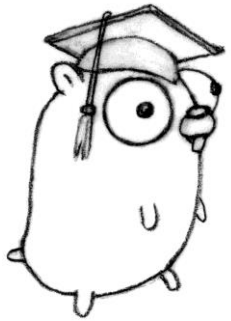
Introdução à Linguagem de Programação

Prof. José Cintra
http://www.josecintra.com/blog
Com algoritmos resolvidos no GITHUB

Revisão 1 – Julho/2020 – Quarto mês da quarentena



Apresentação

Este curso de **Introdução à Linguagem GO** é uma continuação do curso Algoritmos e & Lógica de programação.

Supõe-se, portanto, que o estudante já possua conhecimento **teórico** das estruturas básicas dos algoritmos, bem como noções sobre os tipos e estruturas de dados elementares como Arrays e Strings.

Para um melhor aproveitamento do curso, acesse os exercícios resolvidos no GITHUB e veja também nossa apostila do Curso de Programação C .

Temporada 1

Conceitos Básicos

Características da Linguagem

A linguagem GO foi Desenvolvida em 2007 por uma equipe dentro do Google liderada por Robert Griesemer, Rob Pike e Ken Thompson.



Principais características:

- Sintaxe C Like. Concisa e simples, porém rígida.
- Estaticamente compilada, mas com inferência de tipos!
- Coleta de lixo e concorrência robusta.
- Modular, com suporte para orientação a objetos mas com ênfase em interfaces.
- Ponteiros, sem aritmética.

O que foi deixado de lado:

- Classes!
- Herança!
- Overloading!
- Generics!
- Exceções

Instalação e Preparação do Ambiente

O primeiro passo é preparar o seu **workspace**, ou seja, as pastas de trabalho onde vão ficar seus arquivos de código-fonte, executáveis e etc.
Precisamos criar pelo menos três pastas que podem ficar dentro de qualquer outro diretório de sua preferência, por exemplo: uma pasta de nome **workspace**.

- **pkg:** Pasta onde ficam os pacotes e bibliotecas que você baixou ou criou para referência em seus programas.
- **src:** Seus arquivos de código-fonte.
- **bin:** Aqui estão os arquivos executáveis que você criou e instalou a partir da pasta src.

Agora vamos baixar e instalar o **GO** na pasta padrão (**C:\Go** em Windows ou **/usr/local/go** no Linux)**.**
Site oficial para download**:** https://golang.org/dl/
Instruções de instalação: https://golang.org/doc/install

Depois, vamos criar quatro variáveis de ambiente :

- **GOPATH:** Caminho para o seu **workspace**.
- **GOROOT:** Caminho onde o **GO** foi instalado.
- **GOBIN:** Caminho onde você quer seus executáveis.De preferência a suppasta **bin** do workspace.
- **PATH:** Aponta para os binários do GO, geralmente a subpasta **bin** do caminho onde o **GO** foi instalado.

Instalação e Preparação do Ambiente

Golang enfatiza o uso de sistemas de controle de versão, como o **GIT**. Você precisará para instalar pacotes de biblioteca de terceiros. Se você não tiver um instalado no seu sistema, consulte as instruções de instalação neste link.

O último passo é baixar uma **IDE**. Você pode escolher uma aqui nesse link.
Sugiro fortemente duas delas:

- Visual Studio Code com a extensão para a linguagem GO.
- LiteIDE: ambiente de desenvolvimento exclusivo para linguagem GO.
- Geany: Editor de Textos para programadores. Já vem configurado para compilar programas GO

Feita a instalação e a preparação do ambiente, o próximo passo é digitar nosso primeiro programa, o **Alô Mundo**, que é o primeiro programa para desenvolvemos quando vamos aprender uma nova linguagem. Esse programa apenas apresenta uma mensagem de boas vindas na tela, mas através dele podemos aprender conceitos iniciais muito importantes sobre a linguagem.

Em nossos estudos, sempre vamos fazer comparações com a linguagem C e suas derivadas para facilitar o aprendizado.

Alô Mundo!

```
1  package main
2  // Primeira aplicação GO
3  import “fmt”
4  func main() {
5      fmt.Println("Hello from GO!")
6  }
```

As chaves { } são usadas para marcar o início e fim dos blocos de comandos e devem ser indentadas (*) dessa forma

(*) Muitos dicionários preferem o termo “endentação”.

(1) Todo programa **GO** deve começar com a declaração do pacote ao qual ele pertence. A instrução **package main** e a função **main** da linha 4 dizem ao compilador que este é um programa executável. A função **main**, portanto, é o ponto de partida das aplicações. Quando executamos o programa, essa função é iniciada, e quando ela finaliza, o programa termina. Portanto, o identificador **main** para funções e pacotes possui um significado especial aqui.
(2) Os comentários em **GO** seguem o mesmo estilo da linguagem **C**.
(3) Nessa linha estamos importando o pacote **fmt** que contém funções de formatação de entrada e saída.
(4,5,6) Nessas linhas de código temos a função principal (**main**), que a funçãp de entrada. Neste caso, a função imprimirá a mensagem "Hello from GO!" usando o método **Println()** presente no pacote **fmt**.
Obs:

1. Ao contrário de C, não usamos o separador ponto e vírgula (;) no final dos comandos.
2. Assim como o C, GO é **sensitive case**. No entanto, maiúsculas e minúsculas possuem um significado especial na linguagem como veremos mais à frente.
3. A posição das chaves e a **endentação (*)**, assim como estão, são padrões que é prudente seguirmos.

Compilando e Executando

Nosso programa “Alô Mundo” que está em um arquivo **alo.go** foi colocado em uma subpasta **alo** da pasta **src** do **workspace**.
Se você planeja publicar seu módulo para que outros usem, o caminho do módulo deve ser um local a partir do qual as ferramentas GO possam baixaá-lo. Para ativar o rastreamento de dependência para seu código, vá para a pasta onde se encontra o arquivo **alo.go**, e execute o seguinte comando:

```
go mod init alo
```

Uma vez que **GO** é uma linguagem compilada, para executar nosso programa **Alô Mundo**, devemos compilá-lo para gerar um executável e, para isso, usamos a linha de comando:

```
go build hello.go
```

Ou, melhor ainda, para compilar e executar, digite:

```
go run hello.go
```

Outros utilitários da linha de comando, não abordados nesse curso, são:

- **go vet**: Verifica o código-fonte e exibe as linhas com comandos suspeitos, com problemas de lógica.
- **go get**: Instala pacotes de terceiros, comumente armazenados em repositório de códigos como o GitHub.
- **gofmt**: Formata o código-fonte.
- **goimports**: Verifica o código-fonte e corrige as linhas com imports perdidos ou necessários.

Pacotes e Importação

Para incentivar a modularização e a organização, cada programa **GO** deve fazer parte de um **package**. Um **package** é uma coleção de arquivos de código-fonte, localizados na mesma pasta, que são compilados em conjunto. Para declarar um arquivo de código-fonte como parte de um pacote, digitamos a seguinte declaração no cabeçalho do arquivo:

```
package packagename
```

Esta declaração do pacote deve ser a primeira linha de código. Desta forma, todas as funções, tipos e variáveis definidas no seu arquivo **GO** tornam-se parte desse pacote.
GO tem uma poderosa biblioteca nativa (**GO Standard Library**) que também é organizada em pacotes.

Algumas bibliotecas comuns:

Package	**Funcionalidades**
fmt	Formação de dados de entrada e saída
math	Funções matemáticas
strconv	Conversões
strings	Manipulação de Strings
io	Funções básicas de entrada e saída
net/http	Implementação de clientes e servidores http

Para usar uma dessas bibliotecas, usamos o comando de importação.

Assim:

```
import "fmt"
import “math"
```

Ou assim:

```
import (
  "fmt"
  m “math"
)
```

Obs: No segundo exemplo, usamos um apelido (**alias**) para o **package math**. Isso facilita as referências.

