Le langage Go

Éric Jacoboni

10 décembre 2020

jacoboni@univ-tlse2.fr

Sommaire

Ressources

Ressources

- Le site officiel du langage (en anglais)
- Sa page Wikipédia
- Une liste de livres (en anglais)
- Documentations diverses en anglais (articles, vidéos, blogs, tutoriels)
- Chaîne Youtube consacrée à Go
- Golang crash course (vidéo en anglais)
- Canal Reddit consacré à Go
- Mon livre de référence (en anglais)

Recherches sur le Web

N'utilisez pas le mot-clé **go**, *mais* **golang**. *Par exemple,* **"livres golang"**.

Un premier programme : hello.go

```
package main

import "fmt"

func main() {
   fmt.Println("Bonjour à tous !")
}
```

- La fonction *main()* est le point d'entrée du programme, c'est elle qui sera appelée en premier lors de son exécution.
- Le *corps* de la fonction (le code qu'elle exécute) est délimité par des accolades. L'accolade ouvrante *doit* être sur la même ligne que la déclaration de la fonction, introduite par le mot-clé *func*.
- Ici, la fonction *main()* ne prend aucun paramètre et ne renvoie rien.
- Le fichier contenant la fonction *main()* doit faire partir du paquetage *main*.
- La fonction *Println()* (notez la majuscule) fait partie du paquetage *fmt*, qu'il faut donc *importer*.

Un outil essentiel : le playground

- Le playground est une page web permettant de tester et de partager du code Go.
- Lorsque vous commencez, vous avez à votre disposition un programme de type "Hello Word" que vous pouvez modifier à votre guise.
- Le bouton *Format* permet de formater votre code tandis que le bouton *Import*, s'il est coché, ajoutera automatiquement l'importation des paquetages que vous utilisez (lorsque vous cliquerez sur *Format*).
- Est-il nécessaire d'expliquer ce que fait le bouton *Run* ?
- Pour montrer votre code, donner une solution, demander de l'aide, il suffit de cliquer sur le bouton *Share* afin d'obtenir une URL que vous pourrez envoyer à qui vous le souhaitez.
- Celui qui dispose de votre lien de partage peut modifier votre code et le repartager (une nouvelle URL est recréée : votre ancien code reste donc disponible).

GOPATH

- Go recherche vos fichiers sources relativement à un répertoire particulier appelé *GOPATH*.
- Pour connaître sa valeur pour votre machine, faites la commande *go env GOPATH*
- Dans ce répertoire doivent se trouver 3 sous-répertoires :
  - *bin* : c'est là que les utilitaires Go seront installés (notamment par le plugin Go de VS Code)
  - *pkg* : c'est là que seront installées les versions compilées des paquetages que vous installerez par la suite.
  - *src* : c'est là que vous devrez stocker vos fichiers sources, à raison *d'un répertoire par projet*.

  Exemple :

```
$ go env GOPATH
/Users/jaco/go
$ cd go
$ ls
bin pkg src
```

Commandes go

La commande *go* dispose de plusieurs sous-commandes. Leur liste est disponible en tapant simplement *go*. Nous utiliserons principalement les sous-commandes suivantes :

- *build* : compilation des paquetages et de leurs dépendances
- *fmt* : formate correctement un fichier Go
- *run* : compile et exécute un programme Go (le programme compilé n'est pas conservé sur le disque)

Si le programme précédent est stocké dans le fichier *hello.go*, lui même placé dans un répertoire *hello* sous *GOPATH/src*. On suppose qu'on est dans ce répertoire :

- *go* ***build*** *hello.go* produira un fichier exécutable nommé *hello* sous Unix et *hello.exe* sous Windows, sans l'exécuter.
- *go* ***run*** *hello.go* exécutera le programme après l'avoir compilé, mais sans conserver le fichier compilé.
- *go* ***fmt*** *hello.go* formate correctement le fichier *hello.go* (cette opération est généralement automatique lorsqu'on utilise un IDE).

