Unit-3

Syllabus: *Functions, Variadic Functions, Closure, recursion, Defer, Panic, and Recover, Pointers, The * and & operators, new, Structs and Interfaces.*

Part-B: Structs - Initialization, Fields, Methods - Embedded Types.

Functions:

Function is a block of code that performs a specific task. For example, suppose we want to write code to create a circle and rectangle and color them.

In this case, we can organize our code by creating three different functions:

function to create a circle, function to create a rectangle, function to color

Create a Go Function

In Golang, we use the func keyword to create a function.

```
Func wish(){
//code
}
```

Here, wish() is the name of the function (denoted by ()) and code inside {....} indicates the function body.

Let's now create a function that prints **Good Morning**.

// Program to define a function

```
package main
import "fmt"
// define a function
func wish() {
  fmt.Println("Good Morning")
}
func main() {
}
```

When we run this program, we will not get the output.

This is because we are just defining a function. To execute a function, we need to call it first.

Function Call

We use the function's name followed by parenthesis to call a function.

```
wish()
```

Now, let's add a function call inside main().

```
// Program to define a function
package main
import "fmt"
// define a function
func wish() {
  fmt.Println("Good Morning")
}
func main() {
   // function call
  wish()
}
```

Output

```
Good Morning
```

Example: Go language Function

```
package main
import "fmt"
// function to add two numbers
func addNumbers() {
  num1 := 32
  num2 := 18
  tot := n1 + n2
  fmt.Println("Total:", tot)
}
func main() {
  // function call
  addNumbers()
}
```

Output

```
Sum: 20
```

Function Parameters in Golang

Here, we have created a function that does a single task, adds two numbers, 32 and 18.

However, in real projects, we want our functions to work dynamically. That is, instead of adding 32 and 18, the addNumbers() function should be able to add any two numbers.

In this case, we can create functions that accept external values and perform operations on them. These additional parameters are known as function parameters.

Define Function with Parameters

```
func addNumbers(num1 int, num2 int) {
  // code
}
```

addNumbers() function accepts two parameters: n1 and n2. Here, int denotes that both parameters are of integer type.

To call this function, we need to pass some values (known as function arguments).

// function call

```
addNumbers(32, 18)
```

Example: Function Parameters

// Program to illustrate function parameters

```
package main
import "fmt"
// define a function with 2 parameters
func addNumbers(num1 int, num2 int) {
  tot := num1 + num2
  fmt.Println("Total:", tot)
}
func main() {
  // pass parameters in function call
  addNumbers(32, 18)
}
```

Output

```
Total : 50
```

- The arguments are assigned to the function parameters when we call the function. 32 is assigned to num1 and 18 is assigned to num2.
- when we add num1 and num2 inside the function, we get 50 (32 + 18).

Return Value from Go Function

In last example, we have printed the value of the sum inside the function itself. However, we can also return value from a function and use it anywhere in our program.

Here's how we can create a Go function that returns a value:

```
func addNumbers(num1 int, num2 int) int {
  // code
  return tot
}
```

- Here, int before the opening curly bracket { indicates the return type of the function. In this case, int means the function will return an integer value.
- Return sum is the return statement that returns the value of the sum variable.
- The function returns value to the place from where it is called, so we need to store the returned value to a variable.

```
// function call
result := addNumbers(32, 18)
```

Here, we are storing the returned value to the result variable.

Example: Function Return Value

```
package main
import "fmt"
// function definition
func addNumbers(num1 int, num2 int) int {
  tot := n1 + n2
  return tot
}
func main() {
  // function call
  result := addNumbers(32, 18)
  fmt.Println("Total:",result)
}
```

Output

```
Total: 50
```

- when the return statement is encountered, it returns the value of sum.
- The returned value is assigned to the result variable inside main().
- The return statement should be the last statement of the function.
- When a return statement is encountered, the function terminates.

```
package main
import "fmt"
// function definition
func addNumbers(num1 int, num2 int) int {
  tot := num1 + num2
  return tot
  // code after return statement
  fmt.Println("After return statement")
}
func main() {
  // function call
  result := addNumbers(32, 18)
  fmt.Println("Total:",result)
}
```

We get a "***missing return at the end of function***" error as we have added a line after the return statement.