Tipos de Dados Básicos

O **tipo de dado** de uma variável estabelece uma faixa de valores que ela pode aceitar, bem como as operações passíveis de serem realizadas com esses valores. Em quase toda linguagem, os tipos são divididos em:

- **Primitivos ou básicos:** Não podem ser decompostos em valores mais simples, ou seja, são atômicos e, geralmente, possuem relação direta com a arquitetura do computador.
- **Compostos ou estruturados:** São tipos de dados formados à partir dos tipos básicos, podendo inclusive serem definidos pelo programador.

Os tipos de dados primitivos em **GO** subdividem-se em **Numéricos**, **Caracteres** e **Booleanos**.

Os tipos numéricos, por sua vez, podem ser **inteiros** ou de **ponto flutuante**, de acordo com a tabela exibida na página seguinte.

Os tipos caracteres em **GO** são dois: o **byte** (caracteres ASCII de 8 bits) e o **rune** (caracteres unicode de 32 bits). São, na verdade, tipos numéricos, como veremos a seguir.

GO disponibiliza ainda o tipo **bool** para representar valores lógicos que, diferentemente da linguagem C, não são tratados como tipos numéricos, aceitando apenas dois valores: **true** ou **false.**

Tipos de Dados Numéricos

Tipo de dado	Descrição
uint8	Inteiro de 8 bits sem sinal (0 a 255)
uint16	Inteiro de 16 bits sem sinal (0 a 65535)
uint32	Inteiro de 32 bits sem sinal (0 a 4294967295)
uint64	Inteiro de 64 bits sem sinal (0 a 18446744073709551615)
uint	Inteiro sem sinal dependente da plataforma
int8	Inteiro de 8 bits com sinal (-128 a 127)
int16	Inteiro de 16 bits com sinal (-32768 a 32767)
int32	Inteiro de 32 bits com sinal (-2147483648 a 2147483647)
int64	Inteiro de 64 bits com sinal (-9223372036854775808 a 9223372036854775807)
int	Inteiro com sinal dependente da plataforma
float32	Número de ponto flutuande de 32 bits padrão IEEE-754
float64	Número de ponto flutuande de 64 bits padrão IEEE-754

Variáveis - Declaração e Atribuição

Como sabemos, uma **variável** é o nome dado a um local de memória para armazenar um valor de um tipo de dados específico.
A operação de **declaração** de uma variável consiste em definir seu **nome** e **tipo de dados**. Já, a operação de **atribuição** consiste em armazenar na variável (posição de memória) um valor compatível com seu tipo de dado.

Como a linguagem GO é estática e fortemente tipada, todas as variáveis precisam ser declaradas antes de serem usadas, seja explicitamente ou de forma que o compilador possa **inferir** seu tipo.

A operação de **inicialização** consiste em atribuir um valor inicial para a variável logo depois que ela foi declarada.
Um ponto interessante em **GO** é que **todas** as variáveis são **automaticamente inicializadas** na declaração: variáveis numéricas e caracteres com 0 (zero) e booleanas com **false**.

O operador de **atribuição** padrão é o sinal de igual (**=**). **GO** possui ainda um operador de declaração e inicialização (**:=**) que veremos a seguir.

A sintaxe geral de declaração/inicialização de variáveis é:

```
[var] nome_variável [tipo_dados] [= | := <valor>]
```

Obs: Se você declarar uma variável e não utilizá-la no código, isso causará um erro de compilação.

Variáveis - Declaração e Atribuição

Exemplos de declarações e atribuições:

```
// Aqui declaramos uma variável de nome “idade” do tipo int. Seu valor será zero.
var idade int
```

```
// Neste outro exemplo, declaramos a variável e depois atribuímos o valor 18 para ela.
var idade int
idade = 18
```

```
// Agora, declaração e atribuição no mesmo comando
var idade int = 18
```

```
// Várias variáveis
var (
    idade int     = 18
    peso  float64 = 85.5
)
```

Obs: GO desencoraja o uso de **underscore** ou **hífen** para nomes de variáveis. Por exemplo, uma variável nomeada como **peso_ideal**, deverá ser renomeada para **pesoIdeal** ou **PesoIdeal**.

Variáveis – Inferência de Tipos

Podemos declarar uma variável sem especificar seu tipo de dado. Este será **inferido** pelo compilador de acordo com o valor atribuído:

```
// Como o valor atribuído NÃO possui parte decimal, a variável será declarada como “int”.
var idade = 30
```

```
// Como o valor atribuído POSSUI parte decimal, a variável será declarada como “float64”.
var peso = 73.7
```

GO possui um operador de declaração (**:=**) por **inferência de tipos** conhecida como **short hand declaration**. Essa forma de declaração não necessita da palavra reservada **var** e é a preferida dos **Gophers**.

```
// A variável é declarada como “int” com valor inicial de 30
idade := 30
```

```
// Várias variáveis
// Nesse caso, a variável “idade” será do tipo “int” e a variável “peso” do tipo “float64”
idade, peso := 30, 70.7
```

Constantes

Uma **constante** é um valor armazenado na memória que não pode ser alterado, mantendo-se fixo durante todo o tempo de execução do programa. Podem ser constantes **nomeadas** ou **literais**.

Uma constante **nomeada** é como uma variável, pois recebe um nome, um tipo e uma posição na memória. A diferença é que não podem ser alteradas pelos comandos de atribuição, ou seja, não podem estar do lado esquerdo da atribuição (left-side value).

Para declararmos uma constante, usamos a palavra reservada **const**:

```
const masculino byte = ‘F’
```

O tipo da constante pode ser omitido na declaração, pois pode ser inferido, porém não podemos usar a declaração curta (:=)
Já, uma **constante literal** é aquela que é especificada diretamente no código, sem nome.
Por exemplo: No comando abaixo, os valores **30** e **70.7** são constantes literais e seu tipos são inferenciados pelo compilador.

```
idade, peso := 30, 70.7
```

Operações Aritméticas e Conversões

As operações matemáticas em **GO** não diferem muito das outras linguagens, a não ser em relação à conversão de tipos nas expressões. Um **cuidado especial** devem ter os programadores que estão acostumados com outras linguagens que efetuam o **cast automático** de tipos nas expressões. Em **GO** essas conversões devem ser feitas de forma manual. Vejam um exemplo:

Errado:

```
var a float64 = 1000
var b int = 10
var soma float64 = a + b
fmt.Println(soma)
```

Correto:

```
var a float64 = 1000
var b int = 10
var soma float64 = a + float64(b)
fmt.Println(soma)
```

Para fazer as conversões, use as funções apropriada para cada tipo, como **uint(), float64**(), **float32**(), etc.

Operadores Aritméticos e de Atribuição

Segue um resumo dos operadores aritméticos e de atribuição mais utilizados em GO:

Operadores de atribuição

Operador	Descrição
x = y	Atribuição
x := y	Declaração e atribuição
x += y	Adiciona e atribui
x -= y	Subtrai e atribui
x *= y	Multiplica e atribui
x /= y	Divide e atribui
x++	Atribui e incrementa
++x	Incrementa e atribui
x--	Atribui e subtrai
--x	Subtrai e atribui

Operadores aritméticos

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão
++	Incremento
--	Decremento

Ponteiros

Ponteiros são variáveis que armazenam **endereços de memória**.
Assim como qualquer variável, um ponteiro precisa ser declarado e deve apontar para valores do mesmo tipo de dado para o qual foi criado.
Dessa forma, podemos guardar em uma variável ponteiro o endereço de memória de outra variável e, assim, acessar diretamente o seu valor.
Golang disponibiliza dois operadores para trabalharmos com ponteiros e endereços de memória:

Operador	Nome	Descrição
&	Referência	Retorna o endereço de memória de uma variável
*	Derreferência	Retorna o conteúdo de um endereço de memória

Para declararmos uma variável ponteiro, usamos também o operador '*'.
Vejamos um exemplo:

```
var a int = 7 // a é variável int = 7
var b *int    // b é um ponteiro para int
b = &a        // b recebe o endereço de a

fmt.Println(a)  // Imprime 7
fmt.Println(&a) // Imprime 0xc0000100a0
fmt.Println(b)  // Imprime 0xc0000100a0
fmt.Println(*b) // Imprime 7

*b++ // Incrementa o valor apontado por b

fmt.Println(a)  // Imprime 8
```

Obs: Os endereços que aparecem acima são na minha máquina, naquele momento. Na sua máquina vão aparecer outros endereços.
Ponteiros são elementos importantes da linguagem GO e você vai ter que usá-los. Por isso reservamos um curso à parte sobre eles. Voltaremos a falar sobre endereços de memória na temporada sobre funções.

