de struct (dans le format JSON) transforme la struct en une chaîne JSON :

```
{
    "Name": "Alice",
    "Age": 25
}
```

On peut ensuite enregistrer cette chaîne JSON dans un fichier, ou l'envoyer à travers un réseau, etc. Lorsque nous avons besoin d'utiliser ces données à nouveau, nous désérialisons la chaîne JSON en une instance de la struct Person. Cela permet de sauvegarder l'état de l'objet et de le restaurer plus tard, ou de le transmettre à un autre emplacement ou un autre système.

Dans certaines situations, selon les conventions utilisées, il peut être préférable de nommer les clés en JSON, qui correspondent aux noms des champs en Go, en utilisant le *camelCase*. Pour ce faire, on peut placer des étiquettes (*tags*) dans la déclaration des structures, qui seront utilisées dans le processus de conversion (voir les exemples plus bas). La déclaration de la structure Person deviendra :

```
type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}
```

La création des instances ne sera pas affectée, mais le résultat de la sérialisation donnera du JSON avec les noms spécifiés dans les étiquettes.

```
{
    "name": "Alice",
    "age": 25
}
```

## Exemple complet

Voici un exemple complet de sérialisation et désérialisation d'une structure vers/à partir une chaîne JSON.

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func section3a() {
    // Créer une instance de Person
    p := Person{Name: "John", Age: 30}
```

```

// Sérialiser la struct en JSON
jsonData, err := json.Marshal(p)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(string(jsonData)) // Affiche :
{"name":"John", "age":30}

// Désérialiser le JSON en struct
jsonStr := `{"name":"James", "age":40}`
var p2 Person
err = json.Unmarshal([]byte(jsonStr), &p2)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(p2) // Affiche : {James 40}
}

```

Sortie :

```

{"name":"John", "age":30}
{James 40}

```

1. Une instance de la structure `Person` est créée avec le nom "John" et l'âge 30.

```
p := Person{Name: "John", Age: 30}
```

2. Cette instance est ensuite sérialisée au format JSON à l'aide de la fonction `json.Marshal`. S'il y a une erreur lors de ce processus, elle est imprimée et la fonction retourne immédiatement.

```

jsonData, err := json.Marshal(p)
if err != nil {
    fmt.Println(err)
    return
}

```

```
    }
    fmt.Println(string(jsonData)) // Imprime: {"name": "John", "age": 30}
```

3. Ensuite, une chaîne JSON représentant une autre personne (*nom : "James", âge : 40*) est déserialisée en une instance de la structure Person en utilisant la fonction json.Unmarshal. Là encore, s'il y a une erreur lors de ce processus, elle est imprimée et la fonction retourne immédiatement.

```
jsonStr := `{"name": "James", "age": 40}`
var p2 Person
err = json.Unmarshal([]byte(jsonStr), &p2)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(p2) // Imprime: {James 40}
```

La structure Person et ces opérations (sérialisation et déserialisation en JSON) sont prédominantes dans les scénarios où les données doivent être stockées, transférées ou traitées. JSON (JavaScript Object Notation) est un format populaire à ces fins en raison de sa simplicité et de sa compatibilité avec de nombreux langages, y compris Go.

## Exemple d'écriture d'une structure dans un fichier

```
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
)

type Person struct {
    Name string `json: "name"`
    Age  int    `json: "age"`
}

func section3b() {
    // Créer une instance de Person
```

```

p := Person{Name: "John", Age: 30}

// Sérialiser la struct en JSON
jsonData, err := json.Marshal(p)
if err != nil {
    fmt.Println(err)
    return
}

// Ouvrir le fichier en mode écriture
file, err := os.OpenFile("person.json", os.O_CREATE|os.O_WRONLY,
0644)
if err != nil {
    fmt.Println(err)
    return
}
defer file.Close()

// Écrire les données JSON dans le fichier
_, err = file.Write(jsonData)
if err != nil {
    fmt.Println(err)
    return
}
file.Sync()

// Lire les données du fichier
data, err := os.ReadFile("person.json")
if err != nil {
    fmt.Println(err)
    return
}

// Désérialiser les données JSON pour recréer une instance de
Person
var p2 Person
err = json.Unmarshal(data, &p2)
if err != nil {
    fmt.Println(err)
}

```

```

    return
}

// Afficher l'instance de Person
fmt.Println(p2) // Devrait afficher: {John 30}
}

```

Dans la fonction `section3b`:

1. On commence par créer une instance `Person` avec le nom "John" et l'âge 30, que l'on sérialise en JSON.
2. Ensuite, on ouvre un fichier appelé "person.json" en mode écriture. Si le fichier n'existe pas, il sera créé (`os.O_CREATE`). Si le fichier existe déjà, son contenu sera écrasé (`os.O_WRONLY`). Le chiffre `0644` est une valeur octale représentant les permissions du fichier.
3. On écrit ensuite les données JSON dans le fichier.
4. Après cela, on lit les données du fichier dans un `byte slice`.
5. Enfin, on désérialise les données JSON pour recréer une instance `Person` et on l'affiche.

**Note:** Assurez-vous de gérer vos erreurs correctement dans un véritable environnement de production. Ici, nous nous contentons d'imprimer les erreurs, mais en réalité, vous devriez les gérer d'une manière qui convient à votre application. Aussi, n'oubliez pas de fermer le fichier après l'avoir utilisé pour libérer les ressources système.

## Exemple d'écriture d'une tranche de structures dans un fichier

D'accord, pour écrire et lire une tranche de `Person`, vous pouvez suivre un schéma similaire.

Voici comment :

```

import (
    "encoding/json"
    "fmt"
    "os"
)

type Person struct {
    Name string `json:"name"`
}

```

```

Age int      `json:"age"`

}

func section3c() {
    // Créer une tranche de Person
    people := []Person{
        {Name: "John", Age: 30},
        {Name: "Jane", Age: 25},
        {Name: "Alice", Age: 35},
    }

    // Sérialiser la slice en JSON
    jsonData, err := json.MarshalIndent(people, "", "\t")
    if err != nil {
        fmt.Println(err)
        return
    }

    // Ouvrir le fichier en mode écriture
    file, err := os.OpenFile("people.json",
os.O_CREATE|os.O_TRUNC|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    // Écrire les données JSON dans le fichier
    _, err = file.Write(jsonData)
    if err != nil {
        fmt.Println(err)
        return
    }
    file.Sync()

    // Lire les données du fichier
    data, err := os.ReadFile("people.json")
    if err != nil {
        fmt.Println(err)
    }
}

```

```

    return
}

// Désérialiser les données JSON pour recréer une slice de Person
var people2 []Person
err = json.Unmarshal(data, &people2)
if err != nil {
    fmt.Println(err)
    return
}

// Afficher la slice de Person
fmt.Println(people2) // Devrait afficher : [{John 30} {Jane 25}
{Alice 35}]
}

```

Dans la fonction `section3c`:

1. On commence par créer une tranche de `Person` avec trois personnes, que l'on sérialise en JSON.
  - Dans ce cas-ci, on utilise `json.MarshalIndent` pour faciliter la lecture de la chaîne produite par un humain.
  - On donne la chaîne vide comme préfixe, et le caractère de tabulation pour le caractère d'indentation. On pourrait utiliser des espaces à la place, comme par exemple `" "`.
2. On ouvre ensuite un fichier appelé "people.json" en mode écriture. Si le fichier n'existe pas, il est créé (`os.O_CREATE`). Si le fichier existe déjà, son contenu est écrasé (`os.O_WRONLY`).
3. On y écrit ensuite les données JSON.
4. On lit ensuite les données du fichier dans une tranche de `byte`.
5. Enfin, on désérialise ces données JSON pour créer une nouvelle tranche de `Person` et on l'affiche.

S'il y a une erreur à chaque étape, celle-ci est affichée et la fonction retourne immédiatement.

Contenu du fichier `people.json`:

```
[  
  {  
    "name": "John",  
    "age": 30  
  },  
  {  
    "name": "Jane",  
    "age": 25  
  },  
  {  
    "name": "Alice",  
    "age": 35  
  }  
]
```

# Chapitre 6 : Gestion de la mémoire

Le langage de programmation Go gère la mémoire de manière automatique à travers un ramasse-miettes (*garbage collector* en anglais). Voici des détails importants à noter :

- **Allocation de mémoire:** En Go, lorsque vous déclarez une variable, Go alloue automatiquement de la mémoire pour cette variable. L'espace mémoire est alloué sur le **tas** si la variable est créée à l'aide du mot-clé `new` ou si elle est un composite (slice, map, channel) ou si vous utilisez seulement son adresse pour y accéder, sinon, elle est allouée sur la **pile**. Vous n'avez pas besoin de spécifier la quantité de mémoire nécessaire.
- **Garbage Collection:** Go utilise un ramasse-miettes pour libérer la mémoire. Cela signifie qu'au lieu de devoir libérer explicitement la mémoire que vous avez allouée (comme vous le feriez en C ou en C++), le ramasse-miettes de Go identifiera les portions de mémoire qui ne sont plus référencées par votre programme et libérera automatiquement cette mémoire. Cela contribue à prévenir les fuites de mémoire.
- **Concurrence:** Go est conçu pour être efficace avec le multiprocesseur et les systèmes concurrents. Le *garbage collector* de Go est conçu pour être rapide et ne pas bloquer l'exécution de votre programme pendant qu'il libère la mémoire. Cela facilite la création de programmes Go très concurrents.

Un des plus grands avantages de la gestion de la mémoire en Go est qu'elle diminue la complexité de la programmation en éliminant la nécessité de gérer manuellement la mémoire, prévenant ainsi de fuites de mémoire et d'erreurs d'accès à la mémoire.

