**SECTION 2. GOLANG PROGRAMMING LANGUAGE OVERVIEW**

**2.1.** [**What is the Go Programming Language?**](https://www.techtarget.com/searchitoperations/definition/Go-programming-language)

Since this work uses an integrated approach to software implementation of algorithms, it is necessary to at least briefly describe the basic concepts and constructions of its components: visual algorithmic language DRAKON and programming language Golang. It is thought that this approach is very promising, foremost, in the field of education, because it promotes the formation and development of algorithmic, computer thinking and is quite justified for application in the field of computer science as data structures and algorithms. However, it is assumed that the reader is familiar with at least one of the modern programming languages. At the same time, the Golang language description is limited to constructs sufficient to understand data structure algorithms.

This section focuses on the features of software constructions: *variables, arrays, slicess, maps, pointers, logical operators, cycles, structures, recursion,* etc. Special emphasis is placed on the description of data types both basic and user. It is also considered to be an interface. Everything that goes beyond "Data structures and algorithms" is presented in various materials, which are given both in the course of the text and in the list of literature.

Golang (Go) is a general-purpose language designed with system programming in mind. The language was originally developed at Google in 2007 by Robert Grizemer, Rob Pike and Ken Thompson. It is strongly and statically typed (requires precise indication of types of variables), provides built-in support for garbage collection and supports parallel programming []. The following are the most important features of Go programming language.

* Simple and clear syntax. This makes writing code a pleasant occupation.
* Static typing. Avoids inadvertent errors, simplifies the reading and understanding of code, makes code unambiguous.
  + Speed and compilation. Go’s speed is ten times faster than scripting languages, with less memory consumption. At the same time, the compilation is almost instantaneous. The whole project is compiled into one binary file, without dependencies. As they say, "just add water". And you don’t have to worry about memory, you have a garbage collector.
  + Deviation from the concept of object-oriented programming (OOP). The language does not have classes, but there are data structures with methods. Inheritance is replaced by an embedding mechanism. There are interfaces that do not need to be explicitly implemented, but only enough to implement the interface methods.
* Parallelism. Parallel computations in the language are simple and graceful. Gorutins (something like threads) are lightweight, consume little memory.
* Rich standard library. The language has everything necessary for web development and not only. The number of third-party libraries is constantly growing. You can also use C and C++ libraries.
* Ability to write in a functional style. The language has closures (closures) and anonymous functions. Functions are first-order objects, they can be passed as arguments and used as data types.

**2.2. Program structure in Go**

The structure of the program in the language Golang is presented in Figure 2.1.

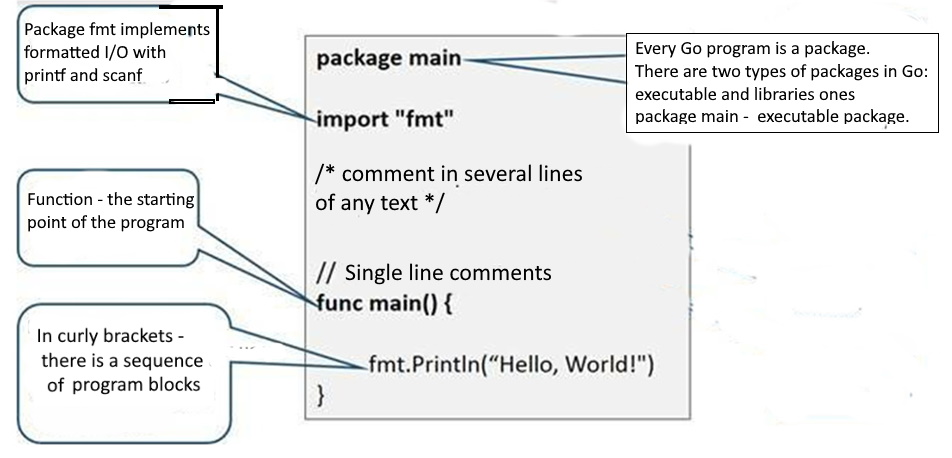


Figure 2.1. Program structure in Go language

Every program on Go is a package. Depending on its content, this package can be the main program file, or a library with problem-oriented code.// название программы (пакета)

package main

……………………….

// loading the right packages if the commands from these packages are needed in our program import (

"fmt",

“flag”

)

/\* main function means that its contents will be executed after the program is run. In addition to it, the program can have as many other functions /\*

func main() {

// function content

fmt.Println("Hello World")

}

Comments in Go exist in two types: single-line and multi-line.