Funções de Saída de Dados

Vamos abordar agora as funções para impressão de dados na tela. Para imprimir expressões envolvendo variáveis e constantes, o pacote **fmt** disponibiliza as seguintes funções:

- **Print()** → Usada para imprimir uma sequência de expressões separadas por vírgulas. Para cada expressão será usado um formato padrão.
- **Println()** → Faz a mesma coisa que print(), no entanto pula de linha após a impressão.
- **Printf()** → Imprime uma expressão usando um **template** com **códigos de formatação (verbos)**.

Os códigos de formatação usados pelo **printf()** são parecidos com os da linguagem C. Vejamos alguns:

Código	Descrição
%v	Formatador genérico. Usa inferência de tipos
%d	Valor inteiro na base 10
%f	Valor em Ponto flutuante
%c	Tipos caractere
%s	Uma string
%t	Valor booleano: true ou false
%T	Mostra o tipo de dados da variável

Funções de Saída de Dados - Exemplos

```
package main

import "fmt"

func main() {
  var (
    peso   float64 = 75.0
    idade  int     = 18
    altura float64 = 1.76
  )
  fmt.Println("Olá!")
  fmt.Println("Sua idade é igual a: ", idade)
  fmt.Printf("Vc apresentou um peso de %f quilos e uma altura de %f metros", peso, altura)
}
```

Irá imprimir:

```
Olá!
Sua idade é igual a:  18
Vc apresentou um peso de 75.000000 quilos e uma altura de 1.760000 metros
```

Funções de Entrada de Dados

Para finalizar essa parte introdutória do curso, vamos abordar agora as funções para leitura de dados através das quais o usuário informa, pelo teclado, os dados de entrada requeridos pelas aplicações. Para essa finalidade, o pacote **fmt** disponibiliza as seguintes funções:

- **Scan()** → Usada para ler os dados vindos do dispositivo de entrada padrão (teclado) e armazenar na memória. O formato dos dados será inferido pelo tipo da variável.
- **Scanf()** → Semelhante ao Scan, porém o formato é definido pelo programador através de **códigos de formatação (verbos)**.

Os códigos de formatação usados pelo **Scanf()** são parecidos com os da linguagem C. Vejamos alguns:

Código	Descrição
%v	Formatador genérico. Usa inferência de tipos
%d	Valor inteiro na base 10
%f	Valor em Ponto flutuante
%c	Tipos caractere
%s	Uma string
%t	Valor booleano: true ou false
%T	Mostra o tipo de dados da variável

Funções de Entrada de Dados - Exemplos

```
package main
import "fmt"
func main() {
  idade, peso, altura := 0, 0.0, 0.0
  fmt.Print("Informe a idade: ")
  fmt.Scanf("%v", &idade) // Veja que passamos o endereço da variável
  fmt.Print("Informe o peso: ")
  fmt.Scanf("%v", &peso)
  fmt.Print("Informe a altura: ")
  fmt.Scanf("%v", &altura)
  fmt.Printf("Olá!\nSua idade é igual a %v\n: ", idade)
  fmt.Printf("Vc apresentou um peso de %g quilos e uma altura de %g metros", peso, altura)
}
```

Irá imprimir:

```
Informe a idade: 30
Informe o peso: 75
Informe a altura: 1.77
Olá!
Sua idade é igual a:  30
Vc apresentou um peso de 75 quilos e uma altura de 1.77 metros
```

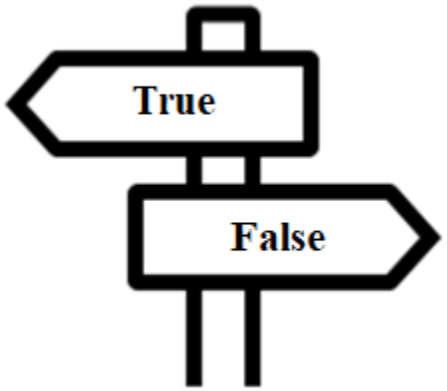
Fim da Temporada 1: Consolidando o Conhecimento

Exercício: Ler duas notas e calcular a média aritmética entre elas.

```
// media.go:

package main
import "fmt“

func main() {
  // Declaração de variáveis
  var nota1, nota2, media float64

  // Leitura das variáveis informadas pelo usuário
  fmt.Print("Informe a nota 1: ")
  fmt.Scan(&nota1)
  fmt.Print("Informe a nota 2: ")
  fmt.Scan(&nota2)

  // Processamento e exibição do resultado
  media = (nota1 + nota2) / 2.0
  fmt.Printf("A média entre %f e %f é %f", nota1, nota2, media)
}
```

Baixe este e outros exercícios resolvidos no GITHUB

Temporada 2



Estruturas de Decisão

Estruturas de Decisão

As estruturas de decisão permitem dividir o fluxo de execução das instruções entre dois ou mais caminhos diferentes, permitindo realizar tomadas de decisão de acordo com a análise de uma expressão lógica (true/false).

GOLANG possui três tipos de estruturas condicionais:

1) if - else → Estrutura de decisão tradicional que permite escolher entre dois fluxos alternativos.

```
if  condição {
  // trecho executado se a condição é verdadeira
} else {
  // trecho executado se a condição é falsa
}
```

2) if - else if → Estrutura de decisão alternativa quando temos mais de duas possibilidades de fluxo.

→

```
if  condição1 {
  // Trecho executado se condição1 é verdadeira
} else if condição2 {
  // Trecho executado se condição2 é verdadeira
} ... else {
  // Trecho executado se todas condições
  // anteriores forem falsas}
}
```

3) switch → Estrutura de decisão especializada em múltiplas alternativas de fluxo, não necessariamente através de expressões lógicas.

```
switch [expressão] {
case expressão1: Trecho de comandos
case expressão2: Trecho de comandos
...
default: Trecho de comandos
}
```

Vamos ver agora exemplos de cada uma →

Estruturas de Decisão - Exemplos

1) Testa se número é par usando **if - else**:

```
var numero int = 1
if numero%2 == 0 {
  fmt.Printf("O número %d é par", numero)
} else {
  fmt.Printf("O número %d é ímpar", numero)
}
```

2) Testa positividade do número com **if - else aninhado**:

```
var numero int = 1
if numero > 0 {
  fmt.Printf("O número %d é positivo", numero)
} else {
  if numero < 0 {
    fmt.Printf("O número %d é negativo", numero)
  } else {
    fmt.Printf("O número %d é neutro", numero)
  }
}
```

3) Testa positividade do número com **if - else if**:

```
var numero int = 1
if numero > 0 {
  fmt.Printf("O número %d é positivo", numero)
} else if numero < 0 {
  fmt.Printf("O número %d é negativo", numero)
} else {
  fmt.Printf("O número %d é neutro", numero)
}
```

Obs:

1. A condição avaliada não precisa ficar entre parênteses
2. As chaves são **obrigatórias.**
3. A **posição** das chaves, "indentação" e **posição** do **else** como estão são importantes e podem causar erros de compilação
4. A cláusula **else** é opcional .