```
package main

import "fmt"

func main() {
    // La mémoire est allouée automatiquement lors de la déclaration
    // d'une variable
    var num *int
    num = new(int) // Le mot-clé "new" alloue de la mémoire pour
    // stocker un int sur le tas

    // Le ramasse-miettes libérera automatiquement la mémoire allouée
```

```
à "num"
    // lorsque "num" ne sera plus utilisé
    fmt.Println(*num)
}
```

Le code ci-dessus est un exemple de comment la gestion de la mémoire est effectuée en Go.

# Section 1 : La pile d'exécution

En Go, chaque goroutine se voit attribuer sa propre pile, (*stack*) qui contient l'espace mémoire pour les variables locales et les arguments de fonction. Lorsqu'une fonction est appelée, un *frame* (aussi appelé *cadre d'exécution*) est créé sur la pile de la goroutine. Ce frame contient les variables locales et les arguments de la fonction.

Voici une vue simplifiée de ce à quoi ressemble la mécanique de la pile et des frames :



Pile et cadres d'exécution

Prenons par exemple deux fonctions A et B. En supposant que la fonction A appelle la fonction B, le frame pour A sera d'abord empilé, suivi par le frame pour B. Chaque frame contient les variables locales et les arguments de fonction pour la fonction correspondante. Le sommet de la pile correspond toujours à la fonction en cours d'exécution (B dans ce scénario).

Concernant l'allocation de mémoire, en Go, les variables locales et les arguments de fonction sont alloués sur la pile. C'est l'une des fonctionnalités clés de Go qui permet des performances élevées, car l'allocation et la désallocation sur la pile sont généralement plus rapides que sur le tas. De plus, la mémoire allouée sur la pile ne nécessite pas de *garbage collection*, ce qui peut également améliorer les performances.

Il est cependant important de noter que ce n'est pas toujours le cas. Si une variable locale est référencée en dehors de la fonction (c'est-à-dire, elle "échappe" la fonction), alors elle sera

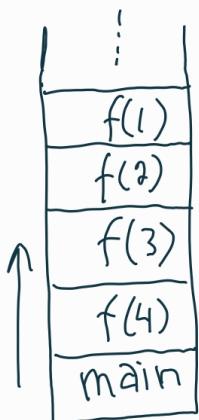
allouée sur le tas. C'est ce qu'on appelle *l'analyse d'échappement*, que le compilateur Go utilise pour déterminer où allouer de la mémoire pour une variable.

## Exemple 1

Question: qu'est-ce que la fonction suivante calcule ?

```
func f(i int) int {
    if i <= 0 {
        return 0
    }
    return i + f(i-1)
}
```

Si on appelle la fonction de cette façon : `f(4)`, nous aurons une pile d'exécution qui ressemblera à ceci à un certain moment, si la fonction est appelée `f` est appelée à partir de la fonction `main`:

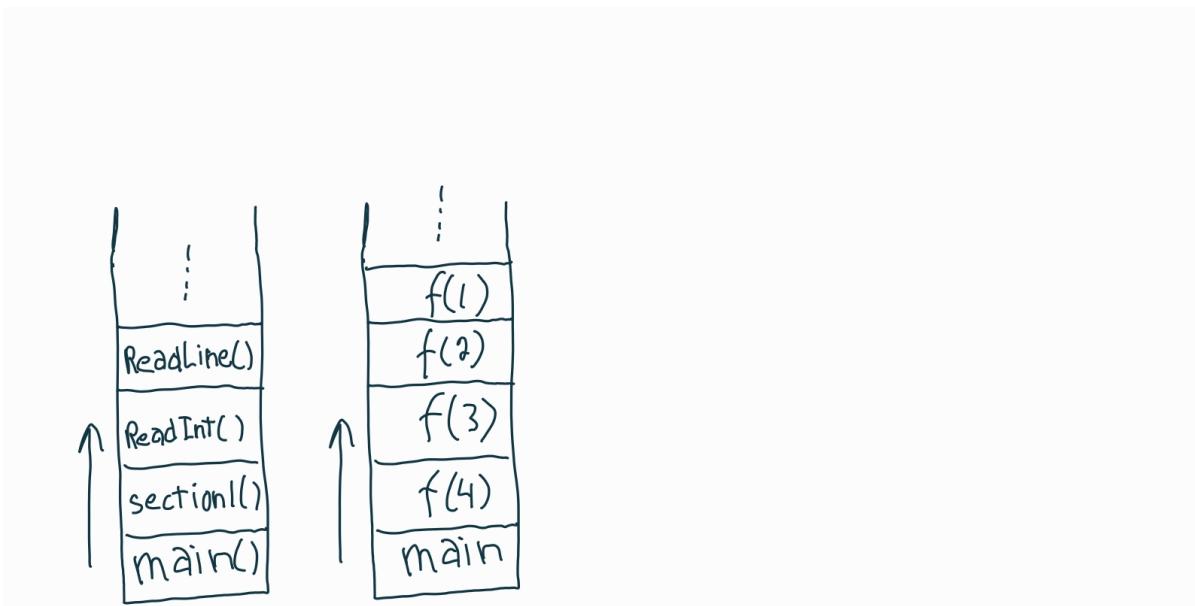


Pile et cadres d'exécution

Chaque rectangle est un cadre d'exécution pour chaque appel de fonction. Dans ce cadre, il y aura, entre autres, la mémoire nécessaire aux paramètres et aux variables locales (dans ce cas-ci, il y a seulement le paramètre `i`). À chaque appel de fonction, un cadre d'exécution est ajouté sur la pile d'exécution.

## Exemple 2

Si la fonction `main` appelle la fonction `section1`, qui elle-même appelle la fonction `ReadInt`, qui elle-même appelle la fonction `.ReadLine`, nous aurons à un moment donné, lors de l'exécution de `.ReadLine`, une pile d'appels de fonctions qui aura les cadres suivants :



Chaque cadre contiendra de l'espace mémoire pour ses variables locales et ses paramètres.

## Exercice

Exécutez les fonctions des 2 exemples avec le débugueur, ligne par ligne, en étudiant la composition de la pile d'exécution.

# Section 2 : Le tas

Le problème avec la mémoire allouée sur la pile d'exécution est qu'à la fin de l'exécution de la fonction, la mémoire est libérée avec le cadre d'exécution de la fonction. Donc une fonction peut partager de la mémoire avec les fonctions qu'elle appelle, en passant des pointeurs vers les variables à partager aux autres fonctions, et une fonction peut retourner une ou des valeurs à la fonction qui l'a appelée, mais ces valeurs rentrées seront copiées lors du retour de fonction (donc, pas vraiment partagées).

Pour partager de la mémoire plus librement, sans avoir besoin de la copier à plusieurs reprises, et sans avoir besoin de se préoccuper de son allocation sur la pile, on peut allouer de la mémoire sur le tas. La mémoire allouée sur le tas est gérée différemment de la mémoire allouée sur la pile, qui est directement liée à la durée d'exécution d'une fonction.

L'allocation de mémoire sur le tas (*heap*) est généralement gérée par le compilateur. Cela se fait principalement par l'intermédiaire du mot-clé `new` et l'opérateur `&` lors de l'initialisation d'une variable.

## Avec le mot-clé new

En Go, `new(T)` alloue de la mémoire sur le tas pour une nouvelle item de type `T`, initialise à zéro et renvoie un pointeur vers cet item. Voici un exemple :

```
i := new(int)
*i = 3
fmt.Println(*i) // Affiche "3"
```

Dans cet exemple, `new(int)` alloue de la mémoire pour un nouvel `int` sur le tas, initialisé à zéro, et renvoie un pointeur vers cet `int`. Ensuite, nous utilisons le `*` pour déréférencer le pointeur et assigner la valeur `3` à l'`int` sur le tas.

## Avec l'opérateur & lors de l'initialisation de variables

Cet opérateur est généralement utilisé pour obtenir l'adresse d'une variable existante. Cependant, lorsqu'il est utilisé pendant l'initialisation d'une variable (par exemple, lors de l'initialisation d'une `struct`), cela provoquera aussi l'allocation de mémoire sur le tas.

