# Cours

# Introduction

## Qui a fait le Go ?

Go (ou Golang) est un langage de programmation open source assez jeune.

Il a été développé en 2007 par Robert Griesemer, Rob Pike et Ken Thompson qui travaillent aujourd'hui chez Google.

Untitled.png

Le langage Go est officiellement lancé en novembre 2009.

## Pourquoi le Go ?

> "*Chez Google, nous pensons que la programmation devrait être rapide, productive et surtout, fun. C'est pourquoi nous sommes ravis de proposer ce nouveau langage de programmation expérimental. Les opérations de compilation sont presque instantanées,et le code compilé propose une vitesse de fonctionnement proche de celle du C*".

> Première phrase du site golang.org "*Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.*"

Pour résumer : Le langage de programmation Go rime avec efficacité et simplicité.

> Énormément de concepts de l'époque sont redécouvert aujourd'hui

## Ceci est un ...

Untitled 1.png

Geomyidae - Wikipédia

# Héritage très fort des langages des années 70'

Go est syntaxiquement similaire au langage C mais contrairement au C, il possède une sécurité de la mémoire avec un Garbage Collector.

Go est souvent comparé au langage Python car tous les 2 se veulent très simples syntaxiquement.

# Les avantages du langage Go

- Une meilleure protection de la mémoire grâce à son Garbage Collector qui permet une gestion automatique de la mémoire.
- Profite de la puissance de calcul des processeurs les plus robustes du marché (processeurs multi-cœurs).
- Un code maintenable
- Possibilité de faire du typage dynamique et intègre de nombreux types avancés tels que les mappages clé-valeur ( dictionnaires).
- Possède une si riche bibliothèque standard, qu'il est même tout à fait possible de concevoir des programmes écrit avec le langage Go sans aucune dépendance externe.
- Possède un **temps de compilation rapide** et intègre aussi un système de build beaucoup moins compliqué que celui de la plupart des langages de compilés (RIP le C et son makefile de l'enfer !!!).
- Au niveau de la **portabilité** il est possible de compiler son code pour une large gamme de systèmes d'exploitation et de plateformes matérielles (Windows, Linux, MAC OS, Android, IOS).

# Utilisation du langage Go

On retrouve le langage Go dans les domaines suivants (liste non exhaustive) :

- Serveurs
- Web
- Systèmes embarqués
- IOT (Internet Of Things)
- Android
- IOS
- Jeux-vidéos
- etc ...

Des entreprises utilisant Go :

- CloudFlare
- DropBox
- Docker □□
- Nokia
- OVH
- Youtube
- SoundCloud
- Github
- Netflix
- etc...

# Principles features du Go

Pour résumer, le Go c'est :

- Langage statique & compilé (Java, C, C++, ...)
- Syntaxe proche du C
- Garbage Collector ➡Gestion de la mémoire automatique □□
- Multi-CPU et parallelism dans le langage (sans avoir recours à des librairies externes)
- Un seul binaire, aucune dépendance !
- Multi-plateforme (Windows, macOS, Linux, arm, ...)

# Par ici l'installation !

The Go Programming Language

# Pour l'IDE

- VSCode (avec les extensions suggérées)
- Jetbrains Goland
- Atom (avec le plugin https://atom.io/packages/go-plus)
- Vim-go (oui oui...)
- SublimeText (avec le plugin https://github.com/DisposaBoy/GoSublime)

Pour en trouver d'autres :

IDEsAndTextEditorPlugins · golang/go Wiki

# Les bases

# Clean architecture Go

Untitled.png

# Le playground

https://play.golang.org/

# Bonjour monde

```
package main

import "fmt"

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

# Les types

Langage fortement typé, avec possibilités de faire de l'inférence.

On peut déclarer une variable avec un type de plusieurs façons.

# Les types basiques

```
bool            // true / false

string          // "Hello", "Goodbye"
```

```
int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr
□□□ // 10, 20, 42255

byte            // 01001110 ==> stockage sur un octet (alias de uint8)

rune            // alias de int32

float32 float64  // 3.14 ==> équivalent à float et double
```

# Déclaration d'une variable

```
var number int = 20
var age int         // = 0

var firstname string = "Nou"
var lastname string  // = ""
```

# Déclaration par inférence

```
number := 10

firstname := "Nono"
```

> On peut aussi utiliser les décalages de bits ! ( >> , <<).
> Non ça n'est pas un smiley

# Cast