Return Multiple Values from Go Function

In Go, we can also return multiple values from a function. For example,

```
// Program to return multiple values from function
package main
import "fmt"
// function definition
func calculate(num1 int, num2 int) (int, int) {
  tot:= num1 + num2
  difference := num1 - num2
  // return two values
  return tot, difference
}
func main() {
  // function call
  tot, difference := calculate(32, 18)
  fmt.Println("Total:", tot, "Difference:", difference)
}
```

Output

Total: 50 Difference: 14

In the above example, we have created a function named calculate().

```
func calculate(num1 int, num2 int) (int, int) {
  ...
}
```

Here, (int, int) indicates that we are returning two integer values: tot and difference from the function.

Since the function returns two values, we have used two variables while calling the function.

```
tot, difference = calculate(32, 18);
```

Benefits of Using Functions

1. Code Reusability

We can reuse the same function multiple times in our program. For example,

```
// Program to illustrate code reusability in function
package main
import "fmt"
// function definition
func getSquare(num int) {
  square := num * num
  fmt.Printf("Square of %d is %d\n", num, square)
}
// main function
func main() {
  // call function 3 times
  getSquare(3)
  getSquare(5)
  getSquare(10)
}
```

Output

Square of 3 is 9

Square of 5 is 25

Square of 10 is 100

Here we have created the function named getSquare() to calculate the square of a number.

Here, we are reusing the same function multiple times to calculate the square of different numbers.

Code Readability: Functions help us break our code into chunks to make our program readable and easy to understand. It also makes the code easier to maintain and debug.

Variadic Functions

- A variadic function is a function that accepts a variable number of arguments.
- In Golang, it is possible to pass a varying number of arguments of the same type as referenced in the function signature.
- To declare a variadic function, the type of the final parameter is preceded by an ellipsis, "...", which shows that the function may be called with any number of arguments of this type.
- This type of function is useful when you don't know the number of arguments you are passing to the function, the best example is built-in Println function of the fmt package which is a variadic function.
- Select single argument from all arguments of variadic function

This program demonstrates the use of a variadic function in Golang. A variadic function is a function that takes a variable number of arguments of a specific type.

- In this program, the function variadic Example takes a variadic parameter of type string, indicated by the ... before the type name. This means that the function can accept any number of string arguments.
- In the main function, we call variadic Example with four string arguments: "red", "blue", "green", and "yellow". These arguments are passed to the s parameter of the variadic Example function as a slice of strings.

Example

```
package main
import "fmt"
func main() {
	variadicExample("red", "blue", "green", "yellow")
}
func variadicExample(s ...string) {
	fmt.Println(s[0])
	fmt.Println(s[3])
}
```

Output

```
red
yellow
```

Accessing the first and fourth elements of the s slice. Note that if we were to pass fewer than four arguments to variadic Example, the program would still run without error, but trying to access an index beyond the length of the slice would result in a runtime error.

Passing multiple string arguments to a variadic function

- This program demonstrates the use of variadic function in Go, which allows a function to accept a variable number of arguments of the same type.
- In this example, the function variadicExample() is defined to accept a variadic parameter of type string. This means that it can accept any number of string arguments.
- The main() function calls variadic Example() multiple times with different numbers of string arguments.
- The first call to variadicExample() is made without any arguments, which is allowed since the function is defined to accept zero or more string arguments.
- The second, third, and fourth calls pass different numbers of string arguments to variadicExample(). In each case, the function prints the contents of the s parameter using fmt.Println().

Example

```
package main
import "fmt"
func main() {
	variadicExample()
	variadicExample("red", "blue")
	variadicExample("red", "blue", "green")
	variadicExample("red", "blue", "green", "yellow")
}
func variadicExample(s ...string) {
	fmt.Println(s)
}
```

Output

```
[]
[red blue]
[red blue green]
[red blue green yellow]
```

The first call to variadicExample() prints an empty slice since no arguments were passed. The subsequent calls print the contents of the s parameter, which contains all the string arguments passed to the function.

