Les bases du langage Go
Partie 1 : variables, tableaux, boucles, conditionnelles, fonctions, structures

Initiation au développement

BUT informatique, première année

Ce TP a pour but de vous présenter les bases du langage Go. C'est ce langage qui sera utilisé pour réaliser tous les exercices de ce cours d'initiation au développement. Il n'y a pas de prérequis particuliers en dehors des notions de conditionnelle, de boucle et de variable, qui ont été étudiées au lycée.

Dans tout ce TP un texte avec cet encadrement est un texte contenant une information importante, il faut donc le lire avec attention et veiller à en tenir compte.

Un texte avec cet encadrement est une remarque.

Exercice 1.0.0 : exemple d'exercice
Un texte avec cet encadrement est un travail que vous avez à faire.

Même si la plupart des exercices de ce TP peuvent vous sembler très faciles, faites les tous sérieusement : cela vous permettra de retenir la syntaxe du langage Go et d'être ainsi plus à l'aise lors des prochains TPs.

Si vous souhaitez avoir un autre point de vue sur l'apprentissage du langage Go, vous pouvez consulter le tutoriel officiel *A Tour of Go*[1] ou le site *Go by Example*[2] en faisant toute fois attention au fait que ces sites sont plutôt faits pour les personnes ayant déjà une bonne expérience de la programmation.

1. `https://tour.golang.org/`
2. `https://gobyexample.com/`

1 Un premier programme

Le programme 1 est un code Go très simple. Avant d'expliquer les différents éléments qui le constituent nous allons voir comment l'exécuter, c'est-à-dire comment le faire fonctionner sur un ordinateur.

```go
package main

import "fmt"

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

Programme 1 – hello.go

Exercice 1.1.0
Récupérez le fichier hello.go sur MADOC et enregistrez-le dans un répertoire de votre choix. Ouvrez un terminal et placez-vous dans ce répertoire. Utilisez enfin la commande go run hello.go. Si tout s'est bien passé vous avez exécuté le programme et le texte *Hello World !* s'est affiché dans votre terminal.

Nous allons maintenant analyser ce programme simple.

— La première ligne indique package main. Nous verrons plus tard dans le cours à quoi exactement cela correspond. Pour l'instant il vous suffit de retenir que tous vos fichiers de code Go doivent commencer par cette ligne.

— La deuxième ligne indique import "fmt". Cette ligne permet de récupérer du code déjà écrit par d'autres développeurs pour pouvoir le réutiliser par la suite. Le code récupéré est ici celui de la *bibliothèque* fmt, qui permet notamment d'afficher des choses dans le terminal.

— La troisième ligne indique func main() { et elle est associée à la dernière ligne } qui ferme l'accolade ouverte auparavant. Nous verrons un peu plus tard en détail à quoi correspond le mot clé func (déclaration d'une fonction) mais pour le moment il vous suffit de retenir que, car cette fonction s'appelle main, le code qui est écrit entre les accolades est celui qui sera exécuté quand on exécutera le programme.

— La quatrième ligne indique fmt.Println("Hello World !"), c'est le code principal du programme. On utilise une fonction Println. Pour savoir que c'est la fonction Println qui provient de la bibliothèque fmt (et pas une autre version de cette fonction) on préfixe le nom de cette fonction de fmt. Cette fonction a ici un argument "Hello World !" et affiche celui-ci.

Exercice 1.1.1 : bonjour
Écrivez puis testez un programme qui affiche *Bonjour !* dans le terminal où on l'exécute.

On ne va bien sûr pas apprendre par cœur le contenu de toutes les bibliothèques disponibles en Go (il y en a beaucoup). Lorsqu'on a besoin d'une fonctionnalité particulière on cherche dans la *documentation du langage*[3] si elle existe. On y trouve notamment la description de toutes les bibliothèques fournies de base avec le langage : la *bibliothèque standard*[4] (qui contient par exemple fmt). Il existe aussi de nombreuses autres bibliothèques, développées par des tiers.

3. https ://golang.org/doc/
4. https ://pkg.go.dev/std

Dans toute la suite on créera nos programmes sur ce modèle (package main, func main() {}) et on les exécutera en utilisant la commande go run.

2 Variables

Quand on écrit un programme informatique on a, en général, besoin d'enregistrer des valeurs pour les réutiliser plus tard. Pour cela on utilise des *variables*.

2.1 Déclaration de variables

Lorsqu'on crée une nouvelle variable on dit qu'on la *déclare*. Le programme 2 montre quelques exemples de déclarations de variables en Go.

```go
package main

import "fmt"

func main() {
    var n int
    fmt.Println(n)

    var b bool
    fmt.Println(b)