```
var n int = 42
f := float64(n) + .42
fmt.Printf("float=%f\n", f)
```

# Tests & Boucles

## Les combinaisons et opérateurs booléens

### Comparaisons

- `==`
- `!=`
- `<`
- `>`
- `>=`
- `>=`

### Opérateurs booléens

- `&&`
- `||`
- `!`

## If

```
age := 10
if age > 10 {
	// something
}
```

### Conditions alternatives

```
age := 10
if age > 10 {
	// something
} else if a > 5 {
    // something else
} else {
    // something else else
}
```

# Switch

> Hé Il n'y a pas de break dans les switch en go

```
age := 10
switch age {
case 10:
	// code cas 10
case 5 :
	// code cas 5
default :
	// code par défaut (facultatif)
}
```

On peut cumuler les valeurs au sein d'un même `case` !

```
age := 10
switch age {
	case 10,8,6:
		// code cas 10 ou 8 ou 6
	case 5 :
		// code cas 5
	default :
		// code par défaut (facultatif)
}
```

Et même mieux encore...

```
hour := 10
fmt.Printf("Il est %dh\n", hour)
switch {
	case hour < 12:
		fmt.Println("C'est le matin !")
	case hour > 12 && hour < 18:
		fmt.Println("C'est l'après midi")
	default:
		fmt.Println("On est le soir")
}
```

# While

> Attention !
> Le `while` n'existe pas en Go ! Haha !

# For

Le `for` en Go est très évolué et permet de remplacer l'utilisation du `while`.

```
sum := 0
for i := 0; i < 10; i++ {
	sum += i
}
fmt.Println(sum)
```

# Remplacement du while

```
n := 1
for n < 5 {
    n *= 2
}
fmt.Println(n)
```

>

> Les instructions `continue` et `break` sont aussi utilisables en Go

# Boucle infinie

```
for {
    fmt.Println("Boucle infinie")
}
```

# Tableaux

# Tableaux à taille fixe

## Définition

> Simplement Un tableau à taille fixe est une séquence d'éléments d'une taille définie

- Tout est alloué d'un seul bloc ➡ les cases sont contiguës en mémoire
- Le premier index démarre à 0.
- La taille est définitive, pour agrandir, il faut allouer un autre tableau ou utiliser une autre technique.
- Le contenu sera toujours initialisé à `0, "", ...`.

## Syntaxe

```
var nom[taille]type
```

```
var tab[5]int
t[3] = 12
```

### Déclaration rapide

```
odds := [4]int{1, 3, 5, 7}
pair := [4]int{2, 4} // [2, 4, 0, 0]
```

## Affichage

```
var names [3]string
names[0] = "Bob"
names[2] = "Alice"
```

```
fmt.Printf("name[2]=%v\n", names[2])
fmt.Printf("names=%v (len=%d)\n", names, len(names))
```

# Tableaux dynamiques (Slice)

## Définition

**Tableau de taille dynamique**

- Slice ⇒ Tranche 🍰
- Un Slice représente une tranche d'un tableau.
- Un Slice est une "vue" sur le tableau sous-jacent
- Modifier le slice ➡ modifier le tableau

## Syntaxe

```
s := make([]type, taille, capacité)
```

- **Taille** : nombre d'éléments du slice
- **Capacité** (facultatif) : nombre d'éléments du tableau

```
s := make([]int, 3)
s[0] = -3
len(s) // 3
cap(s) // 3
```

## Réallocation

```
s := make([]int, 3)
s = append(s, 12)
len(s) // 4
cap(s) // 6
```

Explication :

- Si on dépasse la taille du tableau
- Un nouveau tableau est alloué, de capacité doublée

# Sous-tableaux

```
letters := []string{"g", "o", "l", "a", "n", "g"}
fmt.Printf("%v \n", letters)

// subslicing
sub1 := letters[:2]
sub2 := letters[2:]
fmt.Printf("%v\n", sub1) // ?
fmt.Printf("%v\n", sub2) // ?
```

## Référence des sous tableaux

Que se passe-t-il si on fait ça ?

```
letters := []string{"g", "o", "l", "a", "n", "g"}
sub1 := letters[:2]
sub2 := letters[2:]

sub2[0] = "UP"
fmt.Printf("%v\n", sub2) // ?
fmt.Printf("%v\n", letters) // ?
```

Et là ?