Estruturas de Decisão - Exemplos

4) Testa positividade do número com **switch**:

```
numero := -1
switch {
  case numero < 0:
    fmt.Printf("O número %d é negativo", numero)
  case numero > 0:
    fmt.Printf("O número %d é positivo", numero)
  default:
    fmt.Printf("O número %d é neutro", numero)
}
```

5) Informa o dia da semana com switch:

```
diaSemana := 6
switch diaSemana {
case 0:
  fmt.Println("Hoje é domingo")
case 6:
  fmt.Println("Hoje é Sábado")
case 1, 2, 3, 4, 5:
  fmt.Println("Hoje é dia de semana")
default:
  fmt.Println("Dia inválido")
}
```

Obs:

1. A estrutura **switch** em **GO** é mais poderosa e flexível em relação a outras linguagens .
2. Podemos testar expressões lógicas como no exemplo 4.
3. O comando break não é necessário.

Fim da Temporada 2 – Consolidando o Conhecimento

Exercício: Dadas as dimensões dos três lados A,B,C de um triângulo, desenvolva um algoritmo para determinar o tipo do triângulo de acordo com seus lados: Equilátero, Isósceles ou Escaleno. Obs: Em todo triângulo, cada um dos lados deve ser menor que a soma dos outros dois.

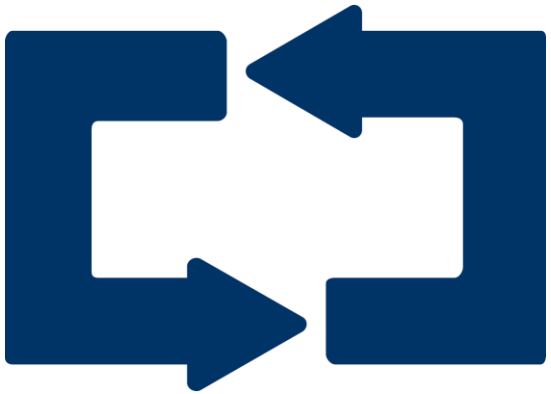
```
// triangulo.go

package main

import "fmt"

func main() {

  // Declaração das variáveis (float64)
  A, B, C := 0.0, 0.0, 0.0

  // Leitura dos dados
  fmt.Print("Informe o lado A: ")
  fmt.Scan(&A)
  fmt.Print("Informe o lado B: ")
  fmt.Scan(&B)
  fmt.Print("Informe o lado C: ")
  fmt.Scan(&C)
```

→

```
  // Teste e exibição dos resultados
  if !(A < (B+C) && B < (A+C) && C < (A+B)) {
    fmt.Println(“Lados inválidos")
  } else {
    switch {
    case A == B && B == C:
      fmt.Println("Triângulo Equilátero")
    case A == B || A == C || B == C:
      fmt.Println("Triângulo Isósceles")
    default:
      fmt.Println("Triângulo Escaleno")
    } // Fim switch
  } // Fim if
} // Fim main
```

Baixe este e outros exercícios resolvidos no GITHUB

Temporada 3



Estruturas de Repetição

Estruturas de Repetição

As estruturas de repetições são construções que permitem a execução repetitiva de um bloco de códigos de acordo com a avaliação de uma condição (expressão lógica). A repetição ocorrerá enquanto essa condição for verdadeira (**condição de parada**).
Curiosa e acertadamente, a linguagem **GO** possui apenas uma estrutura de repetição: O laço **for** que é muito flexível e poderoso, podendo realizar qualquer tipo de repetição das linguagens tradicionais.

```
for [pré; ] [condição;] [pós] {
...
[break]
[continue]
...
}
```

Onde:

- **pré** → Comando executado antes da primeira iteração.
- **condição** → Expressão lógica que representa a condição de parada, avaliada a cada iteração.
- **pós** → Comando executado no fim de cada iteração.
- **break /continue** → Usados para, respectivamente, interromper ou saltar a execução do laço de repetição.

É possível também fazer iterações em coleções de dados como **slices** e **strings**, como veremos na temporada sobre estruturas de dados.

Estruturas de Repetição - Exemplos

1) Laço **for** simples, exibindo os número de 1 até 10:

```
for i := 1; i <= 10; i++ {
    fmt.Println(i)
}
```

2) Laço **while** simples, exibindo os número de 1 até 10:

```
i := 1
for i <= 10 {
 fmt.Println(i)
 i++
}
```

3) Contagem de 1 a 5 com **break**:

```
for i := 1; i <= 10; i++ {
    fmt.Println(i)
    if i == 5 {
      break
    }
}
```

4) Contagem de 6 a 10 com **continue**:

```
for i := 1; i <= 10; i++ {
  if i < 6 {
    continue
  }
  fmt.Println(i)
}
```

Obs:

1. As chaves são obrigatórias e na posição em que estão.
2. Parênteses envolvendo os componentes não são necessários
3. Existe uma maneira mais fácil de iterar sobre coleções de dados, como **arrays** e **slices**: o operador **range** que veremos a seguir.

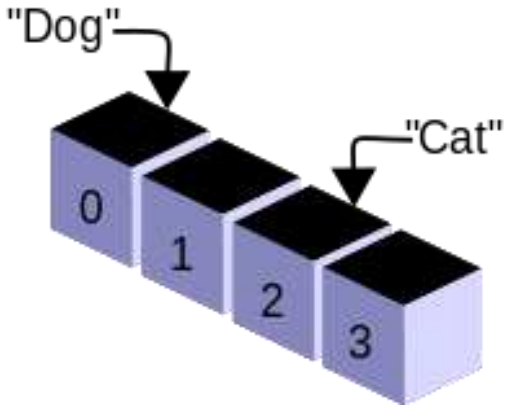
Fim da Temporada 3 – Consolidando o Conhecimento

Exercício: Imprima as temperaturas em graus Celsius, Kelvin e Fahrenheit variando na faixa de -100ºC até 100ºC, sabendo que: $\frac{T_c}{5} = \frac{T_F - 32}{9} = \frac{T_K - 273}{5}$

```
package main

import "fmt"

func main() {
  // Declaração das variáveis para Celsius, Fahreinheit e Kelvin
  var C, F, K float64

  // Repetição de -100 C até 100 C
  for C = -100; C <= 100; C++ {
    F = C*9.0/5.0 + 32.0 // Cálculo de Fahreinheit
    K = C + 273.15  // Cálculo de Kelvin
    fmt.Printf("%f C = %f F = %f K\n", C, F, K) // Impressão
  }

}
```

Baixe este e outros exercícios resolvidos no GITHUB

Temporada 4



Estruturas de Dados Básicas

Estruturas de Dados Compostas

As estruturas de dados compostas são os tipos de dados formados à partir dos tipos primitivos. Neste curso introdutório, vamos estudar as seguintes estruturas disponibilizadas nativamente pela linguagem GO:

- **Strings** → Uma **string** em **GO** possui as mesmas características abstratas de outras linguagens, mas sua implementação é bem diferente. Em **GO**, uma string é uma cadeia **imutável** de caracteres (bytes) representados usando a codificação **UTF-8**.

- **Arrays** → Os **arrays** em **GO** possuem as mesmas características abstratas de linguagens como **C** e **Java**, ou seja, uma coleção com tamanho fixo de variáveis do mesmo tipo e que são acessadas através de índices.

- **Slices** → Um **slice** é uma abstração de um **array**. Como o nome diz, uma **fatia**, os seja, uma **view** para um **array**. Assim como um **array,** um **slice** é indexado e possui um tamanho, mas sua vantagem é que podemos mudar esse tamanho com facilidade.