```
type Person struct {
    Name string
    Age  int
```

```
}
```

```
p := &Person{"Alice", 30}
fmt.Println(p) // Affiche l'adresse mémoire de la struct Person sur
le tas
```

Dans cet exemple, `&Person{"Alice", 30}` crée une nouvelle struct `Person` avec les valeurs fournies, alloue de la mémoire pour celle-ci sur le tas, et renvoie un pointeur vers cette struct.

Il est à noter que les détails de l'allocation de mémoire peuvent être assez compliqués en pratique, car le compilateur Go utilise une technique appelée "analyse d'échappement" pour décider où une variable devrait être allouée. Il existe des cas où une variable pourrait être allouée sur la pile même si vous avez utilisé `new` ou `&` lors de l'initialisation.

## Analyse d'échappement

L'analyse d'échappement (ou *Escape Analysis* en anglais) est un processus effectué par le compilateur pour déterminer si une certaine variable peut être allouée sur la pile au lieu du tas.

La logique principale derrière l'analyse d'échappement est la suivante : si la durée de vie d'une variable est limitée à l'exécution d'une seule fonction, alors le compilateur peut décider d'allouer cette variable sur la pile plutôt que sur le tas. Cela peut offrir un avantage de performance significatif, car l'accès à la pile est généralement plus rapide que l'accès à la mémoire du tas, et les objets de la pile ne nécessitent pas de garbage collection.

Néanmoins, si une variable est accessible (ou « s'échappe ») en dehors de la fonction où elle a été définie (par exemple, si elle est renvoyée par une fonction ou capturée à l'intérieur d'une fermeture), alors elle doit être allouée sur le tas.

Par exemple, dans le code Go suivant :

```
func foo() *Bar {
    b := Bar{}
    return &b
}
```

La variable `b` est allouée sur le tas parce qu'elle est renvoyée par la fonction `foo` et pourrait donc être utilisée ailleurs, en dehors de la portée de la fonction `foo`. Si `b` était allouée sur la pile, elle serait détruite dès que `foo` aurait terminé l'exécution, ce qui ne serait pas correct si une autre partie du code a besoin d'y accéder plus tard.

Le compilateur Go exécute l'analyse d'échappement pour déterminer si les variables doivent être allouées sur le tas ou sur la pile.

## Adresse sur la pile vs. sur le tas

Voici un exemple de code qui montre comment allouer une struct Person sur le tas et une sur la pile, puis imprime leur adresse pour comparaison :

```
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age int
}

func main() {
    // Allocation sur le tas
    pHeap := &Person{"Alice", 30}
    fmt.Printf("Address of person on heap: %p\n", pHeap)

    // Allocation sur la pile
    pStack := Person{"Bob", 35}
    fmt.Printf("Address of person on stack: %p\n", &pStack)
}
```

Dans ce code, pHeap est un pointeur vers une instance de Person qui est allouée sur le tas, et pStack est une instance de Person qui est allouée sur la pile. Les adresses de ces deux variables sont ensuite imprimées pour comparaison. Vous remarquerez probablement que l'adresse de la variable sur le tas est significativement plus grande que celle de la variable sur la pile. Les adresses précises varieront à chaque exécution du programme.

# Section 3 : Le vidangeur

Le *vidangeur* (*ramasse-miettes*, *garbage collector*) (GC) de Go a pour objectif de libérer automatiquement l'espace mémoire qui n'est plus utilisé par un programme, c'est-à-dire les objets qui ne sont plus accessibles par le code.

Le GC de Go est un GC à balayage concurrent, ce qui signifie qu'il peut fonctionner en même temps que votre code sans avoir besoin de l'arrêter complètement (ce qu'on appelle un "stop-the-world" GC).

Le processus de GC se déroule en trois phases principales :

- 1. Phase de marquage (Mark):** Le GC commence à la racine de l'arbre d'objets (par exemple, les variables globales, les piles de chaque goroutine, etc.) et suit chaque pointeur, marquant chaque objet atteint comme étant en cours d'utilisation. Cette phase est réalisée de manière concurrente avec l'exécution du programme, avec une courte pause au début et à la fin de la phase.
- 2. Phase d'assistance (Assist):** Pendant cette phase, le code de l'application aide le GC en participant activement à la phase de marquage. L'application consacre une partie de son temps CPU à la collecte des déchets, proportionnellement à l'allocation de mémoire qu'elle effectue.
- 3. Phase de balayage (Sweep):** Le GC parcourt tous les objets en mémoire et libère ceux qui n'ont pas été marqués comme étant en cours d'utilisation. Cela libère de l'espace pour de nouvelles allocations. Cette phase est également effectuée en arrière-plan, de manière concurrente avec l'application.

Le GC de Go a été conçu pour minimiser l'impact sur les performances de l'application. Il est également possible de contrôler le GC de manière plus fine par le biais de réglages et de la fonction `runtime.GC()` qui force une collecte de déchets.

Cependant, les détails exacts du fonctionnement du GC en Go peuvent varier entre les versions du langage et sont sujettes à des optimisations continues par les développeurs de Go.

# Section 4 : Les tranches

Les tranches (*slices*) sont une abstraction au-dessus des tableaux. Elles se composent de trois composants :

1. un pointeur vers un tableau,
2. une longueur et
3. une capacité.

Lorsqu'une tranche est créée, un tableau sous-jacent est également créé. La tranche pointe vers ce tableau, et la taille de la tranche ne peut jamais dépasser la taille du tableau sous-jacent.

Voici un exemple de création de tranche :

```
s := make([]int, 5, 10) // Crée une tranche de longueur 5 et de capacité 10
```

Dans cet exemple, un tableau de 10 éléments est créé dans le tas, et une tranche de longueur 5 est créée qui pointe vers les cinq premiers éléments de ce tableau. Les cinq éléments restants sont réservés pour permettre à la tranche de s'étendre.

Il est important de noter que les tranches, étant une structure de données dynamique, sont **presque toujours allouées sur le tas**, du moins pour le tableau sous-jacent, même si leur taille est connue au moment de la compilation.

Mais comme toujours en Go, les détails précis de l'allocation de mémoire peuvent dépendre de nombreux facteurs et peuvent être gérés par le *runtime* Go. Par exemple, pour les tranches très petites, Go pourrait décider de les allouer sur la pile au lieu du tas pour des raisons de performances. Cependant, ce sont des détails d'implémentation qui ne sont généralement pas exposés à l'utilisateur.

## Comment les tranches gèrent la mémoire

Lorsque vous créez une tranche, Go alloue dynamiquement un tableau sous-jacent et renvoie une tranche qui fait référence à une partie de ce tableau. Si vous augmentez la taille de la tranche au-delà de la capacité de ce tableau, Go alloue un nouveau tableau, copie les

éléments existants dans le nouveau tableau, puis met à jour la tranche pour qu'elle pointe vers le nouveau tableau. Ce processus est appelé **redimensionnement dynamique**.

Par exemple, lorsque vous utilisez `append` pour ajouter un élément à une tranche, si la tranche a de la capacité disponible dans son tableau sous-jacent, l'élément sera ajouté dans cet espace. Si la tranche est déjà au maximum de sa capacité, un nouveau tableau sera créé avec une capacité doublée, tous les éléments existants seront copiés, et le nouvel élément sera ajouté à la suite.

Il convient de noter que lorsqu'une tranche est redimensionnée, l'ancien tableau sous-jacent reste en mémoire jusqu'à ce qu'il soit éliminé par le *garbage collector* de Go. Cela peut entraîner une utilisation non optimale de la mémoire si le tableau sous-jacent d'une tranche est grand et que seul un petit sous-ensemble de celui-ci est encore nécessaire. Pour éviter une telle situation, vous pourriez utiliser la fonction `copy` et `runtime.GC` de Go pour copier la tranche dans un petit tableau sous-jacent et forcer le *garbage collector* à éliminer l'ancien grand tableau.

En résumé, les tranches en Go fournissent une abstraction flexible et facile à utiliser pour travailler avec des séquences de données, mais il est important de comprendre comment elles gèrent la mémoire sous-jacente pour éviter les écueils potentiels.

## Tranches de structures vs. tranches de pointeurs vers des structures

Travailler avec des tranches de structures ou des tranches de pointeurs vers des structures peut avoir un impact significatif sur la façon dont la mémoire est utilisée et gérée.

Considérons un exemple où `Person` est une structure avec quelques champs. Si vous créez une tranche de `Person`, chaque `Person` dans la tranche est stockée en tant que valeur. Cela signifie que si vous passez des éléments de la tranche à une fonction ou si vous les copiez dans une nouvelle tranche, vous créez une copie de ces structures. Cette copie représente un coût en termes de performances et d'utilisation de la mémoire, surtout si les structures sont grandes.

```
people := []Person{  
    {Name: "John", Age: 30},  
    {Name: "Jane", Age: 25},  
}
```

D'autre part, si vous créez une tranche de pointeurs vers des structures `Person`, vous ne stockez que les adresses mémoire des structures dans la tranche. Lorsque vous passez ces éléments à une fonction ou les copiez dans une nouvelle tranche, vous copiez seulement les pointeurs, pas les structures elles-mêmes. Cela peut être plus efficace en termes d'utilisation

de la mémoire et de performance, surtout pour les grandes structures. De plus, puisque vous travaillez avec des pointeurs, toute modification apportée à la structure à travers le pointeur sera reflétée dans toute votre application, car tous les pointeurs se réfèrent au même emplacement mémoire.

