--Progrank

WHY GO?

-Memory Management

-Security

-Slow Processing

Above issues in language like C,CPP.

-Solved by GO language and uses tools like Docker and Kubernetes.

-Multi-purpose languge

-compiled language

-static and strongly typed

-faster compilation

-simple and readable

-object document like

-pointers

-open source community

Difference Between var and :=

There are some small differences between the var var :=:

|  |  |
| --- | --- |
| **var** | **:=** |
| Can be used **inside**and **outside** of functions | Can only be used **inside** functions |
| Variable declaration and value assignment **can be done separately** | Variable declaration and value assignment **cannot be done separately** (must be done in the same line) |

Constant Rules

* Constant names follow the same naming rules as [variables](https://www.w3schools.com/go/go_variable_naming_rules.php)
* Constant names are usually written in uppercase letters (for easy identification and differentiation from variables)
* Constants can be declared both inside and outside of a function

Constant Types

There are two types of constants:

* Typed constants
* Untyped constants

# Go Output Functions

Go has three functions to output text:

* Print()
* Println()
* Printf()

The Print() Function

The Print() function prints its arguments with their default format.

## The Println() Function

The Println() function is similar to Print() with the difference that a whitespace is added between the arguments, and a newline is added at the end

The Printf() Function

The Printf() function first formats its argument based on the given formatting verb and then prints them.

Here we will use two formatting verbs:

* %v is used to print the **value** of the arguments
* %T is used to print the **type** of the arguments

## Formatting Verbs for Printf()

Go offers several formatting verbs that can be used with the Printf() function.

## General Formatting Verbs

The following verbs can be used with all data types:

|  |  |
| --- | --- |
| **Verb** | **Description** |
| %v | Prints the value in the default format |
| %#v | Prints the value in Go-syntax format |
| %T | Prints the type of the value |
| %% | Prints the % sign |

Go Data Types

Data type is an important concept in programming. Data type specifies the size and type of variable values.

Go is statically typed, meaning that once a variable type is defined, it can only store data of that type.

Go has three basic data types:

* **bool**: represents a boolean value and is either true or false
* **Numeric**: represents integer types, floating point values, and complex types
* **string**: represents a string value

Go Integer Data Types

Integer data types are used to store a whole number without decimals, like 35, -50, or 1345000.

The integer data type has two categories:

* **Signed integers** - can store both positive and negative values
* **Unsigned integers** - can only store non-negative values
* Go has five keywords/types of signed integers:

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| int | Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems | -2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems |
| int8 | 8 bits/1 byte | -128 to 127 |
| int16 | 16 bits/2 byte | -32768 to 32767 |
| int32 | 32 bits/4 byte | -2147483648 to 2147483647 |
| int64 | 64 bits/8 byte | -9223372036854775808 to 9223372036854775807 |
| Type | Size | Range |
| uint | Depends on platform: 32 bits in 32 bit systems and 64 bit in 64 bit systems | 0 to 4294967295 in 32 bit systems and 0 to 18446744073709551615 in 64 bit systems |
| uint8 | 8 bits/1 byte | 0 to 255 |
| uint16 | 16 bits/2 byte | 0 to 65535 |
| uint32 | 32 bits/4 byte | 0 to 4294967295 |
| uint64 | 64 bits/8 byte | 0 to 18446744073709551615 |

The float data types are used to store positive and negative numbers with a decimal point, like 35.3, -2.34, or 3597.34987.

The float data type has two keywords:

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| float32 | 32 bits | -3.4e+38 to 3.4e+38. |
| float64 | 64 bits | -1.7e+308 to +1.7e+308. |

**Tip:** The default type for float is float64. If you do not specify a type, the type will be float64.

## Go Arrays

Arrays are used to store multiple values of the same type in a single variable, instead of declaring separate variables for each value.