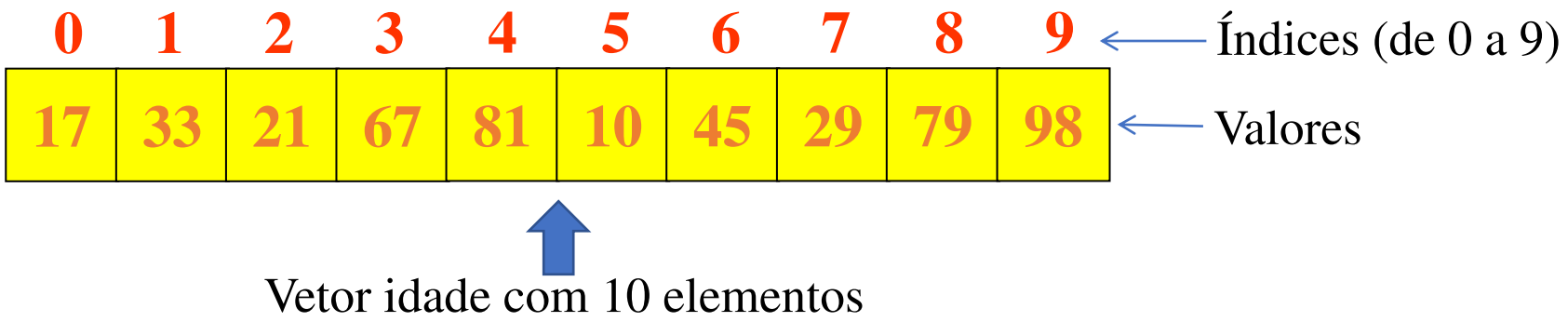
- **Maps** → Equivalem aos **arrays associativos**, onde temos uma coleção de pares "chave → valor"

- **Structs** → Uma **struct** é uma estrutura de dados heterogênea. Abstratamente é um **registro** composto de **campos** de diferentes tipos de dados, mas logicamente relacionados.

Arrays

Um **array unidimensional** (vetor) é uma estrutura **homogênea** de dados que armazena uma **sequência de variáveis** (elementos), todas do **mesmo tipo** acessados individualmente através de um número sequencial que denominamos **índice do array**. Ou seja, cada elemento do vetor é identificado por sua posição dentro do vetor.

- A quantidade de elementos (tamanho do array) e seu tipo devem ser definidas no momento da declaração e são imutáveis;
- Os elementos possuem **ordinalidade**, cada um pode ser identificado pela sua posição (**índice do vetor**);
- Em GO, os índices dos vetores começam em 0 (zero) e vão até (tamanho-1)
- Cada elemento do vetor, por meio do seu índice, pode ser acessado como uma variável individual.
- No array abaixo de nome "idade", o valor do elemento de índice 4 é 81.



Operações com Arrays

Declaração de Arrays

Para declararmos um **array**, precisamos informar seu nome, tamanho e tipo de dado. Exemplo:

```
var nota[3] float64
```

Essa declaração vai criar um **array** de nome **nota** com 3 elementos do tipo **int,** indexados de 0 a 2. Todos os elementos vão ser preenchidos com o valor 0.0.
Outra forma de declarar um **array**, é atribuir valores para seus elementos já na criação:

```
var nota = [35] float64{5,6.5,7}
```

Atribuindo valores para os elementos de um array

Devemos especificar o elemento pelo seu índice:

```
Nota[0] = 5
Nota[1] = 6.5
Nota[2] = 7
```

Representação didática do **array** de nome **nota**:

5	6.5	7	← Elementos
0	1	2	← Índices

Podemos também atribuir valores através da leitura pela função **Scan**(), sempre através do índice:

```
fmt.Print("Digite o valor para o vetor: ")
fmt.Scan(nota[1])
```

Acessando os elementos de um array

Sempre através do índice:

```
fmt.Println(nota[1])
```

Alternativamente, podemos imprimir todo o vetor, dessa forma:

```
fmt.Println(nota)
```

O Operador Range e o Identificador Blank

Para iterarmos sobre os elementos de um **array** podemos usar a estrutura de repetição **for** da forma tradicional:

```
for i:= 0, i < len(nota); i++ {
  fmt.Printf("indice = %d e Valor = %f",i,nota[i])
}
```

Onde: **len** é a função que devolve o tamanho do vetor, **i** é o índice do vetor e **nota[i]** é o conteúdo de cada elemento.

No entanto, em **GO**, existe uma forma mais fácil, usando o operador **range** que devolve, a cada iteração do **for**, o **índice** e o **valor** do **array** (conteúdo):

```
for i,v := range nota {
  fmt.Printf("Indice = %d e Valor = %f",i,v)
}
```

Caso alguns desses valores de retorno não sejam necessários, podemos ignorá-los através do identificador **blank**, o **underscore**. Por exemplo, caso o índice do elemento não seja necessário, podemos refazer a repetição anterior:

```
for _,v := range nota {
  fmt.Printf("Valor = %f",v)
}
```

Na verdade, o identificador **blank** é muito utilizado em **GO**, como veremos na temporada sobre **funções**.

Arrays - Exemplo

Exercício: Leia 10 notas de um aluno, armazene em um array e calcule a média Geral. Logo após, relacione as notas que estão acima da média.

```
package main
import "fmt"

func main() {
  const numNotas int = 10 // Quantidade de notas
  var (
      nota        [numNotas]float64 // Arrqy de notas
      soma, media float64 = 0, 0
  )

  // Parte 1 – Leitura
  for i := 0; i < 10; i++ {
    fmt.Printf("Informe a nota %d: ", i)
    fmt.Scan(&nota[i]) // Armazena nota no array
    soma += nota[i] // Calcula a soma das notas
  }

  // Parte 2 – Cálculo da média
  media = soma / float64(numNotas) // Cálculo da média
  fmt.Printf("Média = %f\n", media)
```

→

```
  // Parte 3 - Notas maiores que a média
  fmt.Println("Notas Maiores do que a média:")
  for i, v := range nota {
    if v > media {
      fmt.Printf("Nota %d = %f\n", i, v)
    }
  }
}
```

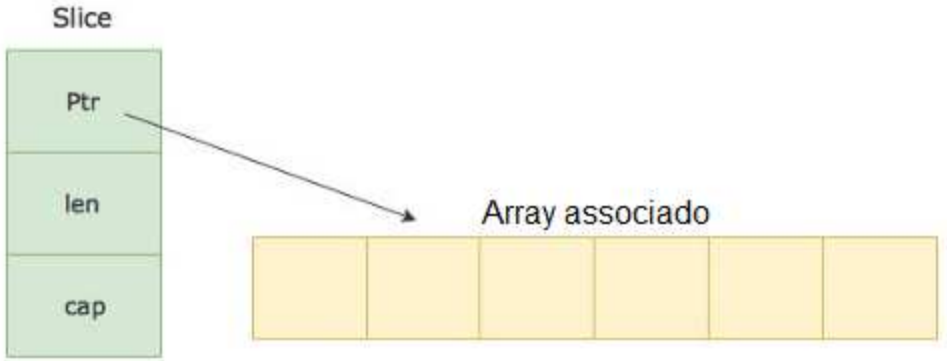
Veja que, nesse exemplo, usamos operador **range** para iterar entre os elementos do vetor.
Quando iteramos uma coleção usando **range**, dois valores são retornados para cada iteração:
O primeiro é o índice (i) e o segundo é uma cópia do valor do elemento desse índice (v).

Slices

Um **slice** é uma estrutura que aponta para um **array**, permitindo manipular os seus elementos e o seu tamanho. Na verdade, um **slice** não armazena nenhum valor: sempre haverá um **array associado** onde os elementos estarão armazenados.

As seguintes propriedades estão associadas a um slice:

- **ptr** → Um ponteiro para o **array** associado.
- **len** → Tamanho do **slice** (número de elementos).
- **cap** → Capacidade do **slice** (tamanho máximo).



Depois de criado o **slice**, podemos manipular seus elementos da mesma forma como fazemos com os **arrays** (através dos índices), com a vantagem de que **slices** possuem funções para adicionar elementos, alterando seus tamanhos.

