Collections

Overview

Arrays

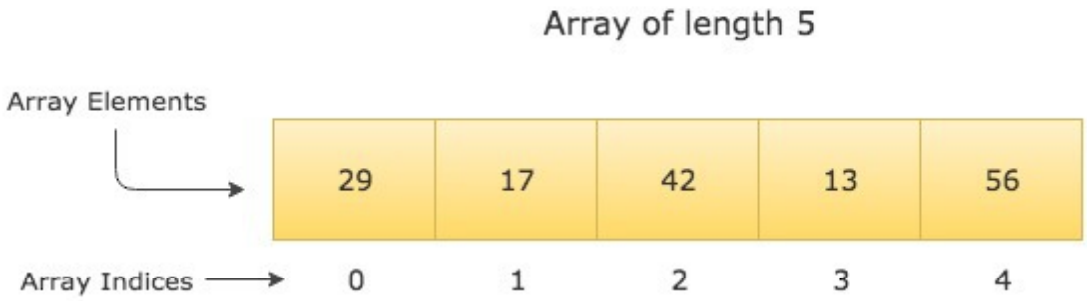
Slices

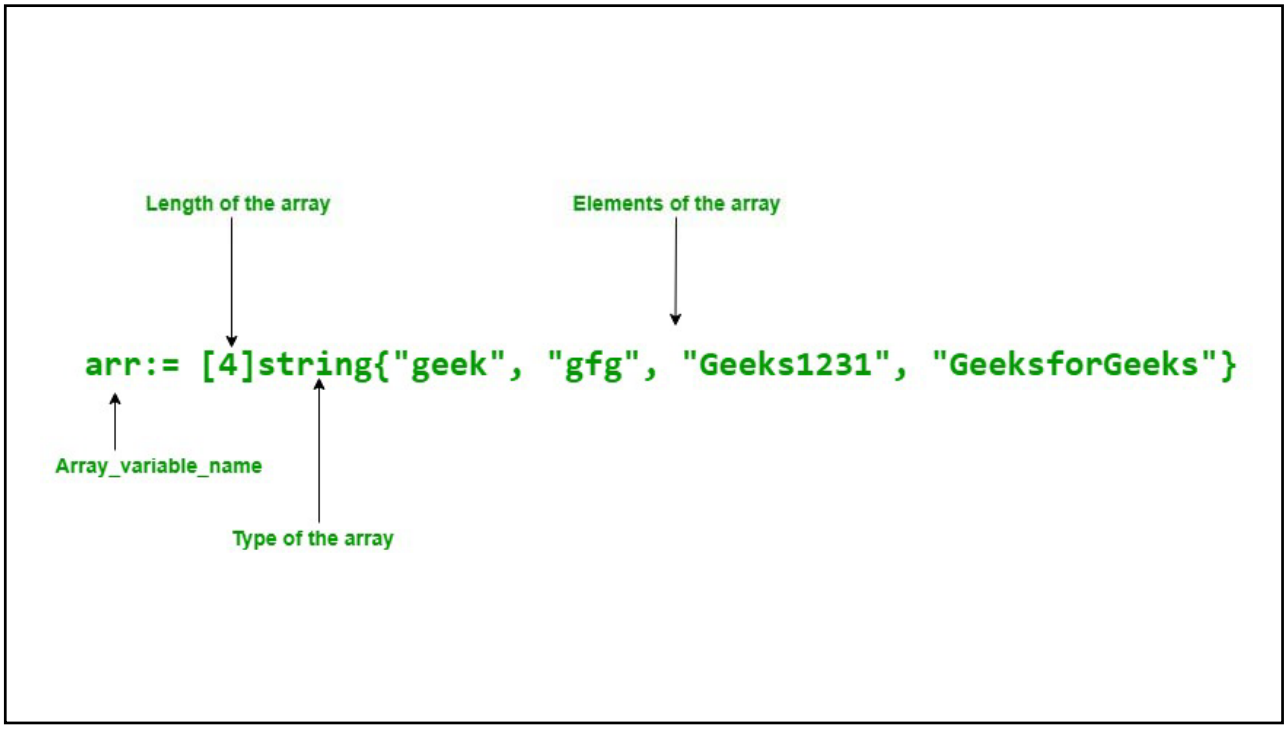
Maps

Structs

Working With Collections

ARRAYS






```go
package main
import "fmt"

func main() {
    var x [5]int // An array of 5 integers
    fmt.Println(x)

    var y [8]string // An array of 8 strings
    fmt.Println(y)
```

```
# Output
[0 0 0 0 0]
[        ]
```

```go
package main
import "fmt"

func main() {
    var x [5]int // An array of 5 integers

    x[0] = 100
    x[1] = 101
    x[3] = 103
    x[4] = 105

    fmt.Printf("x[0] = %d, x[1] = %d, x[2] = %d\n", x[0], x[1], x[2])
    fmt.Println("x = ", x)
}
```

```
# Output
x[0] = 100, x[1] = 101, x[2] = 0
x =  [100 101 0 103 105]
```

```go
// Declaring and initializing an array at the same time
var a = [5]int{2, 4, 6, 8, 10}
```

```go
// Short hand declaration
a := [5]int{2, 4, 6, 8, 10}
```

```go
// Letting Go compiler infer the length of the array
a := [...]int{3, 5, 7, 9, 11, 13, 17}
```

Working With Collections

SLICES

Introduction to Slices

A Slice is a segment of an array. Slices build on arrays and provide more power, flexibility, and convenience compared to arrays.

Just like arrays, Slices are indexable and have a length. But unlike arrays, they can be resized.

Declaring a Slice

A slice of type `T` is declared using `[]T`. For example, Here is how you can declare a slice of type `int` -

```
// Slice of type `int`
var s []int
```

The slice is declared just like an array except that we do not specify any size in the brackets `[]`.

1. Creating a slice using a slice literal

You can create a slice using a slice literal like this -

```
// Creating a slice using a slice literal
var s = []int{3, 5, 7, 9, 11, 13, 17}
```

The expression on the right-hand side of the above statement is a slice literal. The slice literal is declared just like an array literal, except that you do not specify any size in the square brackets `[]`.

2. Creating a slice from an array

Since a slice is a segment of an array, we can create a slice from an array.

To create a slice from an array `a`, we specify two indices `low` (lower bound) and `high` (upper bound) separated by a colon -

```
// Obtaining a slice from an array `a`
a[low:high]
```

The above expression selects a slice from the array `a`. The resulting slice includes all the elements starting from index `low` to `high`, but excluding the element at index `high`.

```
package main
import "fmt"

func main() {