// - one-line commentary;

/\* \*/ - multiline comment placed between these symbols

**2.3. Variables and constants**

A variable in any programming language assigns a storage location for a value associated with a symbolic name or identifier. One variable or list of variables used in the program is declared via the keyword var. Values are assigned to variables by the sign "=". When declaring a variable, you must specify its type so that the compiler knows how to process it. That is why the language Go is called strictly typed.

var i int

var s string

var f bool,

here i is an integer variable; s is a string variable, f is a logical variable.

When declaring a variable, you can assign initial values that can be changed later:

var a int = 25

var a, b, c int 100, 200 300

var y = float32 = 32.5

var z string = "Введите новое значение "

In the absence of initial values, the variables are either zeroed (for numeric types) or filled in with empty lines (for string data). It is possible to briefly declare variables via the operator :=, for example a := 2.5. However, in this case, the variable is local, that is, only available in a separate fragment of the program.

Assigning variable names should be based on certain rules and style.

* + - variable names can only contain one word (without spaces).
    - variable names can only contain letters, digits and underscore characters (\_).
    - variable names cannot start with digits.

The variables are case-sensitive, but the case of the first letter of the variable name in Go is particularly important. If the variable name starts with a capital letter, this means that the variable is available outside the package where it was declared (the exported variable). If the variable name starts with a lowercase letter, it will only be available in the package where it is declared.

Constants are program objects whose values do not change in the program code. Constants are declared with the keyword: const:

const item string

const n int

const y float64

It is allowed to assign initial values without declaring type:const item = "name"

const n = 25

const  y = 45.5

**2.4. Input and output**

Data entry is done by using the Scan(&Variable Name) function to simply put the entered value into a variable or Scanf(%format, &Variable Name) to pre-specify the type of data to be entered.

To implement the data output at the beginning of the program (after package main), the corresponding fmt package is imported, which contains many output functions depending on the type of variables, location and explanatory string enclosed in double quotation marks ( n - line translation). The prefix fmt is used to communicate with the corresponding package.

fmt.Print (“Привет”) // output without newline

fmt.Println () // output with cursor to new line

fmt.Printf() // outputting variable values to text

fmt.fmt.Printf("Hello %d\n", 23) // print string and integer type

fmt.fmt.Fprint("Hello ", 23, "\n") //same with newline

fmt.fmt.Println("Hello", 23) // string and number output

fmt.Printf("This year I passed %v the cities, ", &output)

For example:

package main

import “fmt”

func main() {

    var name string

    var salary int

    fmt.Println("Enter last name ")

    fmt.Scanf("%s \n", &name)

    fmt.Println("Enter salary ")

    fmt.Scanf("%d \n", &salary)

    fmt.Printf("%s \n",name)

    fmt.Printf("%d \n",salary)

    fmt.Printf("Salary %s is %d dollars ", name, salary)

}

Outprt:

Enter last name

Smith

Enter salary

32000

Smith

32000

Зарплата Степанова составляет 32000 рублей

**2.5. Decision operators**

When solving a multitude of problems, the problem arises of choosing a further way of solving depending on some conditions. In the Go language, this possibility is implemented by the following constructions:

* selection by condition: if else; switch case; select;
* repetition of commands using iterations: for (range);
* change of behavior in the course of iterations: break and continue.

a). Conditional statement if-else

The Go language uses three decision-making constructs or, in other words, a choice of further computational process: if-else, switch and select. The syntax of the first operator is as follows:

if condition {

// executable code provided condition == true

} else {

executable code provided condition == false

}

For a single comparison, curly braces can be omitted. The opening bracket stays on the same line as the condition:

package main

import "fmt"

func main() {

// if the conditionis correct

if ID == "Apple" {

//what is done is what is in brackets:

fmt.Println("enter your login and password ")

/\* if the condition is not met, then you can immediately check another condition. You can do this any number of times \*/

} else if ID =="Google" {

// get the second answer

fmt.Println("Your operating system is not supported ")

// do what belongs to the last if

} else {

fmt.Println("Input error")

}

}

b). Switch statement

The switch expression switch provides a simple way to access different parts of the program based on the value of the expression. Syntax of switch statement:

switch optstatement; optexpression{

case expression1: Statement..

case expression2: Statement..