Operações com Slices

Declaração

Podemos usar a função **make()**, passando o tipo de dado, o tamanho e a capacidade do **slice**:

```
nota := make([]float64,3,10)
```

Essa operação vai criar um **slice** de nome **nota**, com 3 elementos numerados de 0 a 2, mas com **capacidade** de crescimento para 10 elementos.
Para obter o tamanho e capacidade de um **slice**, usamos as funções **len()** e **cap()**, respectivamente.

Podemos também criar um **slice** informando seus elementos:

```
nota := []float64(5,6.5,7)
```

Para atribuição dos valores, impressão e leitura de um **slice**, procedemos da mesma forma como fazemos com **arrays**.

Adicionando elementos ao slice

Podemos usar a função **append**, passando o **slice** e os elementos a serem adicionados:

```
a := make([]int, 0,10)
a = append(a, 1, 2, 3)
```

Nesse exemplo, criamos um **slice** vazio e adicionamos três elementos, que podem ser acessados pelos seus índices de 0 a 2.

Percorrendo um slice com range

Podemos usar uma repetição para iterar sobre os elementos do **slice**:

```
var s = []int{1, 2, 4, 8, 16, 32, 64, 128}
for indice, valor := range s {
  fmt.Printf("Índice = %d\n”, indice)
  fmt.Printf("Valor = %d\n", valor)
}
```

Slices - Exemplo

Exercício: Leia um número indeterminado de números inteiros e armazene os pares em um **slice** de nome **par** e os ímpares em um **slice** de nome **impar**. A leitura deverá parar quando for digitado um número negativo.

```
package main
import "fmt"

func main() {
  var par, impar []int // Slices
  var numero int
  // Parte 1 – Leitura e armazenamento nos slices
  for {
    fmt.Print("Dígite um número: ")
    fmt.Scan(&numero)
    if numero < 0 {  // Fim da leitura
      break
    }
    if numero%2 == 0 {  // Slice par
      par = append(par, numero)
    } else { // Slice impar
      impar = append(impar, numero)
    }
  }
```

```
  // Parte 2 - Notas maiores que a média
  fmt.Println("Números Pares:")
  for _, valor := range par {
    fmt.Println(valor)
  }
  fmt.Println("Números Ímpares:")
  for _, valor := range impar {
    fmt.Println(valor)
  }
}
```

Veja no **range** que ignoramos o índice e usamos só o valor (v) do slice. Para isso, usamos o operador **_**.

Strings

De forma abstrata, uma **string** é um tipo de dado que representa um texto, uma palavra ou frase qualquer, ou seja, um conjunto de caracteres.
Em **GO**, o tipo de dados **string** é uma sequência de caracteres **unicode** com a codificação **UTF-8**.

Uma **string** possui um tamanho associado e cada caractere pode ser obtido pela sua posição dentro da **string**. Dessa forma a palavra "GOLANG" pode ser representada didaticamente, como:

G	O	L	A	N	G
0	1	2	3	4	5

← string
← posição dos caracteres

Onde, por exemplo, a letra 'L' está na posição 2.

Internamente as **strings** são tratadas como **slices** de **bytes** e, para sua manipulação, devemos importar o **package** 'strings' da biblioteca padrão.

Para declararmos e inicializarmos uma **string**, usamos o separador de aspas duplas:

```
var s string
s = "GOLANG"
```

ou em um comando só:

```
var s string = "GOLANG"
```

Operações com Strings

Função **len()** para obter o tamanho:

```
s := “PYTHON”
fmt.Prinln(len(s))
```

Vai imprimir: 6

Para obter o caractere em uma posição:

```
s := “PYTHON”
fmt.Printf("%c", s[1])
```

Vai imprimir: Y

Para obter uma substring:

```
s := “GOLANG”
fmt.Println(s[2:5])
```

Vai imprimir: LANG

Para concatenar strings:

```
s := "GOLANG"
S1 = "PYTHON"
fmt.Println(s + " e " + s1)
```

Vai imprimir: GOLANG e PYTHON

Obter a posição de uma substring:

```
s := "GOLANG"
fmt.Println(strings.Index(s,“L")
```

Vai imprimir: 2

Verifica se a **string** contém uma **substring**:

```
s := "GOLANG"
fmt.Println(strings.Constains(s,“GO")
```

Vai imprimir: true

Consulte a documentação do **package strings** para conhecer mais funções.

Obs: Para leitura de **strings** com espaços, o pacote **bufio** contém métodos mais apropriados, como o **readString** que veremos no exercício seguinte.

Strings - Exemplo

Exercício: Leia o nome de uma pessoa e abrevie da seguinte forma: Primeira letra do nome seguida de ponto(.) mais o **último** sobrenome. Exemplo: **José Luiz de Almeida Soares** deve abreviar como: **J.Soares**

```
package main

import (
  "bufio"
  f "fmt“ // f é um alias para o package "fmt"
  "os“
  s "strings“ // s é um alias para “strings"
)

func main() {
  // Parte 1 – Declaração e leitura das variáveis
  var (
    nome, sobreNome string
    nomeAbreviado, primeiraLetra string
    ultimoEspaco int
  )
  f.Println("Digite o nome: ")
  scan := bufio.NewReader(os.Stdin)
  nome, _ = scan.ReadString('\n’)
```

→

```
  // Parte 2 – Calcula o nome abreviado
  if !s.Contains(nome, " ") { // Nome sem sobrenome
    nomeAbreviado = string(nome)
  } else {
    primeiraLetra = string(nome[0])
    ultimoEspaco = s.LastIndex(nome, " ") + 1
    sobreNome = nome[ultimoEspaco:]
    nomeAbreviado = primeiraLetra + "." + sobreNome
  }
  f.Println("Nome abreviado = ", nomeAbreviado)
}
```

Maps

A estrutura **map**, também conhecida como **array associativo** ou **dicionário**, é uma coleção não ordenada de pares chave → valor (**key-value**) onde a chave não se repete. Pode-se fazer analogia remota com um **array**, no qual os índices podem ser de qualquer tipo.
Vamos supor que desejássemos estabelecer um correspondência entre os estados do Brasil e suas siglas. Poderíamos usar um **map** configurado da seguinte forma:

São Paulo	Minas Gerais	Rio de Janeiro	Espírito Santo
SP	MG	RJ	ES

← Valor (Value)
← Chave (Key)

Esse é um exemplo de um **map** onde a chave e seu valor são do tipo **string**.
Dessa forma temos, por exemplo, que à chave "MG" corresponde o valor "Minas Gerais", ambos valores do tipo **string**.

Em **GO** Podemos definir **maps** com chaves e valores de qualquer tipo. Além disso, a linguagem disponibiliza, nativamente, uma série de funções para adicionar, localizar, deletar e manipular elementos de um **map**.

→

Operações com Maps

Declaração

Podemos declarar e inicializar um **map** assim:

```
estado := make(map[string]string)
```

Aqui declaramos um **map** vazio com chaves e valores do tipo **string**

```
estado := map[string]string{
  “SP": “São Paulo”,
  “MG”: “Minas Gerais”
  “RJ": “Rio de Janeiro”,
  “ES”: “Espírito Santo”,
}
```

Aqui declaramos e atribuímos os pares. Repare a última vírgula: isso não causa erro, é um padrão em GO.

Atribuição de valores para as chaves

Para atribuirmos ou adicionarmos um novo par para o **map**, basta especificarmos a chave:

```
estado[“SC”] = “Santa Catarina”
```

Neste exemplo adicionamos um novo estado no **map**. Da mesma forma, podemos alterar qualquer valor pela sua chave.