```
people := [] *Person{  
    &Person{Name: "John", Age: 30},  
    &Person{Name: "Jane", Age: 25},  
}
```

Cependant, il convient de noter que travailler avec des pointeurs peut compliquer la gestion de la mémoire. Par exemple, si une structure est supprimée à un endroit, mais que des pointeurs vers cette structure existent toujours ailleurs dans le programme, cela peut entraîner des bugs subtils et difficiles à repérer.

Il n'y a pas de règle stricte pour déterminer quand utiliser des tranches de structures ou des tranches de pointeurs à des structures. Cela dépend principalement de vos besoins spécifiques. Si la taille de votre structure est petite et que les effets de la copie de structures ne sont pas significatifs, les tranches de structures peuvent suffire. Si les structures sont grandes ou si vous avez besoin de modifications en place sur la structure à travers le programme, les tranches de pointeurs vers des structures peuvent être préférables.

# Opérations sur les tranches

Certaines opérations sur les tranches peuvent être faites **sur place** ou **en place** (*in-place*), ce qui va modifier le contenu des tranches, ou sinon elles peuvent créer une nouvelle tranche sans toucher à l'originale.

## Trier une tranche

Le code suivant définit une fonction appelée `section4a()`. Cette fonction effectue le tri de tranches de différents types (entier, et struct) de différentes façons. Ces opérations sont faites *sur place*, donc les tranches sont modifiées.

```
func section4a() {
    // trier une tranche de nombres entiers
    numbers1 := []int{2, 5, 1, 7, 3, 8, 4, 7, 6, 4}
    fmt.Println(numbers1)
    fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))

    sort.Ints(numbers1)
    fmt.Println(numbers1)
    fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))

    numbers1 = []int{2, 5, 1, 7, 3, 8, 4, 7, 6, 4}
    fmt.Println(numbers1)
    fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))
    sort.Slice(numbers1, func(i, j int) bool {
        return numbers1[i] < numbers1[j]
    })
    fmt.Println(numbers1)
    fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))

    numbers1 = []int{2, 5, 1, 7, 3, 8, 4, 7, 6, 4}
    fmt.Println(numbers1)
    fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))
    // trier en ordre décroissant
    sort.Slice(numbers1, func(i, j int) bool {
        return numbers1[i] > numbers1[j]
    })
}
```

```

fmt.Println(numbers1)
fmt.Println("Trié ?", sort.IntsAreSorted(numbers1))

// trier une tranche de struct Person
people := []Person{
    {Name: "John", Age: 30},
    {Name: "Jane", Age: 25},
    {Name: "Alice", Age: 35},
}
fmt.Println(people)
// trier par âge croissant
sort.Slice(people, func(i, j int) bool {
    return people[i].Age < people[j].Age
})
fmt.Println(people)
// trier par âge décroissant
sort.Slice(people, func(i, j int) bool {
    return people[i].Age > people[j].Age
})
fmt.Println(people)

// trier par nom croissant
sort.Slice(people, personNameCompare(people))
fmt.Println(people)
// trier par nom décroissant
sort.Slice(people, personNameCompareReverse(people))
fmt.Println(people)

}

func personNameCompareReverse(people []Person) func(i int, j int) bool {
    return func(i, j int) bool {
        return people[i].Name > people[j].Name
    }
}

func personNameCompare(people []Person) func(i int, j int) bool {
    return func(i, j int) bool {

```

```
        return people[i].Name < people[j].Name  
    }  
}
```

- 1. Tri d'une tranche d'entiers:** D'abord, elle crée une tranche d'entiers appelée `numbers1`, affiche la tranche, vérifie si elle est triée (en utilisant `sort.IntsAreSorted()`) et affiche le résultat. Elle trie ensuite la tranche en utilisant `sort.Ints()`, affiche de nouveau la tranche et vérifie si elle est triée. Ce processus est répété deux fois, la seconde fois en triant la tranche dans l'ordre décroissant avec une fonction lambda et `sort.Slice()`.
- 2. Tri d'une tranche de structs:** Ensuite, une tranche de structs `Person` est créée. Chaque `Person` a un nom et un âge. Elle affiche la tranche de personnes, la trie par âge en ordre croissant, l'affiche à nouveau, la trie par âge en ordre décroissant, et l'affiche une fois de plus. Elle trie ensuite la tranche de personnes par nom en ordre croissant, l'affiche, la trie par nom en ordre décroissant, et l'affiche une dernière fois.

Les deux fonctions `personNameCompare()` et `personNameCompareReverse()` sont utilisées pour générer des fonctions de comparaison pour le tri de slices. La première fonction retourne une fonction qui compare deux personnes en fonction de leur nom dans l'ordre croissant, tandis que la seconde fonction retourne une fonction de comparaison qui compare deux personnes en fonction de leur nom dans l'ordre décroissant.

## Fonctions : citoyennes de première classe

En programmation, lorsque nous parlons de **citoyenneté de première classe**, nous faisons référence à des entités qui peuvent être manipulées de toutes les façons dont les autres entités peuvent l'être. Si nous disons que les fonctions sont des *citoyennes de première classe* dans un langage de programmation comme Go, cela signifie que les fonctions peuvent être utilisées comme n'importe quel autre type de données.

En Go, les fonctions sont des citoyens de première classe, car elles peuvent être utilisées de manière interchangeable avec les autres types de données. Elles peuvent être affectées à des variables, passées en paramètre à d'autres fonctions, renvoyées par d'autres fonctions, déclarées dans une fonction et même faire partie de structures de données comme les tranches ou les maps.

Dans le code précédent, nous voyons des exemples de cela. Par exemple, les deux fonctions `personNameCompare()` et `personNameCompareReverse()` retournent des fonctions. Ces fonctions renvoyées sont ensuite utilisées comme arguments pour la fonction `sort.Slice()`. C'est un exemple de traitement des fonctions comme des citoyens de première classe : dans

ce cas, des fonctions sont retournées par d'autres fonctions et passées à d'autres fonctions. C'est une caractéristique puissante de Go que l'on retrouve dans d'autres langages fonctionnels.

## Fonctions lambda

Les fonctions lambda (aussi connues sous le nom de fonctions anonymes) en Go sont des fonctions qui ne sont définies avec aucun nom et peuvent être utilisées pour créer des définitions de fonctions en ligne. Il s'agit de définir une fonction juste où vous en avez besoin. En d'autres termes, une fonction lambda est une fonction déclarée sans nom associé.

Les fonctions lambda sont très utiles pour réaliser des opérations simples et cela évite également d'encombrer le code avec de multiples petites fonctions. Elles sont souvent utilisées comme arguments pour des fonctions qui prennent des fonctions en paramètre.

Reprendons le code que nous avons discuté précédemment pour illustrer cela. Par exemple, lorsque nous trions la tranche d'entiers `numbers1` ou la tranche `people` de structures `Person`, nous utilisons des fonctions lambda. Voici l'extrait de code correspondant :

```
sort.Slice(numbers1, func(i, j int) bool {
    return numbers1[i] < numbers1[j]
})

sort.Slice(people, func(i, j int) bool {
    return people[i].Age < people[j].Age
})
```

Dans chacun de ces appels à `sort.Slice()`, le deuxième argument est une fonction lambda qui est utilisée pour déterminer l'ordre de tri. Ces fonctions sont définies sur place, et elles n'ont pas de nom. Ce sont des fonctions lambda.

## Filtrer une tranche

Le code suivant fourni illustre plusieurs opérations de filtrage sur une tranche de structures `Person`. Le but est de démontrer différentes manières de filtrer les données dans un tableau de `Person` selon différentes conditions, notamment l'âge et les caractéristiques du nom.