Normal function parameter with variadic function parameter

Example

```
package main
import "fmt"
```

```
func main() {
	fmt.Println(calculation("Rectangle", 20, 30))
	fmt.Println(calculation("Square", 20))
}
func calculation(str string, y ...int) int {
	area := 1
	for _, val := range y {
		if str == "Rectangle" {
			area *= val
		} else if str == "Square" {
			area = val * val
		}
	}
	return area
}
```

Output

```
600
400
```

Pass different types of arguments in variadic function. In the following example, the function signature accepts an arbitrary number of arguments of type slice.

Example

```
package main
import (
	"fmt"
	"reflect"
)
func main() {
	variadicExample(1, "red", true, 10.5, []string{"foo", "bar", "baz"},
		map[string]int{"apple": 23, "tomato": 13})
}
func variadicExample(i ...interface{}) {
	for _, v := range i {
		fmt.Println(v, "--", reflect.ValueOf(v).Kind())
	}
}
```

Output

1 -- int

red -- string

true -- bool

10.5 -- float64

[foo bar baz] -- slice

map[apple:23 tomato:13] – map

Go Closure

- Go closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.
- Nested Functions, Returning a function

Nested function: In Go, we can create a function inside another function. This is known as a nested function.

Example:

```
package main
import "fmt"
// outer function
func wish(name string) {
  // inner function
  var displayName = func() {
    fmt.Println("Hi", name)
  }
  // call inner function
  displayName()
}
func main() {
  // call outer function
  wish("Mrudhu")  // Hi Mrudhu
}
```

In this example created an anonymous function inside the wish() function.

- Here, var displayName = func() {...} is a nested function.
- The nested function works similar to the normal function.
- It executes when displayName() is called inside the function wish().

Returning a function in Go

We can create a function that returns an anonymous function. For example,

```
package main
import "fmt"
func wish() func() {
  return func() {
    fmt.Println("Hi Mrudhu")
  }
}
func main() {
  g1 := wish()
  g1()
}
```

Output

```
Hi Mrudhu
```

In the above example, we have created the Mrudhu() function.

```
func wish() func() {...}
```

Here, func() before the curly braces indicates that the function returns another function.

Also, if you look into the return statement of this function, we can see the function is returning a function.

From main(), we call the wish() function.

```
g1 := wish()
```

Here, the returned function is now assigned to the g1 variable. Hence, g1() executes the nested anonymous function.

Go Closures

Closure is a nested function that helps us access the outer function's variables even after the outer function is closed.

Example:

```
package main
import "fmt"
// outer function
func wish() func() string {
  // variable defined outside the inner function
  name := "Mrudu"
    // return a nested anonymous function
  return func() string {
```

```
    name = "Hi " + name
    return name
  }
}
func main() {
  // call the outer function
  message := wish()
  // call the inner function
  fmt.Println(message())
}
```

Output

```
Hi Mrudu
```

In the above example, we have created a function named wish() that returns a nested anonymous function.

Here, when we call the function from main().

```
message := wish()
```

- The returned function is now assigned to the message variable.
- At this point, the execution of the outer function is completed, so the name variable should be destroyed. However, when we call the anonymous function using

```
fmt.Println(message())
```

- we are able to access the name variable of the outer function.
- It's possible because the nested function now acts as a closure that closes the outer scope variable within its scope even after the outer function is executed.

Example: Print Odd Numbers using Golang Closure

```
package main
import "fmt"
func calculate() func() int {
  num := 1
  // returns inner function
  return func() int {
    num = num + 2
    return num
  }
}
```

```
func main() {
  // call the outer function
  odd := calculate()
  // call the inner function
  fmt.Println(odd())
  fmt.Println(odd())
  fmt.Println(odd())
  // call the outer function again
  odd2 := calculate()
  fmt.Println(odd2())
}
```

Output

```
3
5
7
3
```

In this example,

odd := calculate()

This code executes the outer function calculate() and returns a closure to the odd number. That's why we can access the num variable of calculate() even after completing the outer function. Again, when we call the outer function using

odd2 := calculate()

a new closure is returned. Hence, we get 3 again when we call odd2().

Closure helps in Data Isolation

As we see in the previous example, a new closure is returned every time we call the outer function. Here, each returned closure is independent of one another, and the change in one won't affect the other.

This helps us to work with multiple data in isolation from one another.