    var s string
    fmt.Println(s)
}
```

Programme 2 – vars.go

On peut remarquer que la déclaration d'une variable se fait avec le mot clé var, suivi du nom de la variable, suivi d'un *type*. Ce type indique la nature des choses qu'on stockera dans la variable. Toute variable a obligatoirement un type qui est fixé une fois pour toute : on ne peut pas le changer après la déclaration.

Dans ce programme on observe trois types différents : int qui représente un entier, bool qui représente un booléen (vrai ou faux) et string qui représente une chaîne de caractères (une séquence de symboles, comme par exemple "Hello World !").

On verra d'autre types par la suite.

2.2 Valeurs par défaut

Exercice 1.2.0
Récupérez le fichier vars.go sur MADOC puis exécutez-le.

On remarque que, bien qu'on n'ait pas donné de valeurs aux variables, celles-ci en ont tout de même : 0 pour l'entier, false pour le booléen, et "" (c'est-à-dire la chaîne vide) pour la chaîne de caractères.

En Go, contrairement à beaucoup d'autres langages, toute variable a une valeur par défaut, qui dépend de son type. Si vous créez un entier, par exemple, vous avez la garantie que, tant que vous ne le modifiez pas, il vaut 0.

2.3 Variables inutilisées

Exercice 1.2.1
Modifiez le fichier vars.go en enlevant une (et une seule) des utilisations de Println puis exécutez-le à nouveau. Vous devez observer une erreur.

L'erreur observée (dans le cas où on supprime l'affichage de b) doit normalement ressembler à ceci : ./vars.go :8 :6 : b declared but not used. Elle indique que, dans le fichier vars.go, à la 6ème lettre de la ligne 8, une variable b est déclarée qui n'est ensuite jamais utilisée dans le reste du programme.

En Go, si on déclare une variable on doit obligatoirement l'utiliser ensuite.

En enlevant toutes les utilisations de Println à vars.go on observe une autre erreur qui nous indique qu'on importe fmt mais qu'on ne l'utilise pas et qu'il faut donc retirer toute la ligne import "fmt".

2.4 Affectation de variables

Tout l'intérêt des variables est de pouvoir changer leur valeur, on dit qu'on les *affecte*. On peut le faire en utilisant le signe = comme le montre le programme 3.

```go
package main

import "fmt"

func main() {
    var n int = 1
    fmt.Println(n)
    n = 2
    fmt.Println(n)
}
```

Programme 3 – vars2.go

On notera qu'on peut affecter une variable dès sa déclaration var n int = 1 ou plus tard n = 2 et dans ce cas on écrit seulement le nom de la variable à gauche du signe = (on ne souhaite pas la redéclarer).

Exercice 1.2.2
Récupérez le fichier vars2.go sur MADOC puis exécutez-le.

Exercice 1.2.3
Modifiez le programme hello.go pour stocker le texte dans une variable avant de l'afficher. Testez votre programme.

2.5 Calculs

Enfin, on peut utiliser nos variables pour faire des calculs. On dispose des opérations standards associées au type des variables manipulées, en particulier :

— l'addition (+), la soustraction (-), la multiplication (*) et la division (/) pour les entiers,
— le OU (||), le ET (&&) et le NON (!) pour les booléens,

— la concaténation (+) pour les chaînes de caractères,
— mais aussi les comparaisons (supérieur strict >, supérieur ou égal >=, inférieur strict <, inférieur ou égal <=, égal ==, différent !=) qui traitent des entiers mais donnent en résultat des booléens.

Il existe d'autres opérations et des raccourcis syntaxiques pour les cas d'utilisations les plus communs de celles-ci, que vous découvrirez au fur et à mesure de votre apprentissage du langage.

Le programme 4 donne quelques exemples d'utilisation de ces opérations. On peut noter que ce n'est pas un problème d'écrire n = n+10 : le calcul à droite du signe = est réalisé avant de changer la valeur de la variable à gauche de celui-ci. On utilisera très souvent ce genre de codes où la valeur d'une variable dépend de sa valeur précédente.

```go
package main

import "fmt"

func main() {
    var n int = 3 * 5
    n = n + 10
    fmt.Println(n)

    var b bool = true
    b = !b
    fmt.Println(b)

    var s string = "Bon"
    s = s + "jour"
    fmt.Println(s)