```
letters := []string{"g", "o", "l", "a", "n", "g"}
sub2 := letters[2:]

subCopy := make([]string, len(sub2))
copy(subCopy, sub1)
subCopy [0] = "UP"
fmt.Printf("%v\n", subCopy) // ?
fmt.Printf("%v\n", letters) // ?
```

# Les fonctions

```
func printInfoNoParam() {
	fmt.Printf("Name=%s, age=%d, email=%s\n", "Bob", 10, "bob@golang.org")
}

func printInfoParams(name string, age int, email string) {
	fmt.Printf("Name=%s, age=%d, email=%s\n", name, age, email)
}

func avg(x, y float64) float64 {
	return (x + y) / 2
}

func sumNamedReturn(x, y, z int) (sum int) {
	sum = x + y + z
	return // c pas bo hein ...
}

func main() {
	printInfoNoParam()
	printInfoParams("Noé", 15, "noe@flex.org")

	result := avg(16.3, 25.0)
	fmt.Printf("Average result=%f\n", result)

	sum := sumNamedReturn(10, 25, 7)
	fmt.Printf("Sum result=%d\n", sum)
}
```

# Multiples return

```
func ToLowerStr(name string) (string, int) {
	return strings.ToLower(name), len(name)
}
```

```
func main() {
	lowerName, len := ToLowerStr("NOE") // on a le droit mais c'est pas beau non plus
	fmt.Printf("%s (len=%d)\n", lowerName, len)

	_, len = ToLowerStr("Paul ABIB, oui le seul le vrai l'unique")
	fmt.Printf("bob len=%d\n", len)
}
```

# Range

> C'est la continuité du `for`, il permet d'itérer sur une collection de donnée

# Syntaxe

```
for <index>, <value> := <dataset> {
	//code
}
```

# Exemple

```
names := []string{"Bob", "Alice", "Bobette", "John"}
	for i, n := range names {
		fmt.Printf("Username=%s (index=%d)\n", n, i)
	}

	// range on string
	// Omit index / value
	for _, c := range "golang" {
		fmt.Printf("%v\n", string(c))
	}
```

# Gestion d'erreurs

## Gestion d'erreurs dans les langages

Il y a plusieurs stratégies possibles :

- Code d'Erreurs
- Exceptions
- Pattern Matching
- ...

## Go et le retour multiple

En Go, nous allons exploiter le retour multiple des fonctions pour gérer nos erreurs

## Exemple classique

```
func MyFunc() (int, error) {
	// code
	return 1
}

func main() {
	v, err := MyFunc()

	if err != nil {
		fmt.Printf("Error in MyFunc: %v", err)
	}
}
```

> nil = NULL

# Gestion d'erreur standard en Go

En Go, on peut retrouver souvent des codes qui auront cette forme-là pour gérer les erreurs

```
v1, err := MyFunc1()
if err != nil {
	return err
}

v2, err := MyFunc2()
if err != nil {
return err
}

v3, err := MyFunc3()
if err != nil {
return err
}

v4, err := MyFunc4()
if err != nil {
return err
}
```

C'est un peu répétitif... Mais efficace ! Cela apporte une lecture du code progressivement.

# Early return

En Go, on va favoriser les tests et retour d'erreurs en tout début de fonction, pour faire un retour d'erreur le plus rapidement possible, permettre à notre code qui suit de grandir plus facilement.

## Code non-early return

```
func MyFunc(condition bool) (int, err) {
	if (condition) {
		if (!condition2) {
			return 0, errors.New("Error 2!")
		}
		// code
```

```
    return 42, nil
  }
  return 0, errors.New("Error!")
}
```

## Code early-return

```
func MyFunc(condition bool) (int, err) {
  if (!condition) {
    return 0, errors.New("Error!")
  }
  if (!condition2) {
    return 0, errors.New("Error 2!")
  }
  // code
  return 42, nil
}
```

# Fichiers

Pour manipuler un fichier en Go, il existe plusieurs librairies permettant différentes actions.

# io/ioutil

C'est sans doute l'approche la plus simple pour manipuler un fichier.

Elle permet de directement lire un répertoire ou le contenu d'un fichier, et même d'écrire dedans.

## Lecture fichier

```
func readFile(filename string) (error) {
	dat, err:= ioutil.ReadFile(filename)
	if err != nil {
		return "", err
	}

	if len(dat) == 0 {
		// return "", errors.New("Empty content")
		return "", fmt.Errorf("Empty content (filename=%v)", filename)
	}
	
	fmt.Printf("%s\n", dat)

	return nil
}
```

## Écriture fichier

```
func writeFile(filename, content string) error {
	err:= ioutil.WriteFile(filename, []byte(content), 0644)
	if err != nil {
		return err
```