...

default: Statement..

}

After the first variant found, the operator performs the necessary actions and stops work.

switch ID {

// first value is checked

case "Apple":

fmt.Println("Enter your username and password ")

// second value is checked

case "Google":

fmt.Println("second value is checked ")

// if nothing was found

default:

fmt.Println("Input error ")

}

c). Select statement

The select statement allows you to select one of several expressions to execute. The main difference between select and switch is that select works on a standby basis. This means that the select command will not run until the message associated with sending and receiving on any channel is executed. It should be noted that the visual algorithmic language DRAKON in the implementation of the editor Drakon Web Editor uses a similar operator Select, which will be discussed in the next section.

d). Loops

The Golang language adopts only one loop format - *for*, which has the following variants.

1. The classical, C-like cycle with variable, condition and cycle step:

for i := 0; i < 8; i++ { loop body }

2. Preconditioned loop

The simplest loop is to declare only the condition, and place the rest inside the loop:

package main

import "fmt"

func main() {

// loop variable

var count = 10

// пока переменная больше 0 — цикл работает

for count > 0 {

// output the current value of the variable

fmt.Println(count)

// output the current value of the variable

count = count - 1

}

**2.6.. Loop within range**

There is another for loop variation that iterates the range of values for the data type. The range keyword is used in a cycle to iterate the elements of an array, slice or map. When using an array and a slice, the loop returns an index of the element as an integer. When using cards, it returns the key-value of the following pair:

package main

import "fmt"

func main() {

    nums := []int{2, 3, 4}

    sum := 0

    for \_, num := range nums {

        sum += num

    }

    fmt.Println("sum:", sum)

    for i, num := range nums {

        if num == 3 {

            fmt.Println("index:", i)

        }

    }

    kvs := map[string]string{"ca": "Paris", "co": "France"}

    for k, v := range kvs {

        fmt.Printf("%s -> %s\n", k, v)

    }

    for k := range kvs {

        fmt.Println("key:", k)

    }

}

The following table provides a complete list of Golang keywords:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | case | chan | const | continue |
| default | defer | else | fallthrough | for |
| func | go | goto | if | import |
| interface | map | package | range | return |
| select | struct | switch | type | var |

**2.7. Data types**

Data types define the types of values that are saved by variables when writing the program. Data types also help identify operations that can be performed using data. The following types of data types are dislocated.

* Fundamental types. This category includes numbers, strings, and logical (boolean) values.
* Aggregate types. This category includes array and structures.. struct
* Reference types. This category includes pointers, cuts, associative array (map) and, functions.
* Embedding types. Go supports user-defined types as aliases or structures.

2.5.1. Basic types

а). Целочисленные и вещественные типы переменных и констант

Integer Go types are divided into whole with a sign and whole without a sign. Integer types with the sign include int, int8, int16, int32, and int64; whole types without the sign include uint, uint8, uint16, uint32, uint64, uintptr. The range of values and amount of occupied memory of integer types are represented in the table. 2.1:

Табл. 2.1

| Тип | Range | Occupied computer memory |
| --- | --- | --- |
| int8 | –128 — 127 | 8 bit (1 bytes) |
| uint8 | 0 — 255 |
| int16 | –32 768 — 32 767 | 16 bit (2 bytes) |
| uint16 | 0 — 65535 |
| int32 | –2 147 483 648 — 2 147 483 647 | 32 bit (4 bytes) |
| uint32 | 0 — 4 294 967 295 |
| int64 | –9 223 372 036 854 775 808 — 9 223 372 036 854 775 807 | 64 bit (8 bytes) |
| uint64 | 0 — 18 446 744 073 709 551 615 |

The Go language adopts two floating point number formats:.

* float32 - The largest float32 is the constant math.MaxFloat32, which is about 3.4e38. The lowest positive value is 1.4e-45/.
* float64 - The largest float64 is the constant math.MaxFloat64, which is about 1.8e308. The smallest positive value is 4.9e-324.

Float32 provides about six decimal places of accuracy, while float64 provides about 15 digits. The float64 type is preferable for most problems, because when float32 is used, many iterative algorithms quickly accumulate a roundoff error.