    fmt.Println(3 == 5)
    fmt.Println(3 != 5)
    fmt.Println(3 >= 5)
    fmt.Println(3 <= 5)
}
```

Programme 4 – calcul.go

Exercice 1.2.4
Récupérez le fichier calcul.go sur MADOC puis exécutez-le.

Exercice 1.2.5
Écrivez un programme qui, à partir de deux variables contenant les chaînes de caractères "Hello" et "World" utilise la concaténation pour construire la chaîne "Hello World !" puis l'affiche. N'oubliez pas l'espace et le point d'exclamation.

Exercice 1.2.6
À l'aide d'un programme simple donnez la valeur de $(6x + 4)/2$ quand x vaut 5. Modifiez votre programme pour donner cette valeur quand x vaut 7. Seule la valeur initiale d'une variable doit être modifiée pour passer d'un programme à l'autre.

Exercice 1.2.7
Écrivez un programme qui affiche true si un nombre entier stocké dans une variable n est strictement plus grand que 5 et qui affiche false sinon.

En Go on a accès à l'opération modulo, qui calcule le reste d'une division euclidienne (une division du entier par un entier). Ainsi n % m calcule le reste de la division de n par m. On a par exemple 7 % 2 qui vaut 1 car $7 = 2 * 3 + 1$ (quotient de 3, reste de 1).

Exercice 1.2.8
Écrivez un programme qui, étant donné un temps t en seconde, calcule son équivalent en heures/minutes/secondes et l'affiche. Le nombre de secondes et le nombre de minutes dans cet équivalent doivent bien sûr toujours être inférieurs à 60. Testez votre programme avec plusieurs temps t différents afin d'avoir confiance dans le fait qu'il est correct.

Il existe d'autres façon de déclarer une variable en Go. À la place de faire var n int = 1 on peut faire var n = 1 ou encore n := 1. Ceci utilise un mécanisme d'*inférence de type* : Go devine quel doit être le type de la variable n à partir de la valeur que l'on donne à celle-ci. Comme il est important d'être bien conscient des types des variables que l'on manipule, il vous est demandé de ne pas utiliser ces manières de déclarer les variables dans les cours d'initiation au développement.

En plus de déclarer des variables on peut, en Go, déclarer des constantes. Pour cela on remplace simplement var par const. Les constantes fonctionnent comme les variables, mais on est obligé de les affecter à la déclaration (pas de valeur par défaut) et on ne peut plus les modifier ensuite.

3 Tableaux

Un tableau est une variable particulière qui contient n valeurs indexées par des entiers entre 0 et $n-1$. Ces valeurs *doivent toutes être du même type*. Cela peut être utile, par exemple, pour représenter des objets mathématiques, comme les vecteurs ou les matrices, ou pour représenter un ensemble ordonné de valeurs.

3.1 Déclaration

Pour déclarer un tableau on peut simplement créer une variable dont le type indique le type des valeurs qui seront contenues dans ce tableau, précédé de crochets. Par exemple, pour un tableau d'entiers, []int. Le programme 5 donne quelques exemples de déclarations de tableaux.

Exercice 1.3.0
Récupérez le fichier tabs.go sur MADOC puis exécutez-le. Quelle est la valeur par défaut d'une variable de type tableau ?

3.2 Accès aux valeurs et affectations

Comme pour les autres variables on peut bien sûr affecter une valeur à une variable de type tableau en utilisant le signe =. Il faut cependant utiliser une syntaxe un peu plus complexe que pour les types de base pour représenter la valeur du tableau, comme on peut le voir dans le programme 6.

```go
package main

import "fmt"

func main() {
    var tint []int
    var tbool []bool
    var tstring []string
    fmt.Println(tint, tbool, tstring)
}
```

Programme 5 – tabs.go

```go
package main

import "fmt"

func main() {
    var tint []int = []int{1, 2, 3, 1000, 27}
    var tbool []bool = []bool{true, false, false}
    fmt.Println(tint, tbool)
}
```

Programme 6 – tabs2.go

Exercice 1.3.1
Récupérez le fichier tabs2.go sur MADOC et exécutez-le. Modifiez-le ensuite pour affecter et afficher un tableau de chaînes de caractères contenant, dans cette ordre, les chaînes "Bonjour", "le" et "monde". Testez votre modification.

On peut aussi accéder à une valeur dans une case d'un tableau. Pour cela on va indiquer entre crochets après le nom du tableau le numéro de cette case. Le programme 7 montre ceci. On voit aussi qu'on peut affecter une valeur à une case d'un tableau sans modifier les autres en accédant à cette valeur et en utilisant le signe =.

```go
package main

import "fmt"

func main() {
    var tint []int = []int{1, 2, 3, 1000, 27}
    fmt.Println(tint[0])
    fmt.Println(tint[3])
    fmt.Println(tint)
    tint[3] = 0
    fmt.Println(tint)
}
```

Programme 7 – tabs3.go

Attention, les cases d'un tableau sont numérotées à partir de 0 : la première case est la case 0 et pas la case 1.

Exercice 1.3.2
Récupérez le fichier tabs3.go sur MADOC et exécutez-le. Modifiez ce programme pour afficher la valeur de la dernière case du tableau. Testez votre code. Modifiez à nouveau le programme pour essayer d'afficher une case du tableau qui n'existe pas (la case numéro 50 par exemple). Testez votre code et analysez l'erreur qui s'affiche pour bien la comprendre.

3.3 Taille

Pour éviter de dépasser les limites d'un tableau, il peut être utile de connaître sa longueur, c'est-à-dire son nombre de cases. On peut faire ceci en Go à l'aide de la fonction len, comme le montre le programme 8.

```go
package main

import "fmt"

func main() {
    var tint []int = []int{1, 2, 3, 1000, 27}
    fmt.Println(len(tint))
    tint = []int{4, 5, 6}
    fmt.Println(len(tint))
}
```

Programme 8 – tabs4.go

Exercice 1.3.3
Récupérez le fichier tabs4.go sur MADOC et exécutez-le. Comment peut-on avoir le numéro de la dernière case d'un tableau t à partir du résultat de len(t) ? Qu'est-ce-que cela veut dire si on trouve que le numéro de cette case est -1 ?

4 Boucles

Lorsqu'on exécute un programme Go tels qu'on les a vus jusqu'à présent, les lignes de code de la fonction main sont exécutées une à une, dans l'ordre où elles sont écrites, puis le programme s'arrête. Ceci ne permet pas de réaliser des calculs très intéressants (comment faire, par exemple, pour afficher l'un après l'autre tous les éléments contenus dans un tableau). On utilisera donc, en plus, des *structures de contrôle* qui vont permettre d'exécuter certaines parties de notre code plus ou moins de fois et dans des ordres différents en fonction de *conditions* sur les valeurs de variables ou sur les résultats de calculs. Ces structures de contrôle sont de deux types : les *conditionnelles* (que nous verrons un peu plus tard) et les boucles.

4.1 La boucle for, utilisation de base

Selon les langages de programmation qu'on utilise on a accès à plus ou moins de différents types de boucles (for, while, do while, etc), cependant, en Go, il n'existe qu'un seul type de boucle : la boucle for.

Le programme 9 montre un premier exemple de boucle for, dans sa forme la plus simple.

On peut schématiser cette boucle ainsi : for initialisation ; condition ; mise-à-jour {code} où les différents éléments ont les caractéristiques suivantes :