```
	}
	return nil
}
```

# Lecture répertoire

```
func readDir() error {
	files, err := ioutil.ReadDir(".")
	if err != nil {
		return err
	}

	for _, file := range files {
		fmt.Println(file.Name())
	}
	return nil
}
```

# Inconvénient

On fait de la lecture / écriture direct, aucun buffer.

Peut poser problème en cas de traitement de gros fichiers....

On peut potentiellement écraser un fichier existant

# os + bufio

`Bufio` implémente des manipulations de flux avec des buffers, un peu plus verbeux, mais beaucoup plus intéressant !

# Lecture

```
func readFile(filename string) error {
	srcFile, errSrc := os.Open(src)
	if errSrc != nil {
```

```
    return errSrc
  }

  lineIdx := 1
  scanner := bufio.NewScanner(srcFile)

  for ; scanner.Scan(); lineIdx++ {
    fmt.Println("Line", lineIdx, ":", scanner.Text())
  }

  srcFile.Close()
  return nil
}
```

# Écriture

```
func writeFile(filename string, lines []string) error {
  dstFile, errDst := os.Create(dst)
  if errDst != nil {
    return errDst
  }

  writer := bufio.NewWriter(dstFile)

  for _, line range lines {
    _, errWrt := fmt.Fprintln(writer, line)
    if errWrt != nil {
      return errWrt
    }
  }

  writer.Flush()
  dstFile.Close()
  return nil
}
```

# Defer

# Repousser l'exécution d'une instruction

## Cas d'utilisation

Dans l'exemple si dessous

```
func main() {
	f := os.OpenFile("foo.txt")
	if condition1 {
		return // Oops...! pas de close ici!
	}
	// code
	f.Close()
}
```

Le `f.close()` peut être très éloigné dans le code du `OpenFile`.
Or, quand on ouvre un fichier, on va surement le fermer ensuite, mais pas tout de suite...

On peut même potentiellement arrêter notre fonction avant de fermer le fichier sans faire attention !

Il paraît donc plus "jolie" de mettre le code d'ouverture du fichier et celui de fermeture côte à côte, car il traite le même sujet.

**Solution**

```
func main() {
	f := os.OpenFile("foo.txt")
	defer f.Close() // exécuté quand on sort de main()

	if condition1 {
		return
```

```
}
}
```

> `defer` est rattaché à la fonction qui l'invoque

# Ordre d'exécution

Les instructions mises en `defer` fonctionnerons comme une pile **LIFO** (Last In First Out).

# Kata Find and Replace

# Énoncé

Programme qui trouve et remplace un mot par un autre dans un fichier.

## Exemple

*Remplacer le mot **Go** par **Python***

| Source: wikigo.txt | Résultat |
| --- | --- |
| Go was conceived in 2007 to improve programming productivity at Google | Python was conceived in 2007 to improve programming productivity at Pythonogle |

# Features

- Affiche le résumé du traitement en console
  - Nombre d'occurrences du mot
  - Numéros de lignes modifiés
- Écrire le texte modifié dans un nouveau fichier pour préserver l'original
- Bonus : prendre aussi en compte que le mot peut avoir une majuscule (ou pas !)

## Exemple de résumé

```
$ go run main.go
== Summary ==
Number of occurrences of Go: 10
Number of lines: 7
Lines: [ 1 - 8 - 15 - 17 - 19 - 23 - 28 ]
== End of Summary ==
```

# Prototypes

```
func ProcessLine(line, oldWord, newWord string) (found bool, result string, occurrences int)
```

- `line` : ligne à traiter
- `oldWord` / `newWord` : ancien mot / nouveau mot
- `found` : vrai si au moins une occurrence trouvée
- `result` : résultat après traitement
- `occurrences` : nombre d'occurrences de `oldWord` dans la ligne

```
func FindReplaceFile(src string, dst string, oldWord string, newWord string) (occurrences int, lines []int, err error)
```

- `src` : nom du fichier source
- `oldWord` / `newWord` : ancien mot / nouveau mot
- `occurrences` : nombre d'occurrences de `oldWord`
- `lines` : tableau des numéros de lignes ou `oldWord` a été trouvé
- `err` : erreur générée par la fonction

> **Astuce**
> `FindReplaceFile` n'arrive pas à ouvrir le fichier, il faut renvoyer une erreur

# Outils

## Bufio