To determine to which data type the Go compiler assigns a variable, the Printf() function is used, the parameters of which have a special symbol %T:

а := 234,45

fmt.Printf("Type %T for %v\n", a, a)

Output: Type float64 for 234.45

b). String variables and constants

A string is an immutable sequence of bytes. Strings may contain arbitrary data, including bytes with the value 0, but usually contain readable text. Text strings are usually interpreted as UTF-8-encoded sequences of code points. The built-in len function returns the number of bytes per string, and the print operator of the %x specification returns one byte of s for the Latin alphabet:

func main() {

    s := "Hello, World"

    fmt.Println("String >> ", s)

    fmt.Println("String lengh= ", len(s))

    fmt.Println("Hex bytes: ")

    for i := 0; i < len(s); i++ {

        fmt.Printf(" %x",s[i])

    }

    fmt.Println()

    q := " Hello, World "

    fmt.Println()

    fmt.Println("String >> ", q)

    fmt.Println("String length = ", len(q))

    for i := 0; i < len(q); i++ {

        fmt.Printf(" %x",q[i])

        }

Output:

String а >> Hello, World

String length = 12

Hex bytes:

48 65 6c 6c 6f 2c 20 57 6f 72 6c 64

String >> Hello, World

String length = 20

d0 9f d1 80 d0 b8 d0 b2 d0 b5 d1 82 2c 20 d0 9c d0 b8 d1 80

c) Logical variables and constants

The logical data type (bool) can be either true (truth) or false (falsehood). Boolean operators are used in programming to compare and control data flow:

func main() {

x := 5

     y := 8

    fmt.Println("x == y:", x == y)

    fmt.Println("x != y:", x != y)

    fmt.Println("x < y:", x < y)

     fmt.Println("x > y:", x > y)

    fmt.Println("x <= y:", x <= y)

    fmt.Println("x >= y:", x >= y)

    }

Output:

x != y: true

x < y: true

x > y: false

x <= y: true

x >= y: false

**2.8. Aggregate types**

а) Array

An array is a series of fixed-length data that is used to store homogeneous elements in memory. Arrays in Go are almost identical to arrays in other programming languages. Array elements are indexed using the [] index with their zero position, which means that the first element index is array[0] and the last element index is array[len(array)-1], where len(array) is the array length. However, due to the fixed length of arrays are not very popular unlike the Slice (Cut) design, which is incomparably used in Go. The syntax of the array looks like this:

[N]T{value1, value2, value3, ...value n}, где N – numbers of elements

b) Structure

Structure design is the type of data defined by the developer and serves to represent any real objects. Structures contain a set of fields that represent different attributes of an object. Type and struct keywords are used to define the structure:

type *s* *name*\_ struct{

    structures fields

}

The following is an example of an enterprise employee data structure:package main

import "fmt"

type Employee struct {

firstName, lastName, address string // string type fields

age, phone, salary int // integer type fields

}

func main() {

// structure initialization

|  |
| --- |
| emp := Employee{firstName: "Max", lastName: "Smith", age: 42, phone: 123456789, salary: 34000, address: "  Amarillo |

"}

fmt.Println("first and last names: ", emp.firstName, emp.lastName)

fmt.Println("Employee age: ", emp.age)

fmt.Println("Employee salary: ", emp.salary)

fmt.Println("Employee telephone: ", emp.phone)

fmt.Println("Employee address: ", emp.address)

}

The memory for a new struct variable is allocated using the new function, which returns a pointer to the allocated memory:

var w \*T = new(T),

or in different lines if the structure is declared in the package area:

var w \*T

w = new(T)

When using the short form of assigning a variable value (:=), that is, w := new(T), the variable z is a pointer to a memory area where the structure fields contain null values according to their types:

type structEmpl struct {

name string

age int

salary float64

}

func main() {

 ms := new(structEmpl)

 fmt.Println(ms)

 ms.name = "Fangur"

 ms.age = 45

 ms.salary = 1200.5

 //fmt.Printf("Last name: %s\n", ms.name)

 fmt.Printf("Age: %d\n", ms.age)

 fmt.Printf("Salary: %8.1f\n", ms.salary)

 fmt.Println(ms)

}

**2.9. Refetence types**

а) Pointer

Pointers in the Golang programming language are variables that are used to store the memory address of another variable. Variables are used to store some data at a specific memory address in the system. The memory address is always represented in hexadecimal format (starting from 0x, for example, 0xFFAAF, etc.) (Figure 2.2):