```
func section4b() {
    people := []Person{
        {Name: "John", Age: 30},
        {Name: "Jane", Age: 25},
```

```

        {Name: "Alice", Age: 35},
    }
fmt.Println(people)

// filtrer une tranche de struct Person
// filtre : plus de 30 ans
filteredPeople := make([]Person, 0)
for _, person := range people {
    if person.Age > 30 {
        filteredPeople = append(filteredPeople, person)
    }
}
fmt.Println(filteredPeople)

// filtrer une tranche de struct Person
// filtre : nom commençant par "J"
filteredPeople = make([]Person, 0)
for _, person := range people {
    if person.Name[0] == 'J' {
        filteredPeople = append(filteredPeople, person)
    }
}
fmt.Println(filteredPeople)

// filtrer une tranche de struct Person
// filtre : nom contenant plus de 5 caractères
filteredPeople = make([]Person, 0)
for _, person := range people {
    if len(person.Name) > 5 {
        filteredPeople = append(filteredPeople, person)
    }
}
fmt.Println(filteredPeople)
}

```

Tout d'abord, une structure Person est déclarée avant la fonction principale :

```

type Person struct {
    Name string

```

```
Age int  
}
```

Cette structure représente une personne avec un nom (type `string`) et un âge (type `int`).

La fonction `section4b` crée d'abord un tableau de `Person` nommé `people`:

```
people := []Person{  
    {Name: "John", Age: 30},  
    {Name: "Jane", Age: 25},  
    {Name: "Alice", Age: 35},  
}
```

Ensuite, il y a plusieurs sections de code qui filtrent ce tableau `people` en fonction de différentes conditions et impriment le résultat.

Voici le premier filtre :

```
filteredPeople := make([]Person, 0)  
for _, person := range people {  
    if person.Age > 30 {  
        filteredPeople = append(filteredPeople, person)  
    }  
}  
fmt.Println(filteredPeople)
```

Cette section filtre toutes les personnes qui ont plus de 30 ans. Le tableau `filteredPeople` est initialement vide. Pour chaque `Person` dans le tableau `people`, si l' `Age` est supérieur à 30, cette `Person` est ajoutée au tableau `filteredPeople`. Le résultat est ensuite imprimé.

Les deux sections suivantes suivent la même procédure, mais avec différentes conditions :

- La deuxième section de filtrage ajoute à `filteredPeople` toutes les `Person` dont le nom commence par la lettre 'J'.
- La troisième section de filtrage ajoute les personnes dont le nom comporte plus de 5 caractères.

## Version améliorée

En examinant le code précédent, il y a un patron répété de création d'une liste filtrée de personnes en fonction de différentes conditions. C'est une bonne occasion d'appliquer la refonte **Extraire la fonction**. En extrayant la logique répétée dans une fonction distincte qui prend une fonction de filtrage comme l'un de ses paramètres, vous pouvez éliminer la répétition du code tout en rendant le code plus facile à comprendre et à modifier.

Le code qui suit est une réécriture de l'exemple précédent qui réduit la répétition.

```
func section4c() {
    people := []Person{
        {Name: "John", Age: 30},
        {Name: "Jane", Age: 25},
        {Name: "Alice", Age: 35},
    }
    fmt.Println(people)

    filteredPeople := filterPeople(people, func(person Person) bool {
        return person.Age > 30
    })
    fmt.Println(filteredPeople)

    filteredPeople = filterPeople(people, func(person Person) bool {
        return person.Name[0] == 'J'
    })
    fmt.Println(filteredPeople)

    filteredPeople = filterPeople(people, func(person Person) bool {
        return len(person.Name) > 5
    })
    fmt.Println(filteredPeople)
}

type FilterFunc func(person Person) bool

func filterPeople(people []Person, filter FilterFunc) []Person {
    filteredPeople := make([]Person, 0)
    for _, person := range people {
        if filter(person) {
            filteredPeople = append(filteredPeople, person)
        }
    }
    return filteredPeople
}
```

```
        }
    }

    return filteredPeople
}
```

FilterFunc est un nouveau type pour une fonction qui prend une Person et renvoie un bool, définissant si cette personne doit être incluse dans les résultats filtrés.

Donc, dans chacun des appels à filterPeople, une nouvelle fonction anonyme est créée qui spécifie la logique de filtrage pour cette liste particulière.

Cette refactorisation élimine la redondance dans votre code, et rend également le code plus facile à comprendre : au lieu de devoir lire les spécificités de chaque boucle pour comprendre ce qui se passe, vous pouvez maintenant voir facilement que chaque séquence filtre la liste people selon un critère distinct. De plus, si vous souhaitez ajouter une nouvelle condition pour filtrer les personnes, vous avez simplement besoin d'ajouter un nouvel appel à filterPeople avec la fonction de filtrage appropriée.

## Autre amélioration

Si désiré, à la place d'utiliser des fonctions de filtrage anonymes, les fonctions de filtrage pourraient être définies un peu à la manière de la fonction personNameCompareReverse de l'exemple de tri précédent.

```
func section4d() {
    people := []Person{
        {Name: "John", Age: 30},
        {Name: "Jane", Age: 25},
        {Name: "Alice", Age: 35},
    }
    fmt.Println(people)

    filteredPeople := filterPeople(people, peopleFilterMinAge(31))
    fmt.Println(filteredPeople)
}

func peopleFilterMinAge(minAge int) FilterFunc {
    return func(person Person) bool {
        return person.Age >= minAge
}
```

```
    }  
}
```

Ce code poursuit la tendance de refactorisation et d'amélioration de la lisibilité en utilisant le filtrage des personnes.

Commençons par examiner la nouvelle fonction `peopleFilterMinAge`.

Elle définit une fonction `peopleFilterMinAge` qui prend un âge minimum `minAge` comme argument et renvoie une fonction de type `FilterFunc` qui, à son tour, accepte une personne et renvoie si l'âge de cette personne est supérieur ou égal à `minAge`.

Nous pouvons voir le rendement de cette fonction utilisée dans un filtre dans la fonction `section4d`:

```
filteredPeople := filterPeople(people, peopleFilterMinAge(31))
```

Ici, `peopleFilterMinAge(31)` renvoie une fonction qui sera utilisée comme critère de filtre par `filterPeople`. Elle vérifiera si l'âge de chaque personne dans le tableau `people` est supérieur ou égale à 31. Si c'est le cas, cette personne sera incluse dans la liste `filteredPeople`.

En résumé, `section4d` utilise une nouvelle fonction `peopleFilterMinAge` pour créer une fonction de filtre dynamique basée sur un âge minimum, puis utilise cette fonction de filtre pour obtenir une liste de personnes dont l'âge est supérieur ou égal à l'âge minimum. Ceci donne encore plus de flexibilité au code, car on peut maintenant facilement créer différents filtres d'âge en appelant simplement `peopleFilterMinAge` avec l'âge minimum souhaité en paramètre. On pourrait poursuivre avec des définitions similaires pour d'autres critères.

# Chapitre 7 : Structures de données

Les structures de données sont une manière d'organiser, de stocker et de manipuler des données afin qu'elles soient efficacement accessibles et utilisables. Elles forment la base de presque tous les systèmes logiciels et algorithmes, car la manière dont les données sont structurées peut avoir un effet significatif sur la performance et sur le fonctionnement du programme.

Il existe plusieurs types de structures de données, qui peuvent être classés de nombreuses manières. Toutefois, on peut les diviser en deux catégories principales : les structures de données linéaires et non linéaires.

- Les structures de données **linéaires** sont celles dans lesquelles les éléments sont arrangés d'une manière séquentielle et chaque élément est connecté au précédent et au suivant. Quelques exemples de ce type sont les tableaux, les listes chaînées, les piles et les files d'attente.
  - Les **tableaux** sont des structures de données qui stockent des éléments du même type dans une séquence contiguë.
  - Les **listes chaînées** sont similaires, mais chaque élément (noeud) contient un lien vers le prochain noeud, permettant un accès plus flexible et dynamique aux éléments.
  - Les **piles** et les **files d'attente** sont des structures de données qui imposent un ordre spécifique pour l'ajout et la suppression d'éléments, généralement *dernier entré, premier sorti* (**LIFO**) pour les piles et *premier entré, premier sorti* (**FIFO**) pour les files.
- Les structures de données **non linéaires** sont celles dans lesquelles les éléments ne sont pas disposés de façon linéaire. Ils incluent les arbres, les graphes, les hachages et d'autres types.
  - Les **arbres** sont composés de noeuds liés où un noeud parent peut avoir de multiples noeuds enfants, mais chaque noeud enfant ne peut avoir qu'un seul noeud parent.
  - Les **graphes** étendent cette idée en permettant aux noeuds d'avoir des liens vers plusieurs autres noeuds.
  - Les **dictionnaires** ou **tableaux associatifs** (ou *maps* en anglais) stockent des éléments dans des emplacements basés sur une clé dérivée d'une fonction de hachage, permettant un accès direct aux valeurs en utilisant ces clés.

Choisir la bonne structure de données dépend du problème spécifique que vous êtes en train de résoudre. Par exemple, si vous avez besoin d'un accès constant à des éléments par leur index, un tableau serait approprié, mais si vous avez besoin d'ajouter et de supprimer fréquemment des éléments, une liste chaînée ou une pile pourrait être préférable. Comprendre la nature de ces différentes structures de données et la façon dont elles fonctionnent vous permettra de concevoir des logiciels plus efficaces, plus propres et plus adaptés à vos objectifs.

## Structures de données en Go

Le langage Go offre plusieurs types de structures de données intégrés sans avoir besoin d'importer de paquets supplémentaires. Les principaux sont :

- 1. Les tableaux:** Un tableau en Go est une collection ordonnée d'éléments de même type. Sa taille doit être connue au moment de la déclaration.

```
var arr [10]int // Un tableau contenant 10 entiers.
```

- 2. Les chaînes de caractères (string):** Une *string* est une séquence de caractères. En Go, les chaînes de caractères sont codées en UTF-8 et peuvent contenir n'importe quel caractère Unicode.

```
s := "hello, world"
```

- 3. Les tranches (tableaux dynamiques):** Les slices sont similaires aux tableaux, mais leur taille peut changer. Ils fournissent un moyen plus flexible de travailler avec des collections de même type.