```
scanner := bufio.NewScanner(srcFile)
for scanner.Scan() {
t := scanner.Text()
}
```

```
writer := bufio.NewWriter(dstFile)
defer writer.Flush()
fmt.Fprintln(writer, "Texte d'une ligne")
```

## Strings

```
c := strings.Contains("go ruby java", "go") // c == true
cnt := strings.Count("go go go", "go") // cnt == 3
res := strings.Replace("old go", "go", "python", -1) // res == "old python"
```

# Solution

```
package main

import (
	"bufio"
	"fmt"
	"os"
	"strings"
)

func ProcessLine(line, oldWord, newWord string) (found bool, result string, occurrences int) {
	oldWordLower := strings.ToLower(oldWord)
	newWordLower := strings.ToLower(newWord)
	result = line
	if strings.Contains(line, oldWord) || strings.Contains(line, oldWordLower) {
		found = true
		occurrences += strings.Count(line, oldWord)
		occurrences += strings.Count(line, oldWordLower)
		result = strings.ReplaceAll(line, oldWord, newWord)
		result = strings.ReplaceAll(result, oldWordLower, newWordLower)
	}

	return found, result, occurrences
}

func FindReplaceFile(src string, dst string, oldWord string, newWord string) (occurrences int, lines []int, err error)
{
	// open src file
	srcFile, errSrc := os.Open(src)
	if errSrc != nil {
		return 0, lines, errSrc
	}
	defer srcFile.Close()

	// open dst file
	dstFile, errDst := os.Create(dst)
	if errDst != nil {
		return 0, nil, errDst
```

```
    }
    defer dstFile.Close()

    oldWord = oldWord
    newWord = newWord
    lineIdx := 1
    scanner := bufio.NewScanner(srcFile)
    writer := bufio.NewWriter(dstFile)
    defer writer.Flush()

    for scanner.Scan() {
        found, res, o := ProcessLine(scanner.Text(), oldWord, newWord)
        if found {
            occurrences += o
            lines = append(lines, lineIdx)
        }

        _, errWrt := fmt.Fprintln(writer, res)
        if errWrt != nil {
            return occurrences, lines, errWrt
        }

        lineIdx++
    }

    return occurrences, lines, nil
}

func main() {
    oldWord := "Go"
    newWord := "Python"
    occurrences, lines, err := FindReplaceFile("wikigo.txt", "wikipython.txt", oldWord, newWord)
    if err != nil {
        fmt.Printf("Error while executing find replace: %v\n", err)
        return
    }

    fmt.Println("== Summary ==")
    defer fmt.Println("== End of Summary ==")
    fmt.Printf("Number of occurrences of %s: %d\n", oldWord, occurrences)
```

```
	fmt.Printf("Number of lines: %d\nLines: [ ", len(lines))
	linesCount := len(lines)
	for i, l := range lines {
		fmt.Printf("%d", l)
		if i < linesCount-1 {
			fmt.Printf(" - ")
		}
	}
	fmt.Println(" ]")

}
```

# Structures & Pointeurs

## Définition

> Simplement
> Type personnalisé représentant une collection de champs

## Syntaxe

```
type <NomStruct> struct {
	var1 int
	var2 string
	var3 float64
}
```

## Exemple

```
type User struct {
	Name string
	Email string
	Age int
}
```

## Déclaration

Il y a 3 types de déclaration possible

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

```
func main() {
  // 1
	var p1 Person
	p1.Name = "Bob"
	p1.Addr.city = "Lyon"
	
	// 2
	p2 := Person{"Paul","Abibi"}

	// 3
	p3 := Person{Name: "Swann"}
}
```

# Règles

- Une structure ne peut contenir que des variables
- La règle de visibilité de package s'applique pour :
  - la structure elle-même
  - les variables de la structure

## Exercice player

- Définir 2 structure :
  - Avatar
    - Url
  - Player
    - Name
    - Age
    - Avatar
    - password
- Le mot de passe doit avoir un scope privé.
- Créer une fonction de création d’un player, qui ne prend que son nom en argument et initialise la structure avec ce dernier, ainsi qu’un mot de passe par défaut.

---

### Solution