```go
package main

import "fmt"

func main() {
    for i := 0; i < 10; i = i + 2 {
        fmt.Println(i)
    }
}
```

Programme 9 – for.go

initialisation. Création d'une variable de boucle, ici la variable est i et est initialisée à 0 par la code i := 0.

condition. Condition sur la valeur de la variable de boucle qui indique quand la boucle doit continuer à s'exécuter. Ici, la boucle s'exécute tant que i est strictement plus petit que 10, ce qui est exprimé par le code i < 10.

mise-à-jour. Changement appliqué à la variable de boucle à chaque *itération* de celle-ci. Ici on ajoute 2 à i, ce qui est représenté par le code i = i + 2.

code. Code qui est exécuté à chaque itération de la boucle.

En d'autres terme, cette boucle va initialiser une variable i à 0, puis exécuter le code fmt.Println(i), puis changer la valeur de i en lui ajoutant 2, exécuter à nouveau le code, ajouter 2 à i, et ainsi de suite jusqu'à ce que en ajoutant 2 à i sa valeur devienne supérieure ou égale à 10.

Exercice 1.4.0
Sans l'exécuter, indiquez ce qu'affiche le programme 9. Récupérez ensuite le fichier for.go sur MADOC et exécutez-le pour vérifier votre réponse.

Exercice 1.4.1
Écrivez un programme qui affiche tous les multiples de 5 compris entre 10 et 100 inclus. Testez votre programme pour vous assurer qu'il est correct.

Exercice 1.4.2
Écrivez un programme qui, à l'aide d'une boucle, affiche 10 fois *Bonjour*.

Exercice 1.4.3
Écrivez un programme qui déclare un tableau d'entiers contenant 10 valeurs de votre choix puis qui, à l'aide d'une boucle for, affiche tous ces entiers dans l'ordre du tableau. Votre variable de boucle servira à déterminer le numéro de la case dont la valeur doit être affichée à chaque tour de boucle. La fonction len vous permettra d'arrêter votre boucle au bon moment. Testez votre programme. Modifiez la déclaration de votre tableau pour qu'il contienne maintenant 12 valeurs. Sans rien modifier d'autre, votre programme doit toujours fonctionner. Testez-le. Changez maintenant votre tableau pour qu'il ne contienne aucune valeur, votre programme doit toujours fonctionner (il n'affiche pas de valeurs puisque le tableau est vide, mais il ne doit pas non plus afficher d'erreur). Testez-le.

En écrivant le programme de l'exercice précédent, vous avez sans doute mis-à-jour la variable de boucle en écrivant i = i + 1. Il existe un raccourci d'écriture pour cette opération : i++

Exercice 1.4.4
Modifiez le programme for.go pour afficher la valeur de i après la fin de la boucle. Testez-le et expliquez l'erreur que vous observez.

4.2 Variable de boucle déclarée à l'extérieur

On a vu, avec le dernier exercice, qu'une fois la boucle terminée, on ne pouvait plus accéder à la valeur de la variable de boucle. Ceci est dû au fait qu'une variable a une durée de vie limitée (pour économiser des ressources mémoire). Dans le cas de la boucle for précédente, la variable a été déclarée dans l'initialisation de la boucle, sa durée de vie est celle de la boucle, et par conséquent elle n'existe plus une fois que celle-ci est terminée.

Si on a besoin d'accéder à une variable de boucle après la fin de celle-ci, il est possible en Go de déclarer cette variable en dehors de la boucle, comme le montre le programme 10.

```
package main

import "fmt"

func main() {
    var i int
    for i = 0; i < 10; i = i + 2 {
        fmt.Println(i)
    }
    fmt.Println(i)
}
```

Programme 10 – for2.go

Exercice 1.4.5
Récupérez le programme for2.go sur MADOC et testez-le.

Si on déclare une variable dans le code exécuté par une boucle, cette variable est redéclaré à chaque tour de boucle : sa durée de vie est celle d'une itération. Pour qu'une variable puisse être mise-à-jour par une boucle il faut donc aussi la déclarer à l'extérieur de celle-ci.

Exercice 1.4.6
Écrivez un programme qui déclare un tableau d'entiers contenant 10 valeurs de votre choix puis qui, à l'aide d'une boucle for, calcule la somme de ces 10 entiers. Testez-le. Modifiez la déclaration de votre tableau pour qu'il contienne maintenant 12 valeurs. Sans rien modifier d'autre, votre programme doit maintenant calculer la somme des 12 entiers. Testez-le

4.3 Intermède : le type float64

En Go il existe bien sûr un type pour représenter les nombre décimaux. Il s'agit du type float64 (il existe en fait aussi float32). On peut donc déclarer une variable x contenant une valeur décimale en écrivant var x float64. Et on peut lui attribuer une valeur comme pour les autres types (la virgule se représente par un point) : x = 12.5 par exemple.

Exercice 1.4.7
Écrivez un programme vous permettant de trouver quelle est la valeur par défaut d'une variable de type float64.

Exercice 1.4.8
Écrivez un programme qui à partir d'un tableau contenant les notes sur 20 d'un étudiant au cours d'un semestre calcule la moyenne de cet étudiant.

4.4 Boucle avec seulement une condition

La condition utilisée pour définir la fin d'une boucle n'est en fait pas obligée de concerner la variable de boucle. On n'est d'ailleurs pas obligé d'avoir une variable de boucle. Le programme 11 montre un exemple de boucle for sans variable de boucle, cette boucle n'a donc qu'une condition (pas d'initialisation et pas de mise-à-jour).