```
sl := []int{1, 2, 3} // Une slice d'entiers.
```

- 4. Les dictionnaires (tableaux associatifs, ou maps):** En Go, une map est une structure de données qui associe des valeurs à des clés uniques, similaires aux dictionnaires en Python, aux *HashMap* ou *TreeMap* en Java, ou aux objets en JavaScript.

```
m := map[string]int{ // Une map qui associe des strings à des
                     entiers.
    "one": 1,
```

```
"two": 2,  
}
```

# Section 1 : Les dictionnaires (map)

Dans les sciences de l'informatique, un **dictionnaire**, également connu sous le nom de *map* et de *tableau associatif*, est une structure de données qui stocke les valeurs en paires de clés et de valeurs. Chaque clé unique est associée à une valeur, ce qui permet de récupérer efficacement la valeur à l'aide de la clé correspondante.

L'un des principaux avantages d'un dictionnaire est qu'il permet un accès rapide à ses éléments. Dans de nombreux cas, l'accès, l'insertion et la suppression d'un élément sont tous effectués en temps constant, indépendamment de la taille du dictionnaire. Cela contraste avec des structures de données telles que les listes ou les tableaux, où le temps nécessaire pour trouver un élément peut augmenter à mesure que la taille de la structure de données augmente.

Dans les dictionnaires, la clé joue un rôle significatif, car elle est utilisée pour accéder à la valeur correspondante, et chaque clé est unique. Cela signifie que si vous essayez d'insérer une nouvelle clé qui existe déjà dans le dictionnaire, la valeur existante de cette clé sera remplacée par la nouvelle valeur.

Les dictionnaires sont utilisés dans un grand nombre d'applications. Par exemple, ils sont souvent utilisés pour implémenter la fonctionnalité de recherche rapide dans les bases de données, où la clé serait un identifiant unique et la valeur serait l'enregistrement associé à cet identifiant. D'autres utilisations courantes des dictionnaires incluent la détection de duplications, le comptage d'occurrences d'éléments, ou le stockage de configuration et de métadonnées.

Il est important de noter que les dictionnaires, en raison de leur fonctionnement interne, *ne maintiennent pas d'ordre spécifique des éléments* stockés, sauf dans des certaines implémentations spécifiques, comme le `TreeMap` de Java. Par conséquent, si vous avez besoin d'une structure de données ordonnée, d'autres options, comme les listes triées ou les arbres, pourraient être plus appropriées.

En résumé, les dictionnaires ou les *maps* sont des structures de données associatives puissantes et flexibles qui opèrent sur des paires de clés et de valeurs, offrant un accès rapide et des opérations efficaces sur les données.

## Les maps en Go

Les maps sont une structure de données fondamentale dans Go, souvent utilisées lorsque vous souhaitez stocker des associations clé/valeur, similaires aux dictionnaires en Python, au

HashMap en Java et aux objets en JavaScript.

Voici comment déclarer une map en Go :

```
var myMap map[string]int
```

Ceci déclare une map appelée myMap qui a des clés de type string et des valeurs de type int.

Veuillez noter que déclarer une map de cette manière l'initialise à nil. Pour assigner une nouvelle map vide à myMap, nous devons utiliser la fonction make:

```
myMap = make(map[string]int)
```

On peut déclarer et initialiser une map en une seule déclaration :

```
myMap := make(map[string]int)
```

Pour ajouter des valeurs à notre map :

```
myMap["One"] = 1
myMap["Two"] = 2
myMap["Three"] = 3
```

Pour récupérer une valeur de la map :

```
value := myMap["Two"] // value est maintenant égal à 2
```

Si vous essayez d'accéder à une clé qui n'existe pas dans la map, Go retournera la valeur zéro pour le type de valeur de la map. Pour un int, cela donnerait 0.

Pour vérifier si une clé existe dans la map, vous pouvez utiliser la déclaration if:

```
val, exists := myMap["Four"]
if exists {
    fmt.Println("La valeur existe et est", val)
} else {
    fmt.Println("La valeur n'existe pas")
}
```

Pour supprimer une entrée de la map, utilisez la fonction delete:

```
delete(myMap, "Two")
```

Cela supprime l'entrée avec la clé "Two" de la map. Si la clé n'existe pas dans la map, `delete` ne fait rien et le programme continue à fonctionner.

Enfin, pour parcourir une map, vous pouvez utiliser une boucle `for`:

```
for key, value := range myMap {
    fmt.Printf("Clé : %s, Valeur : %d\n", key, value)
}
```

Ici, `range` sur une map retourne deux valeurs à chaque itération, la clé et la valeur.

## Exemple 1

```
var menuStrings = make(map[string]string)

func ShowMenu(name string) {

    if _, ok := menuStrings[name]; !ok {
        menuBytes, err := os.ReadFile("menu/" + name + ".txt")
        if err != nil {
            log.Fatalf("lecture du contenu du menu impossible : %s",
err)
        }
        menuStrings[name] = string(menuBytes)
    }
    fmt.Println(menuStrings[name])
}
```

- Tout d'abord, une variable globale `menuStrings` est déclarée qui est une map avec des clés `string` et des valeurs `string`. Cette variable est initialisée avec une map vide à l'aide de la fonction `make`.
- Ensuite, une fonction `ShowMenu` est définie, qui prend un paramètre `name` de type `string`.
- Dans cette fonction, il y a une vérification pour voir si une clé correspondant à `name` existe déjà dans la map `menuStrings`. La syntaxe `_, ok := menuStrings[name]` effectue une recherche de la clé dans la map et renvoie deux valeurs : la valeur du dictionnaire à la clé, et

un booléen indiquant si la clé a été trouvée dans la map. Le symbole `_` est un identifiant vide en Go, qui est utilisé lorsque vous voulez ignorer une valeur renournée. Ici, nous ignorons la valeur renvoyée par la map et nous nous intéressons seulement à la valeur booléenne `ok`.

- Si la clé n'existe pas dans `menuStrings` (c'est-à-dire que `ok` est `false`), alors le programme tente de lire un fichier texte du répertoire `menu/` ayant le nom dans la variable `name` avec l'extension `.txt`. L'opération de lecture de fichier peut échouer pour diverses raisons (par exemple si le fichier n'existe pas), et dans ce cas une erreur `err` est renvoyée.
- Si une erreur est produite lors de la lecture du fichier, un message d'erreur est affiché et le programme se termine avec `log.Fatalf`. Sinon, le contenu du fichier, qui est un tableau de bytes (`menuBytes`), est converti en `string` et stocké dans la map `menuStrings` avec la clé étant `name`.
- Enfin, que la valeur ait été lue à partir du fichier ou qu'elle existait déjà dans la map, la fonction `fmt.Println` est utilisée pour afficher la chaîne correspondante à la clé `name` de la map `menuStrings`.

En somme, ce petit programme effectue une opération de mise en cache des menus basée sur les noms de fichier. Chaque fois que vous demandez un menu par son nom, le programme vérifie d'abord s'il a déjà lu le fichier du disque. Si c'est le cas, il utilise la version en cache. Sinon, il lit le fichier du disque, l'ajoute au cache, et l'utilise.

## Exemple 2

```
package main

import "fmt"

type Person struct {
    Id    int
    Name string
    Age   int
}

func (p Person) String() string {
    return fmt.Sprintf("ID: %d, Nom : %s, Âge : %d\n", p.Id, p.Name,
    p.Age)
```

```

}

func main() {
    // Création des personnes
    person1 := &Person{Id: 1, Name: "John Doe", Age: 30}
    person2 := &Person{Id: 2, Name: "Jane Doe", Age: 28}
    person3 := &Person{Id: 3, Name: "Richard Roe", Age: 33}
    person4 := &Person{Id: 4, Name: "Maggie Roe", Age: 32}

    // Création de la map pour stocker les personnes par leur ID
    persons := make(map[int]*Person)

    // Ajout des personnes dans la map
    persons[person1.Id] = person1
    persons[person2.Id] = person2
    persons[person3.Id] = person3
    persons[person4.Id] = person4