Example:

```
package main
import "fmt"
func displayNumbers() func() int {
  number := 0
  // inner function
  return func() int {
```

```
  number++
  return number
  }
}
func main() {
  // returns a closure
  num1 := displayNumbers()
  fmt.Println(num1())
  fmt.Println(num1())
  fmt.Println(num1())
  // returns a new closure
  num2 := displayNumbers()
  fmt.Println(num2())
  fmt.Println(num2())
}
```

Output

```
1
2
3
1
2
```

In the above example, the displayNumbers() function returns an anonymous function that increases the number by 1.

```
return func() int {
  number++
  return number
}
```

- Inside the main() function, we assign the function call to num1 and num2 variables.
- Here, we first call the closure function using num1().
- In this case, we get outputs 1, 2, and 3.
- Again, we call it using num2(). In this case, the value of the number variable doesn't start from 3; instead, it starts from 1 again.

This shows that the closure returned from displayNumbers() is isolated from one another.

Go Recursion

In computer programming, a recursive function calls itself.

Example:

```
func recurse() {
  … …
  … …
  recurse()
}
```

Here, the recurse() function includes the function call within its body. Hence, it is a Go recursive function and this technique is called recursion.

Example: Recursion in Go prog

```
package main
import "fmt"
func countDown(number int) {
  // display the number
  fmt.Println(number)
  // recursive call by decreasing number
  countDown(number - 1)
  }
func main() {
  countDown(3)
}
```

Output

```
Countdown Starts:
3
2
1
0
-1
…
…
```

In the above example, we have created a function named countDown(). Note that we have added the function call inside the function.

```
countDown(number - 1)
```

Here, this is a recursive function call and we are decreasing the value of number in each call.

However, this function will be executed infinitely because we have added the function call directly within the function

To avoid infinite recursion, we use conditional statements and only call the function if the condition is met.

Recursive Function with conditional statement

In this example, we will use an if...else statement to prevent the infinite recursion.

```
// Program to end the recursive function using if…else
package main
import "fmt"
func countDown(number int) {
  if number > 0 {
    fmt.Println(number)
    // recursive call
    countDown(number - 1)
  } else {
    // ends the recursive function
    fmt.Println("Countdown Stops")
  }
 }
func main() {
  countDown(3)
}
```

Output

```
Countdown Starts
3
Countdown Starts
2
Countdown Starts
1
Countdown Starts
Countdown Stops
```

In the above example, we have added the recursive call inside the if statement.

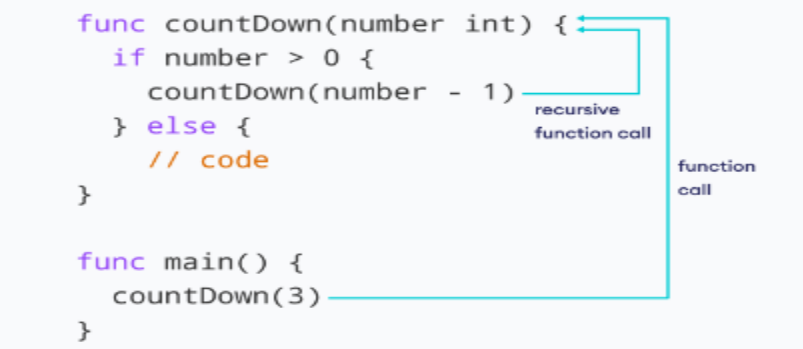
```
if number > 0 {
  fmt.Println(number)
  // recursive call
  countDown(number - 1)
}
```

Here, we are calling the function only if the number is greater than 0.

If the number is not greater than 0, the recursion ends. This is called the stopping condition.

Working of the program

number > 0	Print	Recursive Call
true	3	countDown(2)
true	2	countDown(1)
true	1	countDown(0)
false	Countdown stops	function execution stops



Example: Go program to calculate the sum of positive numbers

```
package main
import "fmt"
func sum(number int) int {
  // condition to break recursion
  if number == 0 {
    return 0
  } else {
    return number + sum(number-1)
  }
}
func main() {
  var num = 50
  // function call
  var result = sum(num)
  fmt.Println("Sum:", result)
}
```

Output

Sum: 1275

In the above example, we have created a recursive function named sum() that calls itself if the value of number is not equal to 0.

return number + sum(number - 1)

- In each iteration, we are calling the function by decreasing the value of number by 1.
- In first call, the value of number is 50 which is not equal to 0. So, the else block is executed which returns 50 + sum(49).
- Again, 49 is not equal to 0, so return 49 + sum(48) is executed.
- This process continues until number becomes 0. When number is 0, return 0 is executed and it is added to the other values.