```
package main

import "fmt"

func main() {
    var reste int = 32
    var quotient int = 0
    const diviseur int = 4
    for reste >= diviseur {
        reste = reste - diviseur
        quotient++
    }
    fmt.Println(quotient)
}
```

Programme 11 – for3.go

Exercice 1.4.9
Récupérez le fichier for3.go sur MADOC. Exécutez-le plusieurs fois en modifiant les valeurs affectées initialement aux variables reste et diviseur. Que calcule ce programme ?

Exercice 1.4.10
Écrivez sur le même modèle un programme qui calcule le produit de deux variables entières sans utiliser l'opération de multiplication

On peut convertir un entier x en flottant en écrivant float64(x)

Exercice 1.4.11
Lors de la naissance de sa petite fille Anaïs, son grand-père lui a ouvert un compte épargne en y déposant 100€. À chaque anniversaire d'Anaïs, il verse sur ce compte un montant correspondant à 10 fois l'age qu'elle vient d'avoir. (Par exemple, à 5 ans, Anaïs va recevoir 50€.) En outre, le compte épargne d'Anaïs a un taux fixe de 3,5% dont les intérêts sont versés à chaque date anniversaire. Écrivez un programme qui étant donné un montant en euros, indique à Anaïs l'age qu'elle devra avoir pour en disposer sur son compte.

La syntaxe des boucles en Go est en fait un peu plus générale que ce que nous avons vu ici. On peut par exemple écrire une boucle infinie en laissant vide l'espace réservé à la condition de boucle : for {code}.

5 Conditionnelles

Nous allons maintenant nous intéresser à un deuxième type de structures de contrôle : les conditionnelles. Ces structures permettent de faire des choix en fonction de valeurs de variables ou de résultats de calculs.

5.1 if

La conditionnelle la plus simple est le if, elle permet d'exécuter un code parmi deux en fonction d'une condition. Le programme 12 montre un exemple d'utilisation de cette conditionnelle.

```go
package main

import "fmt"

func main() {
    var n int = 10
    if n < 5 {
        fmt.Println("Moins que 5")
    } else {
        fmt.Println("5 ou plus")
    }
}
```

Programme 12 – if.go

Le schéma général du if est le suivant : if condition {code1} else {code2}. Si la condition est vérifiée (dans l'exemple, si n est strictement inférieur à 5) alors on exécute le code code1 (ici, afficher *moins que 5*). Si la condition n'est pas vérifiée (dans l'exemple, si n est supérieur ou égal à 5) alors on exécute le code code2 (ici, afficher *5 ou plus*). On appellera code1 et code2 les *branches* de la conditionnelle (qui est parfois appelée branchement).

Exercice 1.5.0
Récupérez le fichier if.go sur MADOC et testez-le. Modifiez la valeur de n pour constater que le programme peut bien exécuter le code de l'une ou l'autre des branches de la conditionnelle.

Exercice 1.5.1
Un magasin propose à ses clients une remise de 5% (arrondi à l'entier inférieur) sur le montant total des achats lorsque celui-ci dépasse 300€. Écrivez un programme qui à partir d'un montant d'achat en euros (stocké dans une variable entière) affiche la somme à payer par le client. Testez votre programme pour différents montants d'achats (en particulier des montants inférieurs à 300€ et des montants supérieurs à 300€)

Exercice 1.5.2
Écrivez un programme qui déclare trois variables entières puis trouve et affiche la valeur de la plus grande d'entre-elles. Testez-le pour différentes valeurs des variables.

Exercice 1.5.3
Écrivez un programme qui déclare deux variables entières puis indique si leur produit est positif, négatif ou nul sans calculer ce produit. Testez-le pour différentes valeurs des variables.

Exercice 1.5.4
Écrivez un programme qui indique si une année (stockée dans une variable entière) est bissextile ou non. Les années bissextiles sont celles qui sont divisibles par 400 ou divisibles par 4 mais pas par 100.

La partie else est facultative, si on l'omet le code est exécutée si la condition est vérifiée et il ne se passe rien sinon. Le programme 13 en donne un exemple.