Syntaxe du langage

Principales différences avec Python

Par rapport à Python (que vous êtes censés connaître), les différences principales sont :

- Go est un langage **compilé**, alors que Python est **interprété**.
- Go est un langage à **typage statique**, alors que Python utilise un **typage dynamique**
- Go permet de définir des constantes, contrairement à Python
- Chaque variable Go doit être déclarée avant son utilisation, soit par le mot-clé *var* soit par une initialisation particulière (voir plus loin). Chaque constante doit être déclarée par le mot-clé *const*.
- Le programme principal doit être placé dans une fonction qui doit s'appeler *main* (voir exemple précédent)
- Les blocs sont définis entre accolades. Contrairement à Python, l'indentation ne sert qu'à la lisibilité.

Principales différences avec Python

- Les fonctions sont introduites par le mot-clé *func* (et non *def*)
- Comme pour les variables, les types des paramètres des fonctions et de la valeur de retour doivent être typés.
- Il n'y a qu'une seule instruction de boucle, *for*, qui permet de réaliser toutes les boucles classiques que l'on connaît.

Déclarations de constantes et de variables

- Go est un langage à *typage statique*, ce qui signifie que les variables, les constantes et les fonctions doivent être associées à un **type** *au moment de la compilation* (c'est donc différent de Python qui, lui, utilise un *typage dynamique* – associé au moment de l'exécution).
- Cette association se fait via une *déclaration* :
  - Une *déclaration de constante* est de la forme ***const*** *nom [type] = valeur*. Si on ne précise pas le type, la constante prend le « type par défaut de la valeur » (voir cet article)
  - Une *déclaration de variable* est de la forme ***var*** *nom type [= valeur]*. Si aucune valeur n'est fournie, la variable est initialisée avec la *valeur zéro du type*. Si une valeur est fournie, on peut omettre le type : ***var*** *nom = valeur* (comme pour les constantes).
  - La forme courte *nom* ***:=*** *valeur* est équivalente à la forme précédente et est donc très souvent utilisée.
  - On peut déclarer plusieurs constantes ou plusieurs variables simultanément (voir exemple plus loin).
- Toute constante ou variable déclarée doit être utilisée !

Types de base : entiers, flottants, booléens et chaînes

- Il y a 4 tailles d'entiers (8, 16, 32 et 64 bits) signés et non signés. Ils sont représentés par les types *int8*, *int16*, *int32*, *int64* et leurs équivalents non signés *uint8*, *uint16*, *uint32* et *uint64*. Il existe également les types *int* et *uint* qui utilisent la taille la plus adaptée à la plateforme. Le type *int* est, de loin, le plus utilisé.
- Le type *rune*, qui sert à représenter un code Unicode de caractère est l'équivalent du type *int32*. Le type *byte*, qui sert à représenter un octet quelconque, est l'équivalent de *uint8*.
- Il y a deux types flottants, *float32* et *float64* (le type par défaut des flottants).
- Les booléens sont représentés par le type *bool* et ne peuvent prendre que deux valeurs, ***true*** et ***false***.
- Les chaînes sont représentées par le type *string* et contiennent des caractères Unicode au format UTF-8. Les littéraux chaînes sont placés entre *apostrophes doubles*.

Opérateurs

En Go, les opérateurs ne s'appliquent qu'à des opérandes *de même type* :

- Opérateurs sur les **nombres** : *+*, *-*, ***, */* et *%*. Si l'une des opérandes de */* est un flottant, le résultat est la division réelle. Si les deux opérandes de */* sont des entiers, le résultat est le quotient de la division entière. L'opérateur *%* ne s'applique qu'aux entiers et renvoie le reste de la division entière.
- Opérateurs sur les **booléens** : *!*, *&&*, *||* représentent les opérateurs *non*, *et* et *ou*. Les opérateurs *&&* et *||* fonctionnent en court-circuit.
- Opérateur sur les **chaînes** : *+* s'applique à deux chaînes et renvoie une chaîne résultant de leur concaténation.