Hence, finally, 50 + 49 + 48 + ... + 0 is computed and returned to the main() function.

Factorial of a number using Go Recursion

```
package main
import "fmt"
func factorial (num int) int {
  // condition to break recursion
  if num == 0 {
    return 1
  } else {
    // condition for recursion call
    return num * factorial (num-1)
   }
}
func main() {
  var number = 3
    // function call
  var result = factorial (number)
  fmt.Println("The factorial of 3 is", result)
}
```

Output

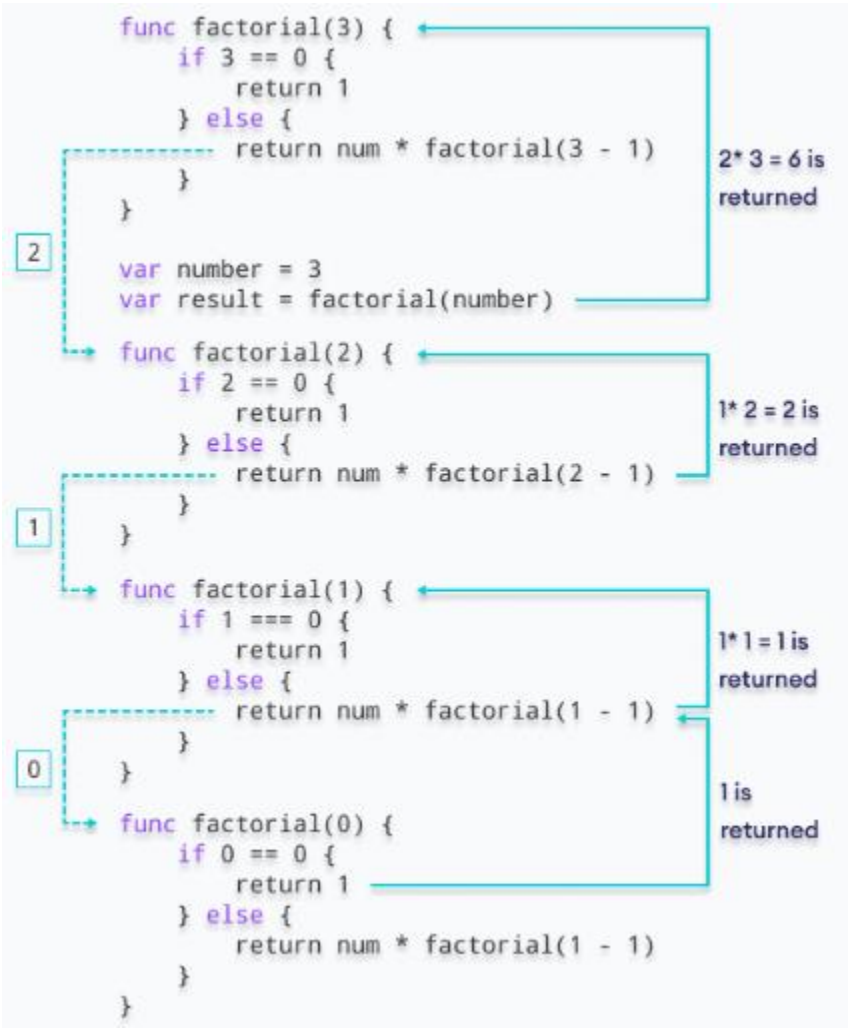
The factorial of 3 is 6

In the above example, we have created a recursive function named factorial() that calls itself if the value of num is not equal to 0.

return num * factorial(num - 1)

In each call, we are decreasing the value of num by 1.

Here's how the program works:



Go defer, panic and recover

- In Go, we use defer, panic and recover statements to handle errors.
- We use defer to delay the execution of functions that might cause an error.
- The panic statement terminates the program immediately and recover is used to recover the message during panic.

Before we learn about error handling statements, make sure to know about Go errors.

defer in Go

We use the defer statement to prevent the execution of a function until all other functions execute. For example,

package main

import "fmt"

func main() {

```
  // defer the execution of Println() function
  defer fmt.Println("Three")
  fmt.Println("One")
  fmt.Println("Two")
}
```

Output

```
One
Two
Three
```

In the above example, we have used the defer before the first print statement.