Acessando um valor de uma chave

Basta indicarmos a chave:

```
x,f := estado[“SC”]
```

Neste exemplo, a variável **x** vai receber o valor “Santa Catarina” e a variável **f** o valor **true**

Para deletar uma chave:

```
delete(estado,”SC”)
```

Maps - Exemplo

Exercício: Leia uma quantidade indefinida de nomes de alunos e notas. Armazene-os em uma **map** com a chave **nome → nota**. A leitura termina quando digitar o nome "fim". Imprimir os alunos com nota maior que a média.

```
package main
import (
  "fmt"
  s "strings"
)

func main() {
  mapNota := make(map[string]float64)
  var nota, soma, media float64 = 0, 0, 0
  var nome string = "nome"
  var i int = 0
  for {
    fmt.Print("Informe o nome: ")
    fmt.Scanln(&nome)
    if s.ToUpper(nome) == "FIM" {
      break
    }
```

→

```
    if _, achou := mapNota[nome]; achou {
      fmt.Println("Nome repetido! Redigite...")
      continue
    }

    fmt.Print("Informe a nota: ")
    fmt.Scanln(&nota)
    mapNota[nome] = nota
    i++
    soma += nota
  } // Fim for
  media = soma / float64(i)
  fmt.Println("Média = ", media)
  for i, v := range mapNota {
    if v > media {
      fmt.Println(i, v)
    }
  }
}
```

Structs

Uma **struct** é um tipo de dado composto heterogêneo definido pelo usuário composto por um conjunto de variáveis, não necessariamente do mesmo tipo, denominados **campos**.

Podemos usar uma **struct** para definir estruturas complexas como, por exemplo, uma ficha de um funcionário ou um produto de comércio eletrônico.
Representam um papel importante na linguagem, pois são a base para construções **orientada a objetos**, como veremos em outro curso.

Além disso, o uso de **structs** em conjunto com **slices**, propiciam ferramentas para a construção de algoritmos modulares, flexíveis e legíveis.

Operações com Structs

Declaração

Exemplo de uma **struct** para representar uma pessoa:

```
type struct regPessoa {
  nome string
  peso float64
  altura float64
}
```

Para declarar uma variável do tipo **regPessoa**, pode-se usar a função **new()**:

```
pessoa := new(regPessoa)
```

Ou podemos instanciar e inicializar:

```
pessoa := regPessoa{
nome:   "José",
peso:   73,
altura: 1.76,
}
```

Acessando os campos

Usamos o separador "." para acessar os elementos individuais:

```
pessoa.nome = "José"
pessoa.peso = 73
pessoa.altura = 1.76
```

Ou para imprimir:

```
fmt.PrintLn(pessoa.nome)
fmt.PrintLn(pessoa.peso)
fmt.PrintLn(pessoa.altura)
```

Fim da Temporada 4 – Consolidando o Conhecimento

Exercício: Crie uma **struct** para armazenar o nome, altura e peso ideal de uma pessoa. Depois crie um **slice** e preencha seus elementos com **structs** do tipo pessoa. Leia os dados com uma repetição até ser digitado um nome igual a "fim". Calcule o peso ideal com base na fórmula: Peso ideal = 72.7 x altura - 58.0. Imprima o slice

```
package main

import (
  f "fmt"
  s "strings"
)

type regPessoa struct {
  nome      string
  altura    float64
  pesoIdeal float64
}

func main() {
  // Slice de regPessoa
  pessoas := make([]regPessoa, 0, 10)
  var nm string // Nome da pessoa
  // Altura e peso
  var al, pi float64
```

→

```
  Leitura, cálculo e impressão do slice de pessoas
  for {
    f.Println("Digite o nome da pessoa: ")
    f.Scanln(&nm)
    if s.ToUpper(nm) == "FIM" {
      break
    }
    f.Println("Digite a altura da pessoa: ")
    f.Scanln(&al)
    pi = 72.7*al - 58.0 // Cálculo do peso ideal
    pessoa := regPessoa{nome: nm, altura: al, pesoIdeal: pi}
    pessoas = append(pessoas, pessoa)
  }
  for i, v := range pessoas {
    f.Println(i, v)
  }
}
```

Baixe este e outros exercícios resolvidos no GITHUB

Temporada 5

Funções

Funções

Como vimos na apostila de algoritmos, uma função é uma técnica de **modularização** que consiste em **encapsular** um bloco de código que possui uma determinada funcionalidade de tal modo que possa ser **reutilizado** por outros módulos do sistema.

Em outras palavras, uma função é um trecho de código identificado por um nome e dedicado a resolver uma tarefa específica, retornando ou não uma informação com base em alguns valores de entrada.

As funções em GO são **first-class citizens**, ou seja, elas podem ser atribuídas a variáveis, passadas como argumento, imediatamente invocadas ou adiadas para execução.

Outras características:

- Pode, opcionalmente, receber parâmetros de entrada de diversos tipos.
- Retorna, opcionalmente, um ou **mais** valores de diversos tipos.
- Pode ser declarada dentro do próprio módulo que a chamou ou ficar em **packages** separados.
- Pode ser usada em expressões.
- Suporta valores de retorno nomeados, **variadic functions** e **funções anônimas**.

Funções

A declaração de uma função pode conter as seguintes partes

```
<func> nome_função ([lista_declaração_parâmetros]) [(lista_tipos_retorno)] {
  ...
  [return <lista_variáveis_retorno>)]
}
```

- A palavra reservado **func** indica que uma função vai ser declarada e é obrigatória
- O nome da função (**nome_função**) pode ser em minúsculas ou **Camel Case**. Evite **underscore**.
- A lista de parâmetros de entrada (**lista_declaração_parâmetros**) é uma lista de declaração de variáveis que serão usadas pela função e cujos valores serão fornecidos na chamada da função. São opcionais: podem haver funções que executam suas tarefas sem a necessidade de nenhuma informação externa.
- A lista de tipos de retorno (**lista_tipos_retorno**) é uma lista de tipos válidos dos valores que serão retornados pela função. É opcional: existem funções que não retornam valores. Não existe o tipo **void**, como em C.
- Se a função retornar algum valor, eles serão declarados através do comando **return** seguido dos valores de retorno.

Funções - Exemplo

Exercício: Crie uma função de nome **soma** que receba 2 valores e retorne a soma desses valores. No programa principal leia dois números e calcule sua soma através da função criada.

```
package main
import f "fmt"

func main() {
  // Declaração e leitura dos números a serem somados
  n1, n2 := 0, 0
  f.Print("Digite o primeiro número: ")
  f.Scan(&n1)
  f.Print("Digite o segundo número: ")
  f.Scan(&n2)

  // Cálculo da soma através da chamada à função
  s := soma(n1, n2) // Cálculo da soma através da função
  f.Printf("A soma entre %d e %d é %d", n1, n2, s)
}

// Definição da função soma
func soma(numero1, numero2 int) int {
  return numero1 + numero2
}
```

Aqui chamamos a função soma e passamos 2 argumentos que são os números lidos. Os argumentos **n1** e **n2** serão atribuídos aos **parâmetros numero1** e **numero2** da função definida abaixo e o retorno da função será atribuído à variável **s**

Aqui definimos a função **soma** que recebe 2 parâmetros como entrada e retorna um único valor que é a soma dos valores recebidos

Funções com Múltiplos Valores de Retorno

Uma característica marcante na linguagem **GO** é que as funções podem retornar mais de uma valor. Em linguagens como **Java** ou **C** isso não é possível, seria necessário retornar um vetor ou classe para essa finalidade.
Vejamos um exemplo: Em matemática, a divisão por zero não é permitida. Vamos fazer uma função que divida dois números e retorne dois valores: Um valor **float64** com o resultado da divisão e outro valor **booleano** indicando se houve erro na divisão, caso o divisor seja 0 (zero):

```
func divisao(numero1, numero2 float64) (bool, float64) {
  erro := false
  result := 0.0
  if numero2 == 0 {
    erro = true
  } else {
    result = numero1 / numero2
  }
  return erro, result
}
```