Conversions entre nombres et opérateurs de comparaison

- Pour tout type numérique *T*, l'opération *T(val)* convertit la valeur numérique *val* dans le type *T*. Dans certains cas (flottant vers entier, ou *int32* vers *int8* par exemple), cette conversion peut provoquer une perte d'information ou un débordement.
- Opérateurs de comparaison : $==$, $!=$, $<$, $>$, $<=$, $>=$. Ils peuvent s'appliquer aux nombres et aux chaînes et leur résultat est un booléen.
- Un littéral entier (42 par exemple) est compatible avec tous les types entiers (idem avec les littéraux flottants et les deux types flottants).

Exemples de conversions entre nombres

```
package main

import "fmt"

func main() {
  var (                    // Regroupement de déclarations de variables
    i   int   = 42
    i16 int16 = 12

    f32 float32 = 3.14
    f64 float64 = .1234

    mess1 string = "Bonjour"
    mess2 string = "à tous"
  )

  /* Code incorrect :
  fmt.Println(i + i16)   // invalid operation: i + i16 (mismatched types int and int16)
  fmt.Println(i + f32)   // invalid operation: i + f32 (mismatched types int and float32)
  fmt.Println(f32 + f64) // invalid operation: f32 + f64 (mismatched types ...)
  */
```

Exemples de conversions de nombres

```
  // Code correct :
  fmt.Println(i + int(i16))       // Ok : 54
  fmt.Println(int16(i) + i16)     // Ok : 54
  fmt.Println(i + int(f32))       // Ok, mais troncature de f32 : 45
  fmt.Println(float32(i) + f32)   // Ok : 45.14
  fmt.Println(float64(f32) + f64) // Ok : 3.263400104904175
  fmt.Println(f32 + float32(f64)) // Ok, mais perte de précision de f64 : 3.2634

  fmt.Println(i + 12)     // Ok car 12 est un littéral entier : 54
  fmt.Println(f32 + 3.14) // Ok car 3.14 est un littéral flottant : 6.28
  fmt.Println(f64 + 3.14) // Ok : 3.2634000000000003

  fmt.Println(5 / 2)   // Quotient : 2
  fmt.Println(5 % 2)   // Reste : 1
  fmt.Println(5 / 2.0) // Division réelle : 2.5

  fmt.Println(mess1 + " " + mess2) // Bonjour à tous
  fmt.Println(mess1 < "Coucou")    // true
}
```

Exemples avec déclarations courtes

```
package main
import "fmt"

func main() {
  i := 42
  i16 := 12

  f32 := 3.14
  f64 := .1234

  mess1 := "Bonjour"
  mess2 := "à tous"

  fmt.Printf("Type de i = %T, de i16 = %T\n", i, i16)             // int
  fmt.Printf("Type de f32 = %T, de f64 = %T\n", f32, f64)         // float64
  fmt.Printf("Type de mess1 = %T, de mess2 = %T\n", mess1, mess2) // string

  fmt.Println(i + i16)   // Ok : 54
  fmt.Println(f32 + f64) // Ok : 3.2634000000000003

  fmt.Println(i + 12)     // Ok car 12 est un littéral entier : 54
  fmt.Println(f32 + 3.14) // Ok car 3.14 est un littéral flottant : 6.28
  fmt.Println(f64 + 3.14) // Ok : 3.2634000000000003
```

Conversions entre nombres et chaînes

- Pour convertir un entier en chaîne et réciproquement, on dispose des fonctions *Itoa()* et *Atoi()* du module strconv et de la fonction *fmt.Sprintf()* :

```
i := 42                              // i est un int
chaine_i := strconv.Itoa(i)          // chaine_i est un string et vaut "42"
chaine_i2 := fmt.Sprintf("%d", i)    // chaine_i2 est un string et vaut "42"
val, err := strconv.Atoi("-42")      // val vaut -42, err vaut nil
```

- Pour convertir un flottant en chaîne et réciproquement, on dispose des fonctions *fmt.Sprintf()* et *strconv.ParseFloat()* :

```
f := 3.14
chaine_f := fmt.Sprintf("%f", f)   // chaine_f vaut "3.14"
// Dans les deux cas, err vaudra nil
val64, err := strconv.ParseFloat("3.14", 64)  // float64
val32, err := strconv.ParseFloat("3.14", 32)  // float32
```

- *strconv* dipose de nombreuses autres fonctions de conversion permettant de gérer les bases de numération, la précision, etc. (voir *go doc strconv*...)