That's why the Println() function is executed last after all other functions are executed.

Multiple defer Statements in Go

When we use multiple defer in a program, the order of execution of the defer statements will be LIFO (Last In First Out).

This means the last defer statement will be executed first. For example,

```
package main
import "fmt"
func main() {
  defer fmt.Println("One")
  defer fmt.Println("Two")
  defer fmt.Println("Three")
}
```

Output

```
Three
Two
One
```

In the above example, we are calling the Println() function using 3 defer statements.

- The order of execution is LIFO.
- The last defer statement is executed first, and the first defer statement is executed last

Golang panic

We use the panic statement to immediately end the execution of the program. If our program reaches a point where it cannot be recovered due to some major errors, it's best to use panic.

The lines of code after the panic statement are not executed. For example,

```
package main
import "fmt"
func main() {
  fmt.Println("Help! Something bad is happening.")
  panic ("Ending the program")
  fmt.Println("Waiting to execute")
}
```

Output

```
Help! Something bad is happening.
panic: Ending the program
```

Here, the program is terminated when it reaches the panic statement. That's why the print statement after the panic is not executed.

Note: panic: in the output specifies that the program is terminated due to panic and it's the panic message.

Example: Panic in Golang

```
package main
import "fmt"
func division(num1, num2 int) {
  // if num2 is 0, program is terminated due to panic
  if num2 == 0 {
    panic("Cannot divide a number by zero")
  } else {
    result := num1 / num2
    fmt.Println("Result: ", result)
  }
}
func main() {
  division(4, 2)
  division(8, 0)
  division(2, 8)
}
```

Output

```
Result:  2
panic: Cannot divide a number by zero
```

In the above example, we have created a function that performs the division of two numbers: num1 and num2.

Inside the function, we have used an if...else statement that checks if num2 (denominator) is 0 or not. If it is 0, the execution of the program stops because of the panic statement.

panic("Cannot divide a number by zero")

Here, when we run the program, we first get result 2 (division of 4 and 2). However, during the second function call, the value of num2 is 0 (division of 8 and 0).

Hence, the execution of the program is terminated.

recover in Go Programming

- The panic statement immediately terminates the program.
- However, sometimes it might be important for a program to complete its execution and get some required results.

In such cases, we use the recover statement to handle panic in Go. The recover statement prevents the termination of the program and recovers the program from panic.

Example:

```
package main
import "fmt"
// recover function to handle panic
func handlePanic() {
  // detect if panic occurs or not
  a := recover()
  if a != nil {
    fmt.Println("RECOVER", a)
  }
}
func division(num1, num2 int) {
  // execute the handlePanic even after panic occurs
  defer handlePanic()
  // if num2 is 0, program is terminated due to panic
  if num2 == 0 {
    panic("Cannot divide a number by zero")
  } else {
    result := num1 / num2
    fmt.Println("Result: ", result)
```

```
  }
}
func main() {
  division(4, 2)
  division(8, 0)
  division(2, 8)
}
```

Output

```
Result:  2
RECOVER Cannot divide a number by zero
Result:  0
```

In the above example, we have created a handlePanic() function to recover from the panicking state.

```
func handlePanic() {
  // detect if panic occurs or not
  a := recover()
  if a != nil {
    fmt.Println("RECOVER", a)
  }
}
```

Here, we have used a := recover() to detect any occurrence of panic in our program and assign the panic message to a.

In our example, a panic occurs, so there will be some value in a. Hence, the if statement is executed, which prints the panic message.

Inside the division() function, we are calling the handlePanic() function

```
defer handlePanic()
```

- calling the handlePanic() before the occurrence of panic.
- It's because the program will be terminated if it encounters panic and we want to execute handlePanic() before the termination.
- We are using defer to call handlePanic(). It's because we only want to handle the panic after it occurs, so we are deferring its execution.

The * and & operators

- In Go (Golang), the * and & operators are used to work with pointers, which are variables that store memory addresses of other values.

- Pointers are powerful tools for working with data directly in memory and for sharing data between functions efficiently.

