1.What is the structure of Golang program.

package main

import "fmt"

func main() {

/\* This is my first sample program. \*/

fmt.Println("Hello, World!")

}

2What are the tokens in Go statement, line seperators and comments

3.What is a Go identifier

4.What are the diffrenet datatypes in Go lang

5.what is the variable definition in GO

var variable\_list optional\_data\_type;

var i, j, k int;

var c, ch byte;

var f, salary float32;

d = 42;

5.Go constants and Literals with examples

6.Explain go lang types of operators.

7. Write a program to print Random number.

package main

import ( "fmt"

"math/rand"

)

func main() { fmt.Println("My favorite number is", rand.Intn(10))

}

8. Import packages

**Imports**

This code groups the imports into a parenthesized, "factored" import statement.

You can also write multiple import statements, like:

import "fmt"

import "math"

But it is good style to use the factored import statement.

package main

import ("fmt"

"math"

)

func main() { fmt.Printf("Now you have %g problems.", math.Sqrt(7))

}

9..

package main

import ("fmt"

"math"

)

func main() { fmt.Println(math.pi)

}

10.Functions

A function can take zero or more arguments.

In this example, add takes two parameters of type int.

package main

import "fmt"

func add(x int, y int) int {

return x + y

}

func main() {

fmt.Println(add(42, 13))

}

## 11. Functions continued

When two or more consecutive named function parameters share a type, you can omit the type from all but the last.

In this example, we shortened

x int, y int

to

x, y int

package main

import "fmt"

func add(x, y int) int { return x + y

}

func main() { fmt.Println(add(42, 13))

}

12.Function with multiple results

**Multiple results**

A function can return any number of results.

The swap function returns two strings.

package main

import "fmt"

func swap(x, y string) (string, string) {

return y, x

}

func main() {

a, b := swap("hello", "world")

fmt.Println(a, b)

}

## 13. Named return values

Go's return values may be named. If so, they are treated as variables defined at the top of the function.

These names should be used to document the meaning of the return values.

A return statement without arguments returns the named return values. This is known as a "naked" return.

Naked return statements should be used only in short functions, as with the example shown here. They can harm readability in longer functions.

package main

import "fmt"

func split(sum int) (x, y int) {

x = sum \* 4 / 9

y = sum - x

return

}

func main() { fmt.Println(split(17))

}

## 14. Variables

The var statement declares a list of variables; as in function argument lists, the type is last.

A var statement can be at package or function level. We see both in this example.

package main

import "fmt"

var c, python, java bool

func main() {

var i int

fmt.Println(i, c, python, java)

}

**15.Variables with initializers**

A var declaration can include initializers, one per variable.

If an initializer is present, the type can be omitted; the variable will take the type of the initializer.

package main

import "fmt"

var i, j int = 1, 2

func main() { var c, python, java = true, false, "no!"

fmt.Println(i, j, c, python, java)

}

## 16. Short variable declarations

Inside a function, the := short assignment statement can be used in place of a vardeclaration with implicit type.

Outside a function, every statement begins with a keyword (var, func, and so on) and so the := construct is not available.

package main

import "fmt"

func main() {

var i, j int = 1, 2

k := 3

c, python, java := true, false, "no!"

fmt.Println(i, j, k, c, python, java)

}

## 17. Basic types

Go's basic types are

bool

string

int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32

// represents a Unicode code point

float32 float64

complex64 complex128

The example shows variables of several types, and also that variable declarations may be "factored" into blocks, as with import statements.

The int, uint, and uintptr types are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems. When you need an integer value you should use int unless you have a specific reason to use a sized or unsigned integer type.

package main

import ( "fmt"

"math/cmplx"

)

var ( ToBe bool = false

MaxInt uint64 = 1<<64 - 1

z complex128 = cmplx.Sqrt(-5 + 12i)

)

func main() {

fmt.Printf("Type: %T Value: %v\n", ToBe, ToBe)

fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)

fmt.Printf("Type: %T Value: %v\n", z, z)

}

## 18.Zero values

Variables declared without an explicit initial value are given their *zero value*.

The zero value is:

* 0 for numeric types,
* false for the boolean type, and
* "" (the empty string) for strings.

package main

import "fmt"

func main() {

var i int

var f float64

var b bool

var s string

fmt.Printf("%v %v %v %q\n", i, f, b, s)

}

## 19. Type conversions

The expression T(v) converts the value v to the type T.

Some numeric conversions:

var i int = 42

var f float64 = float64(i)

var u uint = uint(f)

Or, put more simply:

i := 42

f := float64(i)

u := uint(f)

Unlike in C, in Go assignment between items of different type requires an explicit conversion. Try removing the float64 or uint conversions in the example and see what happens.

package main

import (

"fmt"

"math"

)

func main() {

var x, y int = 3, 4

var f float64 = math.Sqrt(float64(x\*x + y\*y))

var z uint = uint(f)

fmt.Println(x, y, z)

}

## 20. Type inference

When declaring a variable without specifying an explicit type (either by using the := syntax or var = expression syntax), the variable's type is inferred from the value on the right hand side.

When the right hand side of the declaration is typed, the new variable is of that same type:

var i int

j := i // j is an int

But when the right hand side contains an untyped numeric constant, the new variable may be an int, float64, or complex128 depending on the precision of the constant:

i := 42 // int

f := 3.142 // float64

g := 0.867 + 0.5i // complex128

Try changing the initial value of v in the example code and observe how its type is affected.

package main

import "fmt"

func main() {

v := 42 // change me!

fmt.Printf("v is of type %T\n", v)

}

## 21. Constants

Constants are declared like variables, but with the const keyword.