Affectation

- L'affectation se note $=$ et permet d'affecter une valeur à une variable.
- La partie droite de l'affectation doit être une expression produisant une valeur *de même type* que la partie gauche :
  - $i = 12$ et $i16 = 12$ sont correctes car 12 est un littéral entier universel
  - $i = 12 + i16$ est incorrecte car si $i16$ est de type *int16*, alors l'expression $12 + i16$ l'est aussi et est donc incompatible avec i, qui est de type *int*.
- L'affectation peut être combinée avec n'importe quel opérateur arithmétique : $i \mathrel{*}= 12$ est équivalent à $i = i * 12$.
- Les opérateurs d'incrémentation et de décrémentation $++$ et -- sont équivalents à $+= 1$ et à $-= 1$. On peut donc écrire indifféremment $i = i + 1$, $i \mathrel{+}= 1$ et $i{+}{+}$ (idem pour la décrémentation).

Affectation

L'affectation peut s'effectuer en parallèle. On peut donc écrire, par exemple, *i, j = 12, 42* ou *val1, val2 = val2, val1*. Cette possibilité est notamment mise à profit par les fonctions qui renvoient plusieurs valeurs :

```
val1, val2 := 10, 100     // val1 vaut 10 et val2 vaut 100
val1, val2 = val2, val1   // val1 vaut maintenant 100 et val2 vaut 10...

fic, err := os.Open("fichier.txt")
// Si err != nil => problème d'ouverture
// Sinon, fic contient le descripteur du fichier ouvert
```

Remarque :

Dans l'exemple ci-dessus, notez bien la différence d'utilisation entre := et =

Entrées/Sorties

- Pour tout ce qui concerne l'affichage, on fait appel aux fonctions du module fmt, notamment à *Print()*, *Println()* et *Printf()*.
- Pour la saisie, on dispose des fonctions *Scan()*, *Scanf()* et *ScanIn()* de *fmt*

Attention

- N'oubliez pas de mettre un & devant les noms des variables passées aux fonctions *ScanXX*.
- Ces fonctions renvoient deux valeurs : le nombre de caractères saisis et une erreur (qui vaut *nil* s'il n'y a pas eu d'erreur).
- Le problème des fonctions *ScanXX* est qu'elles s'arrêtent dès qu'elles rencontrent un espace... Pour saisir une ligne de texte, il est donc préférable d'utiliser un *bufio.Scanner* et ses fonctions *Scan()* et *Text()*.

Exemples d'entrées/sorties

```
package main

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

func main() {
   var nom string
   var age int

   clavier := bufio.NewScanner(os.Stdin)  // On définit une variable qui décrit le clavier
   fmt.Print("Bonjour, comment t'appelles-tu ? ")
   clavier.Scan()                         // On attend que l'utilisateur saisisse au clavier
   nom = clavier.Text()                   // On récupère ce qui a été saisi... Comparer avec fmt.Scan(&nom)...

   fmt.Printf("Enchanté %s, quel âge as-tu ? ", nom)
   _, err := fmt.Scan(&age)
   if err != nil {           // Scan a renvoyé une erreur...
      fmt.Printf("Erreur : %s\n", err)
   }  else {
      fmt.Printf("Tu as donc %d ans\n", age)
   }
}
```

Octets, caractères et tranches de chaînes

- Une chaîne peut être assimilée à une tranche d'octets *non modifiable*, bien qu'elle soit généralement considérée comme une suite de caractères encodés en UTF-8.
- Ceci a notamment pour conséquence que la fonction *len()* appliquée à une chaîne renvoie le nombre d'octets de celle-ci et *non le nombre de caractères*.
- Une autre conséquence est que l'opérateur d'indexation *[i]* renvoie l'octet – et non le caractère – à la position *i*.
- La notation `chaine[i:j]` renvoie la sous-chaîne comprise entre l'octet à la position *i* et l'octet à la position *j* (non compris). Comme en Python, on dispose également des notations `chaine[:j]`, `chaine[i:]` et `chaine[:]`.