```go
package main

import "fmt"

func main() {
    var n int = 10
    if n < 5 {
        fmt.Println("Moins que 5")
    }
    fmt.Println("Au revoir")
}
```

Programme 13 – if2.go

Dans tous les cas le programme affiche *Au revoir*, par contre il n'affiche *Moins que 5* que si n est strictement inférieur à 5.

Exercice 1.5.5
Récupérez le fichier if2.go sur MADOC et testez-le. Modifiez la valeur de n pour constater ce qu'il se passe lorsque la condition est vraie et lorsqu'elle est fausse.

Exercice 1.5.6
Écrivez un programme qui déclare un tableau d'entiers puis qui, à l'aide d'une boucle for, affiche ceux qui sont strictement inférieurs à 5, dans l'ordre du tableau. Testez votre programme pour plusieurs tableaux de tailles et de contenus différents.

Exercice 1.5.7
Écrivez un programme qui déclare un entier n et un tableau d'entiers puis qui compte (et affiche) combien de fois l'entier n apparaît dans le tableau. Testez votre programme pour plusieurs tableaux de tailles et de contenus différents, ainsi que pour plusieurs valeurs de n.

5.2 Retour sur les boucles

On peut stopper une boucle de manière anticipée en utilisant le mot clé break : au moment où ce mot clé est rencontré dans le code de la boucle, celle-ci se termine immédiatement. Ceci est particulièrement utile en combinaison avec des conditionnelles. Le programme 14 montre un exemple d'utilisation de break.

```go
package main

import "fmt"

func main() {
    var n int = 10
    var p int = 5
    for i := 0; i < n; i++ {
        fmt.Println("L'itération", i, "commence")
        if i == p {
            break
        }
        fmt.Println("L'itération", i, "termine")
    }
}
```

Programme 14 – break.go

Dans ce programme, n tours de boucle devraient avoir lieu. Cependant, au sixième tour, quand la variable i vaut p, la condition de la conditionnelle est vérifiée. On exécute donc le code de la première (et seule) branche de cette conditionnelle, qui est un break, ce qui fait terminer la boucle.

Exercice 1.5.8
Récupérez le fichier break.go et exécutez-le. Faites quelques tests en modifiant n et p, pour bien comprendre le fonctionnement de break.

Exercice 1.5.9
Écrivez un programme qui déclare un tableau d'entiers, puis affiche une à une, dans l'ordre, les valeurs contenues dans ce tableau à l'aide d'une boucle en s'arrêtant dès qu'il rencontre la valeur 3. Testez-le avec des tableaux contenant la valeur 3 et des tableaux ne la contenant pas.

Dans le même ordre d'idée, on peut stopper l'itération en cours d'une boucle (et passer directement au début de la suivante) en utilisant le mot clé continue.

Exercice 1.5.10
Remplacez le mot clé break par le mot clé continue dans le programme du fichier break.go. Testez-le pour différentes valeurs de n et p.

5.3 switch

Si on veut faire un choix entre plus de deux alternatives, on peut imbriquer des conditionnelles en enchaînant directement un if après un else comme le montre le programme 15.

```go
package main

import "fmt"

func main() {
    var n int = 10
    if n < 5 {
        fmt.Println("Moins que 5")
    } else if n <= 10 {
        fmt.Println("Entre 5 et 10")
    } else {
        fmt.Println("Plus de 10")
    }
}
```

Programme 15 – if3.go

Exercice 1.5.11
Récupérez le fichier if3.go sur MADOC et testez-le pour différentes valeurs de n, de façon à observer les trois affichages possibles.

Si on a beaucoup d'alternatives, on se rend bien compte que cette imbrication de conditionnelles peut devenir peu pratique à écrire. On peut alors la remplacer par un switch, qui est une autre structure de contrôle. Le programme 16 est une réécriture du programme 15 à l'aide d'un switch.

```go
package main

import "fmt"

func main() {
    var n int = 10
    switch {
    case n < 5:
        fmt.Println("Moins que 5")
    case n <= 10:
        fmt.Println("Entre 5 et 10")
    default:
        fmt.Println("Plus de 10")
    }
}
```

Programme 16 – switch.go

Exercice 1.5.12
Récupérez le fichier switch.go sur MADOC et testez-le pour les mêmes valeurs de n que le programme précédent. Constatez que ces deux programmes semblent bien donner les mêmes résultats.

Un switch teste les conditions de ses différents case une à une dans l'ordre et exécute uniquement le code correspondant à la première de ces conditions qui est vérifiée.

6 Fonctions

On a utilisé dans le début de ce document le nom de fonction sans vraiment dire de quoi il s'agissait. Une fonction est un bout de code auquel on donne un nom et qui, à partir d'*arguments* va retourner des résultats ou *valeurs de retour*. Un intérêt des fonctions est d'éviter de réécrire plusieurs fois un même code (ce qui peut être source d'erreurs).

La fonction main est une fonction particulière : elle n'a pas d'arguments et ne retourne pas de résultats. De plus, elle est automatiquement appelée au début de l'exécution du programme.

6.1 Déclaration de fonction

Pour déclarer une fonction à un argument et une valeur de retour on utilise le mot clé func suivi du nom de la fonction, du nom et type de son argument entre parenthèses, du type de sa valeur de retour, puis d'accolades qui entourent le code de cette fonction.

Pour utiliser cette fonction (on dit qu'on l'*appelle*) on écrit simplement son nom suivi de la valeur qu'on souhaite utiliser pour son argument entre parenthèses.

Le programme 17 déclare une telle fonction f puis l'utilise à l'intérieur de la fonction main avec différentes valeurs de son argument.