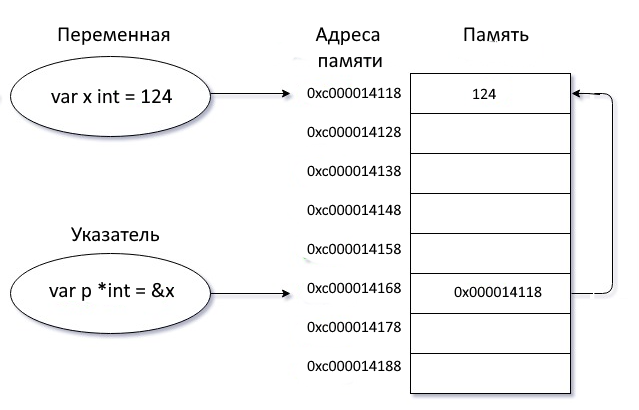


Рис. 2.2. Переменная, указатель, память

The pointer is usually called a special type of variable. The main and only operators of the pointers are: the dereferential operator (\*) and the address operator (&). The dereferencing operator (\*) is used to declare a pointer variable and access the value stored in the address. The address operator (&) is used to return the address of the variable or to access the address of the variable index.Синтаксис объявления указателя:

var pointer\_name \*Data\_Type,

где Data\_Type - any valid data type, for example, var pos \*string.

To work with the pointer, it must be initialized with the memory address of another variable using the &address statement, as shown in the following example:

var a = 124

var s \*int = &a

The uninitialized index will always have a zero value of <nil>. Here is an example of index initialization and processing:func main() {

var z1, z2 int = 64, 128

var p1, p2 \*int

p1 = &z1 // p1 pointer initialized

fmt.Println("Variable value z1 = ", z1)

fmt.Println("Address z1 = ", &z1)

fmt.Println("Variable value z2 = ", z2)

fmt.Println("Address z2 = ", &z2)

// p2 pointer is uninitialized

fmt.Println("The value saved in the variable p1 = ", p1)

fmt.Println("The value saved in the variable p2 = ", p2)

}

Output:

Variable value z1 = 64

Address z1 = 0xc000086080

Variable value z2 = 128

Address z2 = 0xc000086088

The value saved in the variable p1 = 0xc000086080

The value saved in the variable p2 = <nil>

The Dereference Statement \* is used to declare a pointer variable and to access the value stored in the variable to which the pointer points:

func main() {

    var y = 157

    var p = &y

    fmt.Println("Variable value in y = ", y)

    fmt.Println("Variable address y = ", &y)

    fmt.Println("Value saved in p = ", p)

    fmt.Println("Value saved in  y(\*p) = ", \*p)

}

Output:

Variable value in y = 157

Variable address y = 0xc000014078

Value stored in p = 0xc000014078

Value stored in y(\*p) = 157б)

b) Slice

Slices are sequences of variable length, all of whose elements are of the same type. The slice type [] T is written, where the elements are of type T.

There are three components:

*Pointer*: The pointer points to an array element that is accessible through a cut, which is not necessarily the first element of the array.

*Length*: the number of slice t elements. It cannot exceed the capacity.

*Capacitance*: The capacitance is usually the number of elements between the beginning of the slice and the end of the base array representing the change.

The built-in len and cap functions contain slice length and capacitance values respectively. Multiple slices can share the same base array and can refer to overlapping parts of that array. The first position of the index in the slice is always 0, and the last is the slice length minus one (len- 1). The slice declared in the same way as the array, but it does not contain a slice size. Thus, it can grow or decrease according to some algorithm.

The syntax of the cut design is as follows:

[]T или []T{} или []T{value1, value2, value3, ...value n},

here T is the element type. For example:

var my\_slice [] int

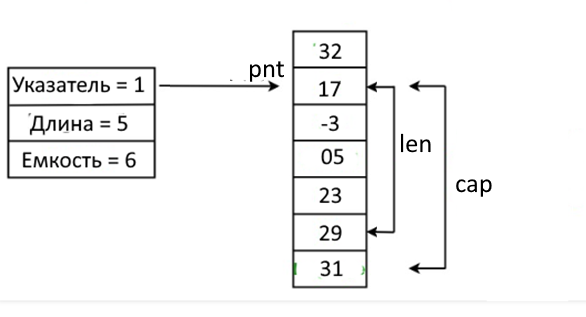
The figure (Figure 2.3) shows the visual representation of the cut through its components - pointer (ptr), length (len) and capacitance (cap): **