*** (Asterisk) Operator:**

- The * operator is used to declare a pointer variable or to dereference a pointer.
- When used in a declaration, it indicates that the variable being declared is a pointer to the specified data type.
- When used as a prefix to an existing pointer variable, it dereferences the pointer, giving access to the value stored at the memory address it points to.

Declaration of a pointer:

var ptr *int

Here, ptr is a pointer to an integer (int).

Dereferencing a pointer:

x := 42

ptr := &x // ptr points to the memory address of x

fmt.Println(*ptr) // Output: 42 (value stored at the memory address pointed by ptr)

& (Ampersand) Operator:

The & operator is used to get the memory address of a variable. When used before a variable, it returns the memory address where the variable is stored.

x := 42

ptr := &x // ptr now contains the memory address of x

In this example, ptr is a pointer to an integer (int), and it holds the memory address of x. So, ptr points to the location in memory where the value of x is stored.

Together, these operators allow you to create and work with pointers in Go. They are used extensively in situations where you need to pass large data structures to functions without copying them entirely, or when you want to modify the original data from different functions, among other use cases. However, it is essential to use pointers carefully to avoid common pitfalls like null pointer dereferences or memory leaks.

new pointer in go programming

In Go programming, the new function is used to create a new value of a specified type and returns a pointer to that newly allocated value. The new function initializes the allocated memory to zero (or the zero value for the specified type).

The syntax of the new function is straightforward:

func new(Type) *Type

Here, Type is the data type for which you want to allocate memory. The return value is a pointer to a newly allocated zero-initialized value of the specified type.

```
package main
import "fmt"
type Person struct {
    Name string
    Age  int
}
func main() {
    // Create a new pointer to a Person struct
    p := new(Person)
    // Since new allocates memory and initializes it to zero,
    // the fields of the struct are already set to their zero values
    // which are "" (empty string) and 0 (integer zero).
    fmt.Printf("Name: %s, Age: %d\n", p.Name, p.Age)
}
```

Output:

```
Name: , Age: 0
```

In the example above, we used new(Person) to create a new pointer to a Person struct. The Person struct has two fields: Name and Age. Since the new function initializes the allocated memory to zero, the fields of the Person struct (Name and Age) are set to their zero values ("" and 0, respectively) when the pointer is created.

It's important to note that the new function is rarely used in idiomatic Go code. Instead, it is more common to use composite literals to create and initialize data structures directly, as it provides more control over the initial values of the struct fields:

```
p := &Person{
    Name: "Mrudhu Sam",
    Age:  30,
}
```

This creates a new Person struct using a composite literal and returns a pointer to it. The fields are explicitly initialized with the provided values ("Mrudhu Sam " for Name and 30 for Age). This method is preferred over new for creating and initializing data structures in a more readable and expressive manner.

Structs and interfaces are essential components in Go programming, offering powerful tools for creating custom data types and enabling polymorphism.

Structs:

A struct is a composite data type that groups together zero or more values with different data types into a single entity. It is similar to a class in object-oriented programming languages.

Syntax: To define a struct, you use the type keyword followed by the struct's name and a set of fields enclosed in curly braces.

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

Initialization: You can create an instance of a struct using a composite literal and then access its fields.

```
person := Person{Name: "Mrudhu”, Age: 30}
fmt.Println(person.Name, person.Age) // Output: Mrudhu 30
```

Methods: A struct can have associated methods that operate on its instance. Methods are defined using the function keyword with a receiver argument.

```
func (p Person) SayHello() {
   fmt.Printf("Hello, my name is %s and I'm %d years old.\n", p.Name, p.Age)
}
```

Interfaces:

Definition: An interface is a collection of method signatures that define a contract. A type implements an interface if it provides definitions for all the methods declared in the interface.