```go
package main

import "fmt"

func f(x int) bool {
    return x < 10
}

func main() {
    fmt.Println(f(3))
    fmt.Println(f(5))
    fmt.Println(f(7))
    fmt.Println(f(11))
    fmt.Println(f(13))
}
```

Programme 17 – func.go

L'argument de la fonction f est un entier qui s'appelle x : à l'intérieur du code de la fonction, quand on fait référence à x cela correspond à la valeur de cet argument. Cette fonction retourne (avec le mot clé return) un booléen.

Quand on utilise, par exemple, f(3) cela correspond à exécuter le code de la fonction f avec x qui vaut 3, jusqu'à obtenir la valeur du return et à remplacer f(3) par cette valeur.

Exercice 1.6.0
Récupérez le fichier func.go sur MADOC puis exécutez-le. Modifiez la valeur du return pour voir l'effet que cela a sur le programme. Essayez aussi d'appeler la fonction avec d'autres valeurs de son argument.

Exercice 1.6.1
Dans un nouveau programme, déclarez une fonction qui prend en paramètre un tableau d'entiers et retourne la valeur de la case 3 de ce tableau. Dans le main utilisez cette fonction avec plusieurs tableaux différents, en affichant à chaque fois le résultat obtenu.

6.2 Fonctions à plusieurs arguments

Une fonction peut avoir plusieurs arguments, on les sépare par des virgules lors de la déclaration de la fonction et on sépare leurs valeurs par des virgules (en respectant leur ordre de déclaration) lors de l'appel de la fonction.

Le programme 18 donne un exemple de déclaration et d'utilisation d'une fonction g à deux arguments.

```go
package main

import "fmt"

func g(x int, y int) bool {
    return x < y
}

func main() {
    fmt.Println(g(3, 4))
    fmt.Println(g(4, 3))
    fmt.Println(g(5, 5))
}
```

Programme 18 – func2.go

Les arguments de g sont deux entiers qui s'appellent x et y. Cette fonction retourne un booléen qui indique si x est strictement plus petit que y ou pas.

Exercice 1.6.2
Récupérez le fichier func2.go sur MADOC puis exécutez-le. Essayez d'appeler la fonction avec d'autres valeurs de ses arguments.

Exercice 1.6.3
Dans un nouveau programme, déclarez une fonction qui prend en paramètre un tableau d'entiers et un entier n et retourne la valeur de la case n de ce tableau. Dans le main utilisez cette fonction avec plusieurs tableaux différents et plusieurs valeurs de n différentes, en affichant à chaque fois le résultat obtenu.

Exercice 1.6.4
Deux nombres entiers positifs n et m sont dits amicaux si la somme des diviseurs de n (excepté n) vaut m et si la somme des diviseurs de m (excepté m) vaut n. Écrivez une fonction qui calcul et retourne la somme des diviseurs d'un nombre entier positif donné en argument. Testez cette fonction sur plusieurs nombres en n'oubliant pas de tester les cas particuliers : 0, nombres premiers, etc. Écrivez ensuite une fonction qui, à partir de la fonction précédente, indique (par un booléen) si deux nombres entiers positifs qui lui sont donnés en arguments sont amicaux ou pas. Testez cette fonction sur plusieurs couples de nombres.

On peut bien sûr avoir plus de deux arguments pour une fonction.

6.3 Fonctions retournant plusieurs valeurs

Une fonction peut aussi retourner plusieurs valeurs, comme le montre le programme 19.

```
package main

import "fmt"

func g(x int, y int) (bool, int) {
    return x < y, x + y
}

func main() {
    var b bool
    var n int
    b, n = g(3, 4)
    fmt.Println(b)
    fmt.Println(n)
}
```

Programme 19 – func3.go

Les types des valeurs de retour (ici bool et int) sont indiqués entre parenthèses et séparés par des virgules à la place habituelle du type de la valeur de retour. Le retour lui même est fait en indiquant les valeurs, dans le même ordre que leurs types ont été indiqués, séparées par des virgules (ici x < y et x + y). On peut récupérer ces valeurs dans des variables (ici b, et n) en les mettant à gauche d'un =, séparées par des virgules.

Exercice 1.6.5
Récupérez le fichier func3.go sur MADOC puis exécutez-le. Essayez d'appeler la fonction avec d'autres valeurs de ses arguments.

Exercice 1.6.6
Dans un nouveau programme, déclarez une fonction qui prend en paramètre un tableau d'entiers et un entier n et retourne 0 et false si le tableau a moins de n+1 cases et la valeur de la case n de ce tableau et true si le tableau à n+1 cases ou plus. Dans le main utilisez cette fonction avec plusieurs tableaux différents et plusieurs valeurs de n différentes, en affichant à chaque fois le résultat obtenu.

Comme on l'a vu dans l'exercice précédent, les fonctions à plusieurs valeurs de retour sont notamment utilisées en Go pour retourner le résultat d'un calcul et une information sur le succès ou l'échec de ce calcul en une seule fois. Quand vous utiliserez des fonctions provenant de bibliothèques vous verrez que c'est une pratique assez commune.

On peut bien sûr avoir plus de deux valeurs de retour pour une fonction.

Exercice 1.6.7
Écrivez une fonction qui prend en arguments deux nombre décimaux x et e et indique (par un booléen) s'il existe un entier positif n tel que $x^n < e$ et, si c'est le cas, retourne le plus petit n vérifiant cette propriété. Dans votre main testez votre fonction pour plusieurs valeurs de x et de e. (Il existe une fonction pour calculer des puissances dans la bibliothèques de maths, mais normalement vous n'en avez pas besoin, et ne devriez pas l'utiliser, ici.)

7 Types structurés

Les types de base (int, float, string, bool, etc) ne sont pas toujours suffisants pour représenter toutes les données qu'on voudrait manipuler. On a vu qu'on peut réunir des éléments d'un même type au sein de tableaux ([]int, []float, etc) ou de maps (map[string]int par exemple), mais il pourrait être intéressant de réunir des éléments de types différents, ou d'avoir des structures plus complexes.

Le mot clé struct permet de définir des nouveaux types, constitués de différents *champs* qui peuvent chacun être d'un type différent. Le programme 20 donne un exemple de création d'un nouveau type cuple qui contient deux champs de type int, nommés first et second.