Octets, caractères et tranches de chaînes

- L'alphabet ASCII (donc non accentué) est codé sur 8 bits dans l'encodage UTF-8. Ce qui signifie que, pour les caractères sans accents, le nombre d'octets sera identique au nombre de caractères.
- Par contre, les caractères accentués des langues européennes sont codées sur deux octets... Un appel à *len()* ne renverra donc pas la bonne valeur. Il faudra passer par *utf8.RuneCountInString()*. Voir ce playground.

Structures de contrôle if

- Structure *if/else* : on n'utilise les parenthèses que pour indiquer les priorités.
- L'accolade ouvrante doit être sur la même ligne que le *if* et le *else*.
- La condition peut contenir une partie initialisation (voir deuxième et quatrième exemples). Cette déclaration est locale au *if* et à toutes ses branches.

```
x := f(x)
if x > max {    // On suppose que max a été défini plus haut...
   max = x
}

if x := f(); x > max {
   max = x
}

if x > y {
   max = x
} else {
   max = y
}

if x := f(); x > 0 {
   fmt.Println("Strictement positif")
} else if x == 0 {
   fmt.Prinln("Nul")
} else {
   fmt.Println("Strictement négatif")
}
```

La structures de contrôle switch

Structure *switch*, avec ou sans expression. Dans le deuxième cas, le *switch* peut être vu comme un *if* multi-branches. La clause *default* est facultative.

```
switch valeur {
   case 0, 1, 2, 3 : ...
   case 4, 5, 6, 7 : ...
   case 42 :  ...
   default : ...
}

switch {
   case x > 0 :
      fmt.Println("Strictement positif")
   case x == 0 :
      fmt.Println("Nul")
   case x < 0 :
      fmt.Println("Strictement négatif")
 }
```

La structure de contrôle for

Go ne possède qu'une seule instruction de boucle – l'instruction *for* :

```
for a < b { ... }                    // Boucle Tant Que a < b
for { ... }                          // Boucle sans fin
for i := 0; i <= 9; i++ { ... }      // Boucle Pour i de 0 à 9 par pas de 1
for i, v := range collection { ... } // i et v dépendent de la collection...
```

- À chaque itération, *range* renvoie deux valeurs : un « indice ou une clé » et la valeur située à cet endroit dans la collection.
- Si *collection* est une chaîne, *i* et *v* contiendront l'indice et la rune située à cet indice dans la chaîne (la boucle ne fonctionne donc pas en termes d'octets...)
- Si *collection* est un tableau, un pointeur vers un tableau ou une tranche, *i* et *v* contiendront l'indice et la valeur située à cet indice dans la collection.
- Si *collection* est un hachage, *i* et *v* contiendront la clé et la valeur située associée à cette clé dans le hachage. L'ordre n'est pas garanti d'un parcours à l'autre.
- L'instruction *break* permet de sortir de la boucle englobante.

TP1

Sous-programmes

Sous-programmes

- La syntaxe générale de définition d'un sous-programme est :

```
func nomSousProgramme([paramètres]) [type du résultat] { ... }
```

- Le résultat éventuel est renvoyé par l'instruction *return* :

```
func max(x, y int) int {
   if x > y {
      return x
   } else {
      return y
   }
}
```

- Un sous-programme peut renvoyer plusieurs valeurs à l'aide d'un tuple :

```
func divEntiere(dividende, diviseur int) (int, error) {
   if diviseur != 0 {
      return dividende / diviseur, nil
   } else {
      return 0, fmt.Errorf("Division par zéro impossible")
   }
}
```

Paramètres des sous-programmes

- En Go, les paramètres ne peuvent pas recevoir de valeurs par défaut (comme en Python, par exemple)
- Les sous-programmes peuvent avoir un nombre de paramètres variable :

```
func somme(vals ... int) int {
   res := 0
   for val := range vals {  // ou : for _, val := range vals
      res += val
   }
   return res
}
...
```

Appel par somme(1, 10, 100, 42) Ou, si on a un tableau d'entiers nommé tab (voir plus loin) : Appel par somme(tab...)