Syntax: To define an interface, you use the type keyword followed by the interface's name and a set of method signatures.

```
type Animal interface {
   Speak() string
   Move() string
}
```

Implementation: A type implicitly implements an interface if it implements all the methods declared in the interface. There is no explicit "implements" keyword like in some other languages.

```
type Dog struct{}
func (d Dog) Speak() string {
   return "Woof!"
```

```
}
func (d Dog) Move() string {
    return "Running"
}
```

Polymorphism: Interfaces enable polymorphism, allowing you to create functions that can accept different types as long as they implement the interface.

```
func Describe(a Animal) {
    fmt.Printf("The animal says %s and moves by %s.\n", a.Speak(), a.Move())
}
func main() {
    dog := Dog{}
    cat := Cat{}
    Describe(dog) // Output: The animal says Woof! and moves by Running.
    Describe(cat) // Output: The animal says Meow! and moves by Jumping.
}
```

Structs and interfaces are essential concepts in Go programming, and they provide a foundation for writing clean, modular, and flexible code. Understanding how to define and use structs and interfaces is crucial for building efficient and maintainable Go programs.

Structs – Initialization:

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}
```

Using Named Fields:

```
p1 := Person{
    FirstName: "John",
    LastName:  "Doe",
    Age:       30,
}
```

Using Positional Arguments:

If you know the order of fields, you can use positional arguments. But this can be error-prone if the structure of the struct changes in the future.

p2 := Person{"Jane", "Doe", 28}

Zero Value Initialization:

when declare a struct without explicit initialization, its fields get their zero values.

var p3 Person // All fields will have their zero values: "", "", 0

Pointer to a Struct:

You can also create a pointer to a new struct using the new function. This will allocate memory and return a pointer to the struct. The fields will have their zero values.

p4 := new(Person) // p4 is a pointer (*Person)

Using the & operator:

This is a common method when you want both initialization and a pointer to the struct:

```
p5 := &Person{
    FirstName: "Bob",
    LastName:  "Builder",
    Age:       40,
}
```

Anonymous Structs:

If you don't intend to use a struct type in multiple places, you can define and initialize an anonymous struct:

```
anon := struct {
    Name string
    ID   int
}{
    Name: "Anonymous",
    ID:   1,
}
```

Fields

We can access fields using the . operator:

fmt.Println(c.x, c.y, c.r)

c.x = 10

c.y = 5

Let's modify the circleArea function so that it uses a Circle:

```
func circleArea(c Circle) float64 {
  return math.Pi * c.r*c.r
}
```

In main we have:

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(c))
```

One thing to remember is that arguments are always copied in Go. If we attempted to modify one of the fields inside of the circleArea function, it would not modify the original variable. Because of this we would typically write the function like this:

```
func circleArea(c *Circle) float64 {
  return math.Pi * c.r*c.r
}
```

And change main:

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(&c))
```

Methods

Although this is better than the first version of this code, we can improve it significantly by using a special type of function known as a method:

```
func (c *Circle) area() float64 {
  return math.Pi * c.r*c.r
}
```

In between the keyword func and the name of the function we've added a “receiver”. The receiver is like a parameter – it has a name and a type – but by creating the function in this way it allows us to call the function using the . operator:

```
fmt.Println(c.area())
```

This is much easier to read, we no longer need the & operator (Go automatically knows to pass a pointer to the circle for this method) and because this function can only be used with Circles we can rename the function to just area.

Let's do the same thing for the rectangle:

```
type Rectangle struct {
  x1, y1, x2, y2 float64
}
func (r *Rectangle) area() float64 {
  l := distance(r.x1, r.y1, r.x1, r.y2)
  w := distance(r.x1, r.y1, r.x2, r.y1)
  return l * w
}
```

main has:

```
r := Rectangle{0, 0, 10, 10}
fmt.Println(r.area())
```

Embedded Types

A struct's fields usually represent the has-a relationship. For example a Circle has a radius. Suppose we had a person struct:

```
type Person struct {
  Name string
}
func (p *Person) Talk() {
  fmt.Println("Hi, my name is", p.Name)
}
```

And we wanted to create a new Android struct. We could do this:

```
type Android struct {
  Person Person
  Model string
}
```

This would work, but we would rather say an Android is a Person, rather than an Android has a Person. Go supports relationships like this by using an embedded type. Also known as anonymous fields, embedded types look like this:

```
type Android struct {
```

```
  Person

  Model string

}
```

We use the type (Person) and don't give it a name. When defined this way the Person struct can be accessed using the type name:

```
a := new(Android)

a.Person.Talk()
```

But we can also call any Person methods directly on the Android:

```
a := new(Android)

a.Talk()
```

The is-a relationship works this way intuitively: People can talk, an android is a person, therefore an android can talk.