```
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c cuple
    fmt.Println(c)
}
```

Programme 20 – struct.go

Exercice 1.7.0
Récupérez le fichier struct.go sur MADOC et exécutez-le. Quelle est la valeur par défaut d'une variable de type cuple ? Quel doit être la valeur par défaut d'une variable d'un type structuré quelconque ?

Il est bien sûr possible de fixer soit même les valeurs des champs d'une variable d'un type structuré lorsqu'on la crée. Le programme 21 montre comment faire cela.

Exercice 1.7.1
Récupérez le fichier struct2.go sur MADOC et exécutez-le. Essayez de donner d'autres valeurs aux champs. Que se passe-t-il lorsqu'on ne définit pas les valeurs de tous les champs ? (quelle est la valeur des autres champs ?)

On n'est pas obligé de donner les noms des champs lorsqu'on fixe la valeur d'une variable d'un type structuré (dans le programme ci-dessus on pourrait écrire var c1 cuple = cuple{1, 2}), mais on doit alors obligatoirement définir tous les champs et les définir dans l'ordre où ils sont déclarés dans la structure.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c1 cuple = cuple{first: 1, second: 2}
    var c2 cuple = cuple{first: 1}
    var c3 cuple = cuple{second: 2}
    fmt.Println(c1, c2, c3)
}
```

Programme 21 – struct2.go

Enfin, on peut accéder indépendamment aux différents champs d'une variable d'un type structuré, et les modifier. Le programme 22 montre comment faire cela.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c, c.first, c.second)

    c.first = 3
    fmt.Println(c, c.first, c.second)
}
```

Programme 22 – struct3.go

Exercice 1.7.2
Récupérez le fichier struct3.go sur MADOC et exécutez-le. Modifiez le programme pour changer la valeur du champs second et testez-le à nouveau.

Quand on définit un type structuré dans un paquet, si le nom du type commence par une majuscule, alors ce type sera utilisable dans d'autres paquets (à condition d'importer tout ce qu'il faut). De même, les noms des champs ne seront visibles dans d'autres paquets que s'ils commencent par des majuscules. Ainsi, on peut avoir un type dont certains champs sont accessibles à l'extérieur du paquet où il est définit, mais d'autres champs ne sont accessibles que dans ce paquet. Ceci permet de n'exporter que les éléments qui sont pertinents.

Avec type on peut aussi donner des noms à des types déjà existants, par exemple type myint int définit un type myint qui correspond au type int. Ceci sera utile en lien avec la partie suivante sur les méthodes.

8 Méthodes

On appelle *méthode* une fonction particulière qui est fortement associée à un type. Vous en avez déjà rencontré, ce sont les fonctions comme Close() que nous utilisions pour fermer un fichier f en écrivant f.Close(). En pratique, on pourrait remplacer toute méthode par une fonction classique mais les méthodes permettent de mieux structurer le code, en associant des fonctionnalités à des types.

Pour définir une méthode on fait comme pour une fonction normale, mais le type sur lequel elle doit s'appliquer est indiqué avant le nom de la fonction. Le programme 23 donne un exemple de déclaration et d'utilisation d'une méthode.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func (c cuple) add() int {
    return c.first + c.second
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c.add())
}
```

Programme 23 – methode.go

Dans ce programme on définit une méthode add qui s'applique sur une variable de type cuple, sans prendre de paramètres supplémentaires, et qui retourne un entier (qui est la somme des deux champs de la variable sur laquelle on applique la méthode).

Exercice 1.8.0
Récupérez le fichier methode.go sur MADOC et testez-le.

Une méthode peut bien sûr modifier la valeur de la variable à laquelle elle s'applique. Cependant, comme d'habitude, pour cela il faut utiliser un pointeur (le passage d'arguments se fait toujours par valeur). Le programme 24 donne un exemple de cela.

Dans ce programme on définit une méthode exchange qui échange les valeurs des deux champs (first et second) d'une variable de type cuple.

Exercice 1.8.1
Récupérez le fichier methode2.go sur MADOC et testez-le. Que remarquez-vous sur la manière dont on appelle la méthode exchange ?

On ne peut définir des méthodes que sur des types qui sont définit au sein du même paquet. Ainsi, vous ne pouvez par exemple pas définir une méthode sur des entiers (int) mais vous pouvez redéfinir un type qui correspond aux entiers et une méthode sur ce nouveau type.

```go
package main

import "fmt"

type cuple struct {
    first  int
    second int
}

func (c *cuple) exchange() {
    c.first, c.second = c.second, c.first
}

func main() {
    var c cuple = cuple{first: 1, second: 2}
    fmt.Println(c)
    c.exchange()
    fmt.Println(c)
}
```

Programme 24 – methode2.go