```
type Avatar struct {
	Url string
}
```

```
type Player struct {
	Name     string
	Age      int
	Avatar   Avatar
	password string
}

func New(name string) Player {
	return Player{
		Name:     name,
		password: "defaultpassword",
	}
}
```

# Embedded struct

```
type Avatar struct {
	Url string
}

type Player struct {
	Name     string
	Avatar   Avatar
}
```

Un Player a un Avatar

---

Parfois, on veut exprimer un autre type de relation → Un XXX est un YYY

Dans d’autres langages, cette relation est exprimée par l’héritage

Go préfère la composition avec l’embedded struct :

```
type Avatar struct {
	Url string
}

type Player struct {
	Name     string
```

```
Avatar // Pas de nom de variables
}

var p Player
p.Url = "https://photodemoi.jpg"
```

Dans ce code, Avatar est embarqué dans le type Player

Player est un Avatar

# Receiver function

Grâce à ce fonctionnent, on peut enfin reproduire quasiment à l'identique le comportement d'un objet tel qu'en Java par exemple.

```
type User struct {
Name string
}
func (u User) SayHello() {
fmt.Printf("Hello %v!\n", u.Name)
}
```

- **u** est un value receiver pour la méthode SayHello()
- Les champs de **u** sont accessibles pour la fonction
- C'est bien beau ça, mais à quoi ça nous sert ?

```
func main() {
    u := User{"Paul"}
    u.SayHello()
}
```

Lorsqu'on utilise cette technique, notre struct User passé en argument est en réalité "copiée" pour la méthode.

Conséquence : Une méthode avec un value receiver ne peut pas modifier la structure originale. Cela peut permettre de favoriser l'immutabilité en renvoyant une nouvelle instance de notre structure avec les propriétés mise à jour !

## Exercice rectangle

- Définir une structure rectangle qui contient
  - Longueur
  - Largeur
- Créer 3 receiver functions pour cette structure :
  - `Area()` : renvoie l'air du rectangle
  - `String()` : Affiche les informations de la structure bien formatées
  - `DoubleSize()` : Renvoie une nouvelle structure du rectangle avec sa taille doublée

> Astuce
> Définir la receiver function `String()` d'une structure viendra surcharger l'affichage par défaut de cette dernière !

---

## Solution

```
package main

import (
	"fmt"
)

type Rect struct {
	Width, Height int
}

func (r Rect) Area() int {
	return r.Width * r.Height
}

func (r Rect) String() string {
	return fmt.Sprintf("Rect ==> width=%v, height=%v", r.Width, r.Height)
}

func (r Rect) DoubleSize() Rect {
	r.Width *= 2
	r.Height *= 2
	return r
}

func main() {
```

```
	r := Rect{2, 4}
	fmt.Printf("Rect area=%v\n", r.Area())
	fmt.Println(r)

	r2 := r.DoubleSize()
	fmt.Println("r", r)
	fmt.Println("r2", r2)
}
```

# Pointeurs

En Go, lorsque qu'on passe un paramètre à une fonction, on passe en réalité une copie de cette dernière,

Les pointeurs en Go fonctionnent presque comme en C, à l'exception que nous n'avons pas à gérer l'allocation et la libération de la mémoire.

```
x := -42
s := "Bob"
p := &x // Création d'un pointer vers la variable x
i := *p // Déréférencement de p pour récupérer la valeur de x
```

# Manipulations

```
func UpdateVal(val string) {
	val = "value"
}

func UpdatePtr(ptr *string) {
	*ptr = "pointer"
}

func main() {
	i := 1
	var p *int = &i

	fmt.Printf("i=%v\n", i)
```

```
	fmt.Printf("p=%v\n", p)
	fmt.Printf("*p=%v\n", *p)
	fmt.Println("----------")

	s := "Paul"
	sPtr := &s
	s2 := *sPtr
	fmt.Println("String pointer")
	fmt.Printf("*s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("----------")

	*sPtr = "Clément"
	fmt.Println("Dereference and update")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Printf("s2=%v\n", s2)
	fmt.Println("----------")

	UpdateVal(s)
	fmt.Println("Func UpdateVal()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("----------")

	UpdatePtr(&s)
	UpdatePtr(sPtr)
	fmt.Println("Func UpdatePtr()")
	fmt.Printf("s=%v\n", s)
	fmt.Printf("*sPtr=%v\n", *sPtr)
	fmt.Println("----------")
}
```

# Pointer Receiver

Comme dit plus tôt, les paramètres de fonctions sont des copies des objets originaux.

Ça vaut aussi pour les fonctions `value receiver` sur les structures. Elles ne peuvent donc que faire de la lecture simple, et ne peuvent pas modifier la structure d'origine.

Grâce aux pointeurs, on peut régler ce problème et le couplant à des `receiver functions`.