    // Affichage des personnes de la map
    for _, person := range persons {
        fmt.Println(person)
    }
}

```

Dans cet exemple, quatre instances de Person sont créées (elles sont allouées avec & pour retourner un pointeur vers l'instance). Ensuite une map appelée persons est créée avec make(map[int]\*Person). Cette map associe des entiers (les identifiants des personnes) à des pointeurs de Person.

Chaque personne est ajoutée à la map avec son ID comme clé et elle-même comme valeur (persons[person1.Id] = person1).

Enfin, une boucle for est utilisée pour parcourir la map persons et chaque personne est affichée avec son ID et ses détails.

# Section 2 : Les listes chaînées

Start typing here...

# Chapitre 8 : Fils d'exécution

Les *threads*, aussi connus sous le nom de **fils d'exécution**, sont une façon pour un programme d'effectuer plusieurs tâches simultanément dans le même espace de processus. Un processus peut contenir plusieurs *threads*, tous partageant les mêmes ressources, comme la mémoire, mais pouvant s'exécuter indépendamment les uns des autres.

Chaque *thread* dans un processus fonctionne comme un flux séquentiel unique d'exécution, signifiant qu'il suivra une séquence d'instructions dans un ordre spécifié. Cela permet aux applications d'exécuter diverses tâches de manière simultanée, comme gérer des frappes au clavier pendant la lecture d'un disque dur.

Les *threads* ont deux types principaux : les *threads* utilisateur et les *threads* noyau. Les *threads* utilisateur sont gérés par l'application, et le système d'exploitation n'en a pas connaissance. D'autre part, les *threads* noyau sont reconnus par le système d'exploitation et ont accès à ses ressources.

**Avoir plusieurs *threads* dans un programme peut améliorer l'efficacité et la performance**, car plusieurs tâches peuvent être effectuées en parallèle. Cependant, cela peut également **amener des problèmes de complexité, de synchronisation et de concurrence**. Par exemple, deux threads peuvent tenter d'accéder à la même ressource simultanément, provoquant une **situation de compétition** (un état de *race condition*).

Il convient de noter que, bien que les *threads* puissent offrir une efficacité significative, leur utilisation et leur gestion nécessitent une conception soigneuse pour **éviter des problèmes comme les conditions de concurrence, les interblocages et les problèmes de synchronisation**.

Dans ce chapitre, nous verrons les bases de l'utilisation de fils d'exécution dans le langage Go. Mais avant, il est bon de revenir sur la nature des fonctions en Go, et les différentes façons d'utiliser les fonctions.

# Section 1 : Retour sur la nature des fonctions

## Les fonctions sont des citoyens de première classe

Quand on dit que \_les fonctions sont des citoyens de première classe\_ dans un langage de programmation, cela signifie que les fonctions peuvent être utilisées de la même manière que n'importe quel autre type de données du langage. Cela peut inclure :

- La capacité d'affecter des fonctions à des variables ou de les stocker dans des structures de données.
- La possibilité de passer une fonction en tant que paramètre à une autre fonction.
- La possibilité de retourner une fonction en tant que résultat d'une autre fonction.

En bref, une fonction est traitée comme une entité qui peut être manipulée par la syntaxe du langage, tout comme un entier, une chaîne ou un objet. Ce concept est un élément clé des langages de programmation fonctionnelle, mais il est également présent dans de nombreux langages de programmation impératifs modernes.

En ce qui concerne le langage Go, il traite également les fonctions comme des citoyens de première classe. Voici un exemple dans Go :

```
// Déclaration d'une fonction qui retourne une autre fonction
func createAdder(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}

// Utilisation de cette fonction
adder := createAdder(10)
fmt.Println(adder(5)) // Affiche 15
```

Dans cet exemple, non seulement une fonction est retournée par une autre fonction, mais la fonction interne est une fermeture (*closure* en anglais), elle a également accès à l'argument de la fonction externe.

## Fermeture

Une fermeture, ou *closure* en anglais, est une fonction qui a accès aux variables de son contexte lexical, c'est-à-dire à la fonction ou l'environnement de code dans lequel elle a été définie. Une fermeture est capable de "se rappeler" et de manipuler ces variables, même après que la fonction dans laquelle elle a été définie ait terminé son exécution.

Voyons cela avec un exemple en Go :

```
package main

import "fmt"

// Cette fonction crée une fermeture
func createAdder(x int) func(int) int {
    // La fonction renournée est une fermeture
    return func(y int) int {
        // Elle "se souvient" de la valeur de x
        return x + y
    }
}

func main() {
    adder := createAdder(10)
    fmt.Println(adder(5)) // Affiche 15
}
```

Dans cet exemple, la fonction `createAdder` retourne une autre fonction. Cette fonction renournée est une fermeture, parce qu'elle a accès à la variable `x` de `createAdder`. Même après que `createAdder` ait fini de s'exécuter et ait renvoyé la fermeture, la fermeture peut toujours accéder à `x`.

En d'autres termes, la fermeture capte, ou "emprisonne" son environnement lexical, c'est-à-dire l'ensemble des variables auxquelles elle peut accéder au moment de sa création. Ce mécanisme est très puissant et est la base de nombreuses constructions de programmation fonctionnelle.

Cependant, avec une grande puissance vient une grande responsabilité. Il est important d'être attentif lorsque vous travaillez avec des fermetures, car elles peuvent parfois conduire à des

problèmes de performance ou de mémoire si elles sont utilisées de manière imprudente, par exemple en capturant accidentellement de grosses structures de données.

## Autre exemple

En Go, vous pouvez aussi passer des fonctions en paramètres à d'autres fonctions et les stocker dans des structures de données. Cela facilite l'écriture de fonctions de haut niveau qui peuvent manipuler les opérations basées sur d'autres fonctions.

Voici un autre exemple illustrant comment une fonction peut accepter une autre fonction en paramètre en Go.

Supposons que nous ayons une fonction qui parcourt une tranche de nombres et applique une certaine fonction à chaque nombre :

```
package main

import "fmt"

// Notre fonction prend une tranche et une fonction en paramètre
func apply(nums []int, fn func(int) int) []int {
    result := make([]int, len(nums))
    for i, num := range nums {
        result[i] = fn(num)
    }
    return result
}

func main() {
    nums := []int{1, 2, 3, 4, 5}

    // Nous passons une fonction anonyme qui multiplie chaque nombre
    // par 2
    doubled := apply(nums, func(n int) int {
        return n * 2
    })

    fmt.Println(doubled) // Affiche : [2 4 6 8 10]
}
```

Dans cet exemple, la fonction `apply` prend une fonction `fn` comme argument. Cette fonction `fn` est ensuite appliquée à chaque élément de la liste `nums`. Ceci illustre que les fonctions en Go sont effectivement des citoyens de première classe - elles peuvent être passées comme arguments à d'autres fonctions.

## Différence entre appeler une fonction et passer une fonction en paramètre

La différence entre appeler une fonction et passer le résultat en paramètre à une fonction, et passer une fonction, sans l'appeler, à une autre fonction, peut être une source de confusion courante. La différence principale réside dans le moment où la fonction est exécutée.

Si vous appelez une fonction et passez son résultat en tant que paramètre, cette fonction sera exécutée immédiatement, avant que l'autre fonction ne soit appelée. Seule la valeur renvoyée par cette fonction est passée en paramètre.

Voici un exemple en Go :

```
package main

import "fmt"

func multiplyByTwo(n int) int {
    return n * 2
}

func printNumber(n int) {
    fmt.Println(n)
}

func main() {
    // La fonction multiplyByTwo est appelée immédiatement avec 10
    // comme argument
    // printNumber reçoit uniquement le résultat de cette fonction,
    // soit 20.
    printNumber(multiplyByTwo(10))
}
```

Cependant, si vous passez une fonction en tant que paramètre à une autre fonction sans l'appeler, ce que vous transmettez est une référence à la fonction elle-même, et non le

résultat de son exécution. Cela vous permet d'appeler cette fonction à un moment ultérieur dans le contexte de la deuxième fonction.

Exemple :

```
package main

import "fmt"

func multiplyByTwo(n int) int {
    return n * 2
}

func printAfterOperation(n int, fn func(int) int) {
    fmt.Println(fn(n))
}

func main() {
    // Ici, multiplyByTwo est passé en tant que référence de fonction
    // à printAfterOperation
    // Il ne sera exécuté qu'à l'intérieur de printAfterOperation,
    // avec 10 comme argument
    printAfterOperation(10, multiplyByTwo)
}
```

Dans ce deuxième exemple, nous n'évaluons pas immédiatement `multiplyByTwo`, mais nous la passons plutôt comme une fonction à `printAfterOperation`. Cette deuxième fonction peut ensuite l'appeler avec ses propres arguments.

# Section 2 : Goroutines

Les **goroutines** sont une des caractéristiques du langage Go, qui facilitent la programmation concurrente. Une goroutine est une sorte de *thread* léger, mais géré par le *runtime* de Go et non par le système d'exploitation. Vous pouvez démarrer une goroutine simplement en utilisant le mot clé `go` devant un appel de fonction.

## Exemple : avec `time.Sleep`

Voici un exemple simple qui illustre l'utilisation de goroutines :

```
package main

import (
    "fmt"
    "time"
)

func countdown(n int, id int) {
    for i := n; i >= 0; i-- {
        fmt.Printf("goroutine %d : %d\n", id, i)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    for i := 1; i <= 4; i++ {
        go countdown(5, i) // Lancer la goroutine
    }
    time.Sleep(6 * time.Second) // Laisser du temps pour l'exécution
    des goroutines
}
```

Dans cet exemple, `countdown` est une fonction qui affiche le décompte de `n` à 0. En préfixant l'appel de fonction `countdown(5, i)` par le mot clé `go` dans la boucle de la fonction `main`, Go créera une nouvelle goroutine pour chaque appel de fonction et commencera son exécution de manière concurrente.

Le `time.Sleep` à la fin de la fonction `main` est nécessaire pour empêcher que le programme principal ne se termine avant que les goroutines aient fini leur décompte. En vrai code, vous devriez utiliser des mécanismes de synchronisation comme des groupes d'attentes (*WaitGroups*) ou des canaux (*channels*) pour contrôler l'exécution de vos goroutines, mais pour un exemple simple comme celui-ci, un simple `time.Sleep` suffit pour illustrer le principe. À noter que l'ordre d'exécution précis des goroutines n'est pas garanti et peut varier à chaque exécution. C'est l'un des défis de la programmation concurrente : la nécessité de bien synchroniser les différentes tâches pour éviter des conflits ou des comportements imprévisibles.

## Exemple : avec canal