Constants can be character, string, boolean, or numeric values.

Constants cannot be declared using the := syntax.

package main

import "fmt"

const Pi = 3.14

func main() {

const World = "世界"

fmt.Println("Hello", World)

fmt.Println("Happy", Pi, "Day")

const Truth = true

fmt.Println("Go rules?", Truth)

}

## 22.Numeric Constants

Numeric constants are high-precision *values*.

An untyped constant takes the type needed by its context.

Try printing needInt(Big) too.

(An int can store at maximum a 64-bit integer, and sometimes less.)

package main

import "fmt"

const (

// Create a huge number by shifting a 1 bit left 100 places.

// In other words, the binary number that is 1 followed by 100 zeroes.

Big = 1 << 100

// Shift it right again 99 places, so we end up with 1<<1, or 2.

Small = Big >> 99

)

func needInt(x int) int { return x\*10 + 1 }

func needFloat(x float64) float64 {

return x \* 0.1

}

func main() {

fmt.Println(needInt(Small))

fmt.Println(needFloat(Small))

fmt.Println(needFloat(Big))

}

## 23. For

Go has only one looping construct, the for loop.

The basic for loop has three components separated by semicolons:

* the init statement: executed before the first iteration
* the condition expression: evaluated before every iteration
* the post statement: executed at the end of every iteration

The init statement will often be a short variable declaration, and the variables declared there are visible only in the scope of the for statement.

The loop will stop iterating once the boolean condition evaluates to false.

*Note*: Unlike other languages like C, Java, or JavaScript there are no parentheses surrounding the three components of the for statement and the braces { } are always required.

package main

import "fmt"

func main() {

sum := 0

for i := 0; i < 10; i++ {

sum += i

}

fmt.Println(sum)

}

## For continued

The init and post statement are optional.

package main

import "fmt"

func main() {

sum := 1

for ; sum < 1000; {

sum += sum

}

fmt.Println(sum)

}

## Forever

If you omit the loop condition it loops forever, so an infinite loop is compactly expressed.

package main

func main() {

for {

}

}

## If

Go's if statements are like its for loops; the expression need not be surrounded by parentheses ( ) but the braces { } are required.

package main

import ("fmt"

"math"

)

func sqrt(x float64) string {

if x < 0 { return sqrt(-x) + "i"

}

return fmt.Sprint(math.Sqrt(x))

}

func main() { fmt.Println(sqrt(2), sqrt(-4))

}

## If with a short statement

Like for, the if statement can start with a short statement to execute before the condition.

Variables declared by the statement are only in scope until the end of the if.

(Try using v in the last return statement.)

package main

import ("fmt"

"math")

func pow(x, n, lim float64) float64 {

if v := math.Pow(x, n); v < lim {

return v

}

return lim

}

func main() {

fmt.Println(

pow(3, 2, 10),

pow(3, 3, 20),

)

}

## If and else

Variables declared inside an if short statement are also available inside any of the elseblocks.

(Both calls to pow are executed and return before the call to fmt.Println in main begins.)

package main

import (

"fmt"

"math"

)

func pow(x, n, lim float64) float64 {

if v := math.Pow(x, n); v < lim {

return v

} else { fmt.Printf("%g >= %g\n", v, lim)

}

// can't use v here, though

return lim

}

func main() { fmt.Println(pow(3, 2, 10),

pow(3, 3, 20),

)

}

## Switch

A switch statement is a shorter way to write a sequence of if - else statements. It runs the first case whose value is equal to the condition expression.

Go's switch is like the one in C, C++, Java, JavaScript, and PHP, except that Go only runs the selected case, not all the cases that follow. In effect, the break statement that is needed at the end of each case in those languages is provided automatically in Go. Another important difference is that Go's switch cases need not be constants, and the values involved need not be integers.

package main

import (

"fmt"

"runtime"

)

func main() {

fmt.Print("Go runs on ")

switch os := runtime.GOOS; os {

case "darwin":

fmt.Println("OS X.")

case "linux":

fmt.Println("Linux.")

default:

// freebsd, openbsd,

// plan9, windows...

fmt.Printf("%s.", os)

}

}

## Switch evaluation order

Switch cases evaluate cases from top to bottom, stopping when a case succeeds.

(For example,

switch i {

case 0:

case f():

}

does not call f if i==0.)

package main

import ("fmt"

"time"

)

func main() {

fmt.Println("When's Saturday?")

today := time.Now().Weekday()

switch time.Saturday {

case today + 0: fmt.Println("Today.")

case today + 1: fmt.Println("Tomorrow.")

case today + 2: fmt.Println("In two days.")

default: fmt.Println("Too far away.")

}

}

## Switch with no condition

Switch without a condition is the same as switch true.

This construct can be a clean way to write long if-then-else chains.

package main

import ("fmt"

"time"

)

func main() {

t := time.Now()

switch {

case t.Hour() < 12:

fmt.Println("Good morning!")

case t.Hour() < 17:

fmt.Println("Good afternoon.")

default:

fmt.Println("Good evening.")

}

}

## Defer

A defer statement defers the execution of a function until the surrounding function returns.

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

package main

import "fmt"

func main() { defer fmt.Println("world")

fmt.Println("hello")

}

## Stacking defers

Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

package main

import "fmt"

func main() {

fmt.Println("counting")

for i := 0; i < 10; i++ {

defer fmt.Println(i)

}

fmt.Println("done")

}