Figure.2.3. Representation of slicing in memory through components

Slices can be created from existing slices with corresponding indices (Figure 2.4):

package main

import (

    "fmt"

)

func main() {

    var sr = []int{17, -21, 5, 62, 24, 48, 78, -43}

    sr0 := sr[2:6]

    fmt.Println("sr0 = ", sr0)

    sr1 := sr[2:]

    fmt.Println("sr1 = ", sr1) // prints [5 7 9 11 13 15]

    sr2 := sr1[:3]

    fmt.Println("sr2 = ", sr2)

}

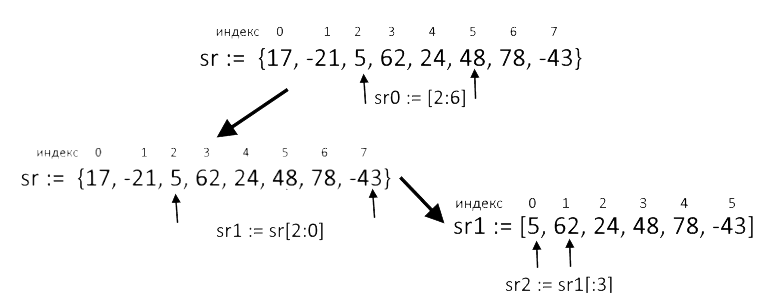


Figure 2.4. Create new slice from existing slice

It is important to note that each created cut sr0, sr1, sr2 has its own memory, that is, all created cuts have different addresses, even if they are fragments of one slice.:

sr --> 0xc000004078; sr0 --> 0xc000004090; sr1 --> 0xc0000040c0;

sr2 > 0xc0000040f0

2.7. Associative arrays (Map)

Associative arrays (later cards) are storage containers of the pair "key-value". Maps are one of the most common and useful data structures. Unlike data structures such as arrays or cut-offs, maps provide quick and efficient searches for elements. The map does not allow duplicate keys, but may have duplicate values.

The announcement syntax of the card with the key type int and the value - string:

var names map[int]string

The card can be initialized in two ways:Using the function *make()*: var names = make(map[int]string);

Using literal syntax: var films = map[int]string;

Here is an example illustrating both methods of initialization of the map:

func main() {

  names := map[int]string{0: " [Cameron](https://www.imdb.com/name/nm0000116?ref_=nmls_hd)", 1: "Spielberg", 2: "Lee", 3: " Donner"}

    fmt.Println(names)

    films := make(map[string]string)

    films["Жаров"] = " The Abyss"

    films["Лавров"] = " [The Goonies](https://en.wikipedia.org/wiki/The_Goonies)"

    films["Ульянов"] = "Jungle Fever"

    films["Матвеев"] = "Superman"

    fmt.Sprintln(films)

    for key, value := range films {

        fmt.Println(" \n", key, “>”,value)

    }

}

}

Output:

[Cameron](https://www.imdb.com/name/nm0000116?ref_=nmls_hd) > The Abyss

Spielberg > [The Goonies](https://en.wikipedia.org/wiki/The_Goonies)

Lee > Jungle Fever

Donner > Superman

**2.10. Function**

A function is a group of operators that share a task. With the help of functions you can repeatedly call its operator block as a unit in other parts of the program. Functions in Go can be assigned to variables, passed as an argument, and can be returned from another function. When declaring a function, you need to specify what type of variables are transferred to the function(s) and what type of data the function returns (return values). In Go, you must specify the data type for each parameter. To declare a function, the keyword func is used. The general structure of the function declaration is shown below:

func name  > [Parameters] ) [Returned types]

{

Function body

}

Функция умножения двух вещественных чисел multiply(x,y) выглядит следующим образом:

package main

import "fmt"

func multiply(x,y float64) float64 {

var res

res = x \* y

return res // return result

}

func main() { // function performed

var x float64 = 15

var y float64 = 120.6

var mult float64 // variable for result

mult = multiply(x,y) // function calculation

fmt.Print(mult) // outputting

}

The function can return as many values as required:func main() {

    var a, b int = 60, 20

    vAdd, vSub := addSub(a, b)

    fmt.Printf("a + b = %d\n", vAdd) // prints "35 + 25 = 60"

    fmt.Printf("a - b = %d\n", vSub) // prints "35 - 25 = 10"

}

func addSub(x, y int) (int, int) { // multiple return values (int, int)

    return x + y, x - y

}

An important feature of functions in Go is the ability to accept parameters by value or by reference [].