## Declare an Array

In Go, there are two ways to declare an array:

#### **1. With the var keyword:**

### **Syntax**

var array\_name = [length]datatype{values} // here length is defined  
  
or  
  
var array\_name = [...]datatype{values} // here length is inferred

#### **2. With the := sign:**

### **Syntax**

array\_name := [length]datatype{values} // here length is defined  
  
or  
  
array\_name := [...]datatype{values} // here length is inferred

**Note:** The length specifies the number of elements to store in the array. In Go, arrays have a fixed length. The length of the array is either defined by a number or is inferred (means that the compiler decides the length of the array, based on the number of values).

## Access Elements of an Array

You can access a specific array element by referring to the index number.

In Go, array indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

## Change Elements of an Array

You can also change the value of a specific array element by referring to the index number.

## Array Initialization

If an array or one of its elements has not been initialized in the code, it is assigned the default value of its type.

**Tip:** The default value for int is 0, and the default value for string is "".

## Initialize Only Specific Elements

It is possible to initialize only specific elements in an array.

arr1 := [5]int{1:10,2:40}  
  fmt.Println(arr1)

The array above has 5 elements.

* 1:10 means: assign 10 to array index 1 (second element).
* 2:40 means: assign 40 to array index 2 (third element).

Go Slices

Slices are similar to arrays, but are more powerful and flexible.

Like arrays, slices are also used to store multiple values of the same type in a single variable.

However, unlike arrays, the length of a slice can grow and shrink as you see fit.

In Go, there are several ways to create a slice:

* Using the []*datatype*{*values*} format
* Create a slice from an array
* Using the make() function

In Go, there are two functions that can be used to return the length and capacity of a slice:

* len() function - returns the length of the slice (the number of elements in the slice)
* cap() function - returns the capacity of the slice (the number of elements the slice can grow or shrink to)

# Go Access, Change, Append and Copy Slices

## Access Elements of a Slice

You can access a specific slice element by referring to the index number.

In Go, indexes start at 0. That means that [0] is the first element, [1] is the second element, etc.

## Change Elements of a Slice

You can also change a specific slice element by referring to the index number.

## Append Elements To a Slice

## You can append elements to the end of a slice using the append()function

## Append One Slice To Another Slice

To append all the elements of one slice to another slice, use the append()function:

## **Note:** The **'...'** after slice2 is **necessary** when appending the elements of one slice to another.

## Change The Length of a Slice

Unlike arrays, it is possible to change the length of a slice.

## Memory Efficiency

 When using slices, Go loads all the underlying elements into the memory.

If the array is large and you need only a few elements, it is better to copy those elements using the copy() function.

The copy() function creates a new underlying array with only the required elements for the slice. This will reduce the memory used for the program.

## Go Maps

Maps are used to store data values in key:value pairs.

Each element in a map is a key:value pair.

A map is an unordered and changeable collection that does not allow duplicates.

The length of a map is the number of its elements. You can find it using the len() function.

The default value of a map is nil.

Maps hold references to an underlying hash table.

Go has multiple ways for creating maps.

## The order of the map elements defined in the code is different from the way that they are stored. The data are stored in a way to have efficient data retrieval from the map.

## Creating an Empty Map

There are two ways to create an empty map. One is by using the make()function and the other is by using the following syntax.

## The make()function is the right way to create an empty map. If you make an empty map in a different way and write to it, it will causes a runtime panic.

## Allowed Key Types

The map key can be of any data type for which the equality operator (==) is defined. These include:

* Booleans
* Numbers
* Strings
* Arrays
* Pointers
* Structs
* Interfaces (as long as the dynamic type supports equality)

Invalid key types are:

* Slices
* Maps
* Functions

These types are invalid because the equality operator (==) is not defined for them.

## Allowed Value Types

The map values can be **any** type.

## Accessing Map Elements

You can access map elements by

## Updating and Adding Map Elements

## Remove Element from Map

Removing elements is done using the delete() function.

## Check For Specific Elements in a Map

You can check if a certain key exists in a map using:

## Maps Are References

Maps are references to hash tables.

If two map variables refer to the same hash table, changing the content of one variable affect the content of the other.

## Iterating Over Maps

You can use range to iterate over maps.

## Iterate Over Maps in a Specific Order

Maps are unordered data structures. If you need to iterate over a map in a specific order, you must have a separate data structure that specifies that order.