```
package main

import (
    "fmt"
    "time"
)

// Nous ajoutons un canal done de type bool
func countdown(n int, id int, done chan<- bool) {
    for i := n; i >= 0; i-- {
        fmt.Printf("goroutine %d : %d\n", id, i)
        time.Sleep(1 * time.Second)
    }
    done <- true // Envoie true sur le canal quand la goroutine a terminé
}

func main() {
    done := make(chan bool, 4) // Crée un canal de booléens avec une capacité de 4

    for i := 1; i <= 4; i++ {
        go countdown(5, i, done) // démarre une goroutine et lui passe le canal done
    }
}
```

```

for i := 1; i <= 4; i++ {
    <-done // Attend un vrai de chaque goroutine
}
}

```

Dans cet exemple, nous avons modifié la déclaration du canal `done` pour qu'il soit un canal de `bool`. Nous avons aussi ajouté une capacité à la création du canal pour éviter un blocage si les goroutines finissent leur décompte avant que `main` commence à recevoir sur le canal. Ensuite, nous envoyons `true` sur le canal `done` dans chaque goroutine après qu'elle a fini son décompte. Enfin, à la fin de `main`, nous recevons de chaque goroutine sur le canal `done` pour attendre qu'elle ait fini.

## Exemple avec WaitGroup

Le package `sync` de Go fournit une structure `WaitGroup` qui peut être utilisée pour synchroniser plusieurs goroutines. Voici comment on pourrait modifier notre exemple pour utiliser un `WaitGroup`:

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func countdown(n int, id int, wg *sync.WaitGroup) {
    defer wg.Done() // Signale au WaitGroup que cette goroutine est
    terminée lorsque la fonction se termine
    for i := n; i >= 0; i-- {
        fmt.Printf("goroutine %d : %d\n", id, i)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    var wg sync.WaitGroup // Crée un WaitGroup

    for i := 1; i <= 4; i++ {

```

```
    wg.Add(1) // Augmente le compteur du WaitGroup de 1
    go countdown(5, i, &wg) // Démarré une goroutine avec le
WaitGroup
}

wg.Wait() // Attend que toutes les goroutines signalent qu'elles
sont terminées
}
```

La méthode `Add` est utilisée pour augmenter le compteur du `WaitGroup`. Chaque fois que nous lançons une goroutine, nous augmentons le compteur de 1. La méthode `Done` est appelée pour diminuer le compteur lorsque la goroutine a fini son travail. Enfin, la méthode `Wait` est utilisée pour bloquer la goroutine actuelle (dans ce cas, la goroutine principale) jusqu'à ce que le compteur du `WaitGroup` atteigne zéro, c'est-à-dire que toutes les goroutines qu'il attend ont signalé qu'elles sont terminées. Cela nous permet d'attendre de manière sûre et flexible l'achèvement de toutes les goroutines.

# Section 3 : Canaux

En informatique, un **canal** (ou *channel* en anglais) est un mécanisme de communication utilisé pour transférer des données entre des *threads* concurrents, ou plus généralement entre des blocs de code concurrents. Les canaux sont particulièrement utiles dans les systèmes de programmation concurrente et parallèle pour aider à synchroniser l'exécution de différents threads et éviter les conditions de concurrence.

Les canaux fonctionnent généralement sur le modèle de la communication point à point, ce qui signifie que les données sont envoyées depuis un point (généralement un thread ou une goroutine) et reçues à un autre point. Cela permet de synchroniser les deux points de la communication : l'émetteur ne continue pas jusqu'à ce que les données aient été reçues, et le récepteur ne continue pas jusqu'à ce qu'il ait reçu des données.

Dans le contexte de Go, les canaux sont un mécanisme de communication de première classe, intégré au langage lui-même. Go supporte des canaux typés qui peuvent être utilisés pour transmettre des données de tout type entre des goroutines.

En Go, les canaux sont utilisés en conjonction avec le mot-clé `go` pour démarrer des goroutines et le mot-clé `chan` pour déclarer des canaux. Les opérations d'envoi et de réception sur les canaux sont réalisées en utilisant les opérateurs `<-` et `->`. Les canaux en Go sont également assortis de mécanismes de synchronisation : une opération d'envoi sur un canal non tamponné bloquera jusqu'à ce qu'une goroutine ailleurs dans le programme exécute une opération de réception correspondante, et vice versa.

Par conséquent, en Go, les canaux sont un moyen élégant et sûr de gérer la synchronisation et la communication entre goroutines, leur permettant de fonctionner ensemble de manière fluide et sans conflits.

## Exemple : somme des éléments d'une tranche (sans goroutines)

```
package main

import "fmt"

func sumSlice(ints []int) int {
    sum := 0
    for _, num := range ints {
        sum += num
    }
    return sum
}
```

```

    }
    return sum
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Printf("La somme des numéros est : %d\n", sumSlice(nums))
}

```

## Exemple : somme des éléments d'une tranche (avec goroutines)

```

package main

import (
    "fmt"
)

func sumSlice(ints []int, results chan<- int) {
    sum := 0
    for _, num := range ints {
        sum += num
    }
    results <- sum
}

func parallelSum(nums []int) int {
    // Création du canal
    results := make(chan int)

    // Divise la slice et lance les goroutines.
    half := len(nums) / 2
    go sumSlice(nums[:half], results)
    go sumSlice(nums[half:], results)

    // Recevoir les résultats des goroutines et calcule la somme
    // totale.
    sum1, sum2 := <-results, <-results
}

```

```
    return sum1 + sum2
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    totalSum := parallelSum(nums)
    fmt.Printf("La somme des numéros est : %d\n", totalSum)
}
```

Dans cet exemple, j'ai créé une nouvelle fonction `parallelSum`. Au début de `parallelSum`, nous créons un canal de type `int` appelé `results`. Ensuite, la slice de nombres est divisée en deux, avec chaque moitié étant passée à une nouvelle goroutine qui exécute la fonction `sumSlice`. Chaque goroutine calcule sa somme partiellement et envoie le résultat sur le canal `results`.

Dans `parallelSum`, nous recevons les résultats des deux goroutines directement dans les variables `sum1` et `sum2` en utilisant l'opération de réception de canal (`<-chan`). Cela bloque la goroutine en cours (qui est la goroutine principale) jusqu'à ce que chaque goroutine ait envoyé son résultat, ce qui remplace effectivement le besoin d'un `WaitGroup`.

Enfin, dans la fonction `main`, nous appelons `parallelSum` directement avec notre slice de nombres, et nous imprimons la valeur renournée qui représente la somme totale de tous les nombres dans la slice.

# Section 4 : WaitGroup

En général, dans le monde de la programmation parallèle, une structure similaire à un "WaitGroup" est souvent utilisée pour synchroniser l'exécution de multiples threads. Le concept principal est d'avoir un moyen de notifier un certain morceau de code (qui pourrait être un autre thread) que plusieurs autres tâches ou threads ont terminé leur exécution.

Un "WaitGroup" est généralement composée d'un compteur qui est incrémenté chaque fois qu'une nouvelle tâche est lancée et décrémenté chaque fois que cette tâche s'achève. Une opération pour attendre (habituellement appelée "wait") va bloquer jusqu'à ce que le compteur atteigne zéro, indiquant que toutes les tâches ont terminé leur exécution.

Dans le contexte de la programmation Go, un `WaitGroup` appartient au package `sync` et offre une manière simple et effective de synchroniser l'exécution de goroutines. C'est un compteur de tâches. Le `WaitGroup` fournit principalement trois méthodes : `Add`, `Done`, et `Wait`.

- `Add` augmente le compteur de tâches du `WaitGroup`. Il est utilisé pour informer le `WaitGroup` qu'il y a une nouvelle tâche à attendre.
- `Done` décrémente le compteur de tâches. Il est généralement mis à la fin de la goroutine pour indiquer que cette goroutine a terminé son exécution.
- `Wait` est utilisé pour bloquer l'exécution jusqu'à ce que le compteur de tâches soit revenu à zéro. Il est généralement utilisé là où vous souhaitez attendre que les tâches se terminent.

Ainsi, dans un programme Go, `WaitGroup` fournit un moyen simple, mais puissant de gérer et attendre l'achèvement de plusieurs goroutines. C'est très utile lorsqu'un nombre dynamique de goroutines est créé ou lorsqu'un morceau de code dépend de l'achèvement d'un ensemble de goroutines.