**2.11. Methods**

Methods together with functions provide different ways of organizing program code. A method in Go is a function associated with a particular type that acts on a variable of a certain type called a receiver (recipient). The Golang language does not support the Class concept common in other languages. It is with the method that the object provides its properties, including the behavior of the object. The method must meet the following conditions: it must be of a certain type and must be defined in the same package.

The method declaration looks like a function declaration, but it has an additional part of the parameter declaration. An optional parameter can contain one and only one parameter of the method recipient type. The recipient parameter should be enclosed in parentheses between the func keyword and the method name.

Func (*Receiver name* Type) *Method name* (Parameters)(Return type){

// Operator block

}

In Go, it is allowed to define a method whose receiver has a structure type. This receiver is available inside the method as shown in the following example. The rec receiver structure describing the rect object is declared:

type rect struct {

var width, height int

}

Two methods are associated with this structure that determine the behavior of the object described by the structure *rect*:

|  |
| --- |
| func (r rect) area() int {  return r.width \*r.height  } |
| func (r rect) perim() int {  return 2\*r.width + \*r.height  } |
|  |

**2.12. Interface type**

Golang is not a classical object-oriented language, that is, it does not support the implementation of the concept of "classes" and "inheritance" in a direct way. However, Go contains a very flexible concept of interfaces that provides many aspects of object-oriented programming.

Interfaces in Go provide a way to indicate the behavior of an object by a set of methods defined by the interface type. An interface-type variable can store any type of value with a set of methods, which is any superset of the interface. The concept of interface allows to organize different groups of methods applied to objects of different nature. In other words, interfaces are collections of method signatures declaring name, type parameters, and return types of methods in the interface.

The syntax of the interface is as follows:type Namer interface { // Namer – interface type

Method1(param\_list) return\_type

Method2(param\_list) return\_type

For example, you need to create a method to determine the area of geometric shapes - circle and rectangle. type shape interface {

    area() float64

    perimeter() float64

}

This code defines the interface for shapes and declares two functions, area() and perimeter() with a return float64 type.

package main

import "fmt"

// Interface declarationtype shape interface {

    area() float64

    perimeter() float64

}

// Outline declaration "rectangle"type rectangle struct{

  length, height float64

}

// Declaration of the "circle structure

type circle struct{

  radius float64

}

// Declaring Methods for a Rectanglefunc (r rectangle) area() float64 {

    return r.length \* r.height

}

func (r rectangle) perimeter() float64 {

    return 2 \* r.length + 2 \* r.height

}

// Declaration of methods for the circle

func (c circle) area() float64 {

    return 3.142 \* c.radius \* c.radius

}

func (c circle) perimeter() float64 {

    return 2 \* 3.142 \* c.radius

}

func main() {

    r := rectangle{length: 10.0, height: 5.0}

    c := circle{radius: 5.0}

    fmt. Printf("Rectangle area - %8.1f n", r.area()) fmt.Printf("Rectangle parimeter - %8.1f n", r.perimeter()) fmt.Printf("Circle area - %8.1f n", c.area()) fmt.Printf("Circle Parimeter - %8.1f n", c.perimeter())

}

Output:

Rectangle Area - 50.0Rectangle Parimeter - 30.0Circle area - 78.5Lap Circummetre - 31

**Conclusion**

It should be noted once again that this section provides basic information on the programming language Golang, sufficient to understand the material set out later. Outside the section such information as Go-subroutines and channels, parallelism and shared variables, packages and tools go, reflection, etc. Next, in the relevant sections will explain in more detail the tools and constructions, as set out in this section. In addition, the recursion tool will be explained in detail because of its special value for iterative processes in data structure processing algorithms.

In addition, it should be noted that in this book, which implements the concept of hybrid programming, Linear constructions of the Go language are used as filling icons of the visual algorithmic language DRAKON. Other designs, in particular, cycles in the range or operator Select is different from Go-designs. Moreover, some DRAKON language designs are converted into other designs during the automatic generation of code. In particular, the software implementation of the Select statement is converted into a composite if-else choice operator. This is due to the desire of the authors of the DRAKON language to make the generated code more efficient, increasing its speed and saving computer memory.