- On peut également fixer un nombre minimum de paramètres. Par exemple, *func somme(n1, n2 int, reste ... int)* attendra au minimum deux paramètres.

Mode de passage des paramètres

- Le seul mode de passage de paramètres autorisé en Go est le *passage par valeur* (ou *par copie*), ce qui signifie que les paramètres réels ne sont *jamais* modifiés par un sous-programme.

```
func swap(a, b int) {
      temp := a
      a = b
      b = temp
}
```

- Toutefois, grâce aux pointeurs, on peut simuler un passage *par référence* (voir plus loin).
- Les tranches et les dictionnaires étant des références (voir plus loin), un sous-programme peut donc modifier leurs contenus.

Cas de la fonction `main`

- Le passage des paramètres à la fonction principale utilise un mécanisme différent car, en ce cas, c'est le shell qui passe les paramètres au programme...
- Les éventuels paramètres passés au programme sont récupérés dans la variable *os.Args* (il faut donc importer le module *os*).
- *os.Args* est un *tableau* (voir plus loin) : le programme peut donc savoir combien on lui a passé de paramètres en utilisant la fonction *len()* qui lui indiquera le nombre d'éléments de ce tableau.
- Chaque paramètre passé par le shell sera stocké dans ce tableau, en commençant à l'indice **1** (l'élément d'indice 0 contient le nom du programme).

Attention

Chaque paramètre passé au programme est une chaîne de caractères : il faudra donc la convertir en entier si, par exemple, on attend un entier...

Exemple

```
package main

import (
   "fmt"
   "os"
)

func main() {
   nbParams := len(os.Args) - 1
   fmt.Printf("Vous m'avez passé %d paramètres\n", nbParams)

   fmt.Printf("Ce programme s'appelle %s\n", os.Args[0])
   if nbParams != 0 {
      for i := 1; i <= nbParams; i++ {
         fmt.Print(os.Args[i] + " ")
      }
   }
   fmt.Println()
}
```

Si, par exemple, vous attendez un entier comme premier paramètre :

```
func main() {
   lim, err := strconv.Atoi(os.Args[1])
   // Tester err pour être sûr que la conversion a bien marché
```

Tranches

Tranches

- Une tranche est un *tableau dynamique* (comme une liste Python) : elle peut grossir et diminuer en fonction des besoins. En Go, on utilise donc plutôt des tranches que des tableaux.
- Le plus simple, pour créer une tranche, consiste à utiliser un *littéral tranche* :

```
daltons := []string{"Joe", "Jack", "William", "Averell"}
```

- On peut aussi créer une tranche vide (qui vaudra donc `nil`) :

```
var tranche []int
tranche := []int{}
```

- Ou créer une tranche avec une taille initiale (en ce cas, ses éléments seront initialisés à zéro) :

```
tranche := make([]int, 5)   // Taille initiale de 5 éléments
tranche := make([]int, 0)   // Tranche vide
```

Tranches

- On peut créer une tranche à partir d'une autre tranche :

```
tranche := daltons[1:3]              // ["Jack", "William"]
```

- Mais, en ce cas, la tranche créée partage les mêmes données que la tranche ou le tableau initial :

```
tranche[1] = "Ma"       // daltons = ["Joe", "Jack", "Ma", "Averell"]
```

Tranches

- On peut étendre une tranche avec la fonction *append()*, qui renvoie une nouvelle tranche :

```
slice := []int{3, 1, 66}
slice = append(slice, 44)            // [3, 1, 66, 44]
slice = append(slice, 0, 1, 2)       // [3, 1, 66, 44, 0, 1, 2]
tranche := []int{99, 88, 77}
slice = append(slice, tranche...)    // [3, 1, 66, 44, 0, 1, 2, 99, 88, 77]
newSlice := append(slice, 42)        // slice n'est pas modifiée...
```

Tranches

- Pour réduire une tranche, il suffit d'en couper une tranche... Dans l'exemple qui suit, on suppose que `slice2` est réinitialisée à sa valeur initiale avant chaque opération :

```
slice2 := []int{10, 9, 8, 7, 6, 5, 4}
slice2 = slice2[:5]     // [10, 9, 8, 7, 6]
slice2 = slice2[3:]     // [7, 6, 5, 4]
slice2 = append(slice2[:3], slice2[6:]...)  // [10 9 8 4]
```