A chamada da função ficaria assim:

```
  e, r := divisao(n1, n2)
  if !e {
    f.Println(“Resultado = ",r)
  } else {
    f.Println(“Divisão não permitida")
  }
```

Veja que a função **divisao** retorna dois valores: **erro**, valor booleano indicando se a divisão é possível, e **result** com o resultado da divisão. Na chamada da função, atribuímos esses retornos para as variáveis **e** e **r**, respectivamente. Assim, testamos o valor do erro para exibirmos a mensagem apropriada.
Caso seja desejável ignorar um dos retornos, usamos o identificador **blank** (**underscor**e).
Exemplo:

```
_, r := divisao(n1, n2) // Nesse caso, ignoramos o retorno booleano de erro
```

Funções Variádicas e Funções como Variáveis

Funções variádicas: Mais conhecidas como **variadic functions** são funções que aceitam um número indeterminado de parâmetros. Na chamada da função pode-se passar os valores separados por vírgula ou através de um **slice**. Como exemplo, faremos uma função **soma** que faça uma soma de vários valores passados como argumentos:

```
func soma(nums ...int) int {
  result := 0
  for _, n := range nums {
    result += n
  }
  return result
}
```

Exemplos de chamada da função:

```
fmt.Println(Sum()) // Retorna 0
fmt.Println(Sum(5)) // Retorna 5
fmt.Println(Sum(2, 5, 1)) // Retorna 8
```

Veja que, para declararmos, colocamos **...** seguidos do tipo dos parâmetros.

Funções como variáveis: É possível atribuir uma função para uma variável e até passá-la como argumento para uma outra função:

```
f := func() {
  fmt.Println(“Alô Mundo!")
}
f() // Vai imprimir “Alô Mundo!”
```

Escopo de Variáveis e Tempo de Vida

Visibilidade dentro do pacote

O **escopo** ou **visibilidade de variáveis** tem a ver com o princípio do **ocultamento de informações** e permite que as variáveis só possam ser acessadas de dentro da função que as criou. Isso é mais seguro, pois evita que módulos externos interfiram na lógica das funções.
A linguagem **GO** possui os seguintes escopos de variáveis:

- **Global**: São as variáveis declaradas fora de qualquer bloco ou função e são acessíveis em todo o pacote onde foram declaradas. O tempo de vida dessas variáveis se encerra quando o programa se encerra.
- **Local**: São variáveis declaradas dentro de uma função ou bloco de código e são vistas somente na função/bloco em que foram declaradas. O tempo de vida dessas variáveis se encerra quando a função/bloco se encerra

Obs: O escopo **static** não existe em GO.

Visibilidade entre pacotes

Quando importamos um pacote, obtemos tem acesso a todas variáveis e funções desse pacote que possuam a **primeira letra maiúscula**. Repetindo: o critério para que uma função seja acessível (visível) entre os pacotes é que ela se inicie por uma **letra maiúscula**.
Repare as funções que estudamos até agora. Você não se perguntou porque usamos **frm.Printf** e não **fmt.printf**?
As funções que começam com **letra minúscula** são acessíveis somente no pacote e **não são exportadas**.

Escopo de Variáveis e Tempo de Vida

```
package main
import "fmt"

var v int = 7          // v é variável global vista em todo pacote
func main() {
  fmt.Println(v)       // Vai imprimir 7 → variável global
  {
    fmt.Println(v)     // Vai imprimir 7 → variável global
    v := 2
    fmt.Println(v)     // Vai imprimir 2 → variável local do bloco
  }
  fmt.Println(v)       // Vai imprimir 7 → Variável global
  f()
  // fmt.Println(x)    // Erro → x não é acessível
}

func f() {
  x := 5
  fmt.Println(x)       // Vai imprimir 5 na chamada da função → variável local
}
```

Passagem de Parâmetro por Valor e por Referência

Existem duas formas de passarmos os valores dos argumentos para os parâmetros das funções:

- Passagem por valor: o valor das variáveis dos argumentos não se alteram, mesmo que os parâmetros sejam modificado na função.
- Passagem por referência: Os valores das variáveis dos parâmetros refletem nos argumentos.

Na linguagem GO, as variáveis passadas como argumentos para as funções são sempre passados por valor.
Para passamos valores por referência, podemos usar o operadores de referência (&) que vimos no estudo dos ponteiros. Dessa forma passamos o endereço da variável.

```
package main
import "fmt"
func main() {
  v := 0
  incrementa(v)
  fmt.Println(v)  // 0
}
func incrementa(x int) {
  x++
}
```

```
package main
import "fmt"
func main() {
  v := 0
  incrementa(&v)
  fmt.Println(v) // 1
}
func incrementa(x *int) {
  *x++
}
```

No primeiro exemplo, o valor de **v** foi passado por valor e a função **incrementa** não vai funcionar.
Já no segundo exemplo, o valor de **v** foi passado por referência e seu valor foi incrementado para 1 corretamente.

Fim da Temporada 5 – Consolidando o Conhecimento

Exercício: Crie uma função para calcular as raízes de uma equação do segundo grau pela fórmula de Bhaskara. O retorno da função será um **slice** com as raízes. Caso o **slice** retorne vazio significa que equação não possui raízes reais.

```
package main

import (
  f "fmt"
  m "math"
)

func main() {
  var a, b, c float64
  f.Print("Digite o coeficiente a: ")
  f.Scan(&a)
  f.Print("Digite o coeficiente b: ")
  f.Scan(&b)
  f.Print("Digite o coeficiente c: ")
  f.Scan(&c)
  r := bhaskara(a, b, c)
```

→

Baixe os exercícios resolvidos no GITHUB

```
  if len(r) == 0 {
    f.Println("Equação não possui raízes reais")
  } else if len(r) == 1 {
    f.Println("x = ", r[0])
  } else {
    f.Printf("x1 = %f e x2 = %f\n", r[0], r[1])
  }
} // Fim main
func bhaskara(a, b, c float64) []float64 {
  raizes := make([]float64, 0, 10)
  delta := b*b - 4.0*a*c
  if delta == 0 {
    raizes = append(raizes, -b/(2.0*a))
  } else if delta > 0 {
    raizes = append(raizes, (-b+m.Sqrt(delta))/(2.0*a))
    raizes = append(raizes, (-b-m.Sqrt(delta))/(2.0*a))
  }
  return raizes
}
```

Exercícios Propostos

1) Um sistema de equações lineares como: $ax + by = c$ $dx + ey = f$ pode ser resolvido segundo: $x = \frac{ce - bf}{ae - bd}$ e $y = \frac{af - cd}{ae - bd}$

Escreva um algoritmo em GO que lê os coeficientes **a,b,c,d,e** e **f** e calcule de x e y.

2) Escreva um algoritmo em GO que leia 3 números inteiros e mostre o maior deles.

3) Leia um número N e escreva os N número maiores que N.
Exemplo: N = 4 → Escrever: 5,6,7,8

4) Leia um número **N** inteiro e calcule **S = 1 – 1/2 + 1/3 – 1/4 + ... +/- 1/N**

5) Leia um número inteiro N e um vetor (array ou slice) com números reais e Calcule o produto escalar entre o número N e o vetor.

6) Escreva uma função em GO para verificar se uma palavra informada pelo usuário é um palíndromo ou não.

FIM

Obrigado!

Neste curso fizemos uma pequena introdução à linguagem GO com o objetivo de dar condições ao aluno para se aprofundar nos estudos e nos assuntos não abordados aqui, tais como:

- GO Routines (Threads) e tratamento de erros
- Orientação a objetos
- Estruturas de dados complexas
- Bancos de dados e WEB

Veja a bibliografia a seguir para mais informações.

Saiba mais...

Links

- TutorialsPoint
- Geeks for Geeks
- A Tour of GO
- GO by Example

Livros

- The Little Go Book
- Essencial GO

Exercícios resolvidos e outros recursos

- GitHub
- http://www.josecintra.com/blog