```
type Post struct {
	Title     string
	Text      string
	published bool
}

func (p Post) Headline() string {
	return fmt.Sprintf("%v - %v", p.Title, p.Text[:50])
}

func (p *Post) Publish() {
	p.published = true
}

func (p *Post) Unpublish() {
	p.published = true
}

func main() {
	p := Post{
		Title: "Go release",
		Text: "Go is a programming language...",
	}

	fmt.Println(p.Headline())

	fmt.Printf("Post published? %v\n", p.Published())
	p.Publish()
	fmt.Printf("Post published? %v\n", p.Published())
}
```

Enfin, si on souhaite créer une structure directement sous la forme d'un pointeur, on peut faire autrement que :

```
p := Post{
	Title: "Go release",
```

```
□□Text: "Go is a programming language...",
□}
pointer := &p
```

Comme ceci :

```
pythonPost := &Post{
□□Title: "Python Intro",
□□Text: "Python is an interpreted high-level programming language",
□}
```

`pythonPost` est un pointeur.

# Maps

# Définition

Structure associant des clés à des valeurs

On peut mettre en clé tout ce qui est comparable (on peut mettre une structure comme clé)

# Syntaxe

La syntaxe “longue” de déclaration d’une map est la suivante :

```
var m map[KeyType]ValueType
---------------------------
var m map[string]int = make(map[string]int)
```

Grâce à l’inférence des types à la déclaration, on peut encore simplifier cette syntaxe en :

```
m2 := make(map[string]int)
```

# Opérations

Pour ces exemples, nous avons une map qui a pour clé une chaine de caractère et pour valeur un entier.

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

On peut récupérer la taille de map en utilisant une fonction que nous connaissons depuis les slices `len()`

```
fmt.Printf("Map size %v\n", len(myMap))
```

# Assignation

L’assignation et très simple, trèèèèès simple !

```
myMap["hello"] = 5
myMap["goodbye"] = 10
```

# Récupération

Pour récupérer la valeur, comme pour l'assignation, il suffit de faire comme avec un tableau

```
fmt.Printf("key=hello, value=%v\n", myMap["hello"])
```

# Présence d’une clé

Pour tester la présence, on utilise le retour multiple caché dans la récupération d’une valeur

```
j, isPresent := myMap["helo"]
fmt.Printf("j=%v, isPresent =%v\n", j, isPresent )
```

`isPresent` est un type booléen et est égale à `false` si la clé n’existe pas.

Dans le cas où la clé n’existe pas, la valeur sera celle par défaut du type (0 pour un entier, chaine vide pour une chaine de caractères, ...)

Si on souhaite mettre ce test dans une condition, on peut faire de cette manière :

```
if _, present = myMap["hel"]; present {
	// ... code
}
```

# Supprimer une clé / valeur

On utilise la fonction `delete`

```
delete(myMap, "hello")
```

# Assignation rapide et parcours

On peut assigner des valeurs dans notre map dès la déclaration comme avec les tableaux

```
myMap := map[string]int{
"Noé": 10,
"Paul": 15,
"Swann": 18,
"Nathanael" : 0
}
```

Pour parcourir une map, on peut utiliser `range`

```
for name, idk := range myMap{
fmt.Printf("name=%v, idk=%v\n", name, idk)
}

// Only keys
for name := range m {
fmt.Printf("name=%v\n", name)
}
```

# Map & Struct

Pour illustrer la mise en place d'une map constitué de structures, on va utiliser les structures suivantes

```
type User struct {
name string
}

type Key struct {
ID   int
Name string
}
```

On peut créer une map de cette manière :

```
myMap := make(map[Key]User)

myMap[Key{1,"ceo"}] = User{"Swann"}
```

```
fmt.Printf("%v"), myMap[Key{1,"ceo"}])
```

> Lorsqu’on récupère une structure depuis une map, on récupère en réalité une copie de cette dernière. Pour palier à ça on peut transformer notre map en une map de pointeurs

```
myMap := make(map[Key]*User)

myMap[Key{1,"ceo"}] = &User{"Swann"}

fmt.Printf("%v"), *myMap[Key{1,"ceo"}])
```