Tranches

- Appliquée à une tranche, la forme *range* de la boucle *for* renvoie un couple *indice, élément* :

```
for i, elt := range slice2 {
    fmt.Printf("Indice %d : %d\n", i, elt)
}
```

- On ne peut pas comparer deux tranches entre elles : la seule comparaison possible est celle d'une tranche avec `nil` (pour savoir si cette tranche est vide).
- Une tranche étant, en réalité, un pointeur, l'affectation d'une tranche à une autre ne copie pas les éléments de l'une dans l'autre. Les deux tranches désignent désormais les mêmes éléments (ceux de la tranche source).
- De la même façon, une tranche passée en paramètre peut être modifiée par un sous-programme, comme en Python :

```
func reinit(tab []int) {
   for i, _ := range tab {
      tab[i] = 0
   }
}
```

Pointeurs

Pointeurs : adresse d'une variable

- Toute variable est stockée dans un emplacement mémoire dont la taille dépend du type de la valeur. Cette emplacement a une *adresse* généralement représentée par un nombre hexadécimal.
- Go dispose de l'opérateur & qui renvoie l'adresse du début de la zone mémoire où est stockée la variable placée après lui :

```
var valeur = 5
fmt.Printf("L'entier valeur vaut %d et est stocké à l'adresse %p\n",
                valeur, &valeur)
```

,

- L'affichage sera de la forme :

```
L'entier valeur vaut 5 et est stocké à l'adresse 0xc0000160a0
```

- Une adresse a un type précis, appelé *pointeur* et chaque pointeur permet de contenir l'adresse d'un type de données (on a donc des pointeurs de *int16*, des pointeurs de *int32*, etc.)

Pointeurs : adresse d'une variable

- En Go, un type pointeur est introduit par l'opérateur `*` :

```
var valeur = 5          // valeur est un int
var ptrInt *int         // ptrInt est un pointeur vers un int
ptrInt = &valeur        // ptrInt "pointe" vers valeur
*ptrInt = *ptrInt + 2   // *ptrInt est un autre nom pour valeur
fmt.Printf("%d et %d\n", valeur, *ptrInt)  // 7 et 7
```

- Dans notre exemple, `valeur` et `*ptrInt` désignent le *même* emplacement mémoire... Modifier l'un revient à modifier l'autre.
- L'opérateur `&` est appelé « opérateur de référence » car il permet d'obtenir une référence (une adresse) vers un emplacement mémoire.
- L'opérateur `*` est appelé « opérateur d'indirection » car il permet de désigner un emplacement de façon indirecte (via son adresse).

Pointeurs : adresse d'une variable

- En résumé, une « variable pointeur » (comme `ptrInt`) contient l'adresse mémoire d'une autre valeur.
- La valeur « zéro » d'une variable pointeur est la valeur spéciale `nil`.
- La taille d'une variable pointeur est toujours la même : 32 bits sur les machines 32 bits, 64 bits sur les machines 64 bits – *quel que soit le type de la valeur pointée*.

Pointeurs et paramètres

- On rappelle que Go passe *toujours* les paramètres *par valeur* (ou *par copie*) : un sous-programme *ne peut pas* modifier les paramètres réels puisqu'il travaille sur des *copies*.
- Si on passe un pointeur en paramètre, le sous-programme ne pourra donc pas le modifier. Par contre, il pourra modifier la valeur pointée par celui-ci, ce qui revient à simuler un passage de paramètre *par référence*.
- En ce cas, il faudra veiller à passer l'adresse de la valeur en utilisant l'opérateur & :

```
func noSwap(a, b int) {
   temp := a
   a = b
   b = temp
}

func swap(a, b *int) {
   temp := *a
   *a = *b
   *b = temp
 }

func main() {
   val1, val2 := 10, 100
   noSwap(val1, val2)      // NON...
   swap(&val1, &val2)      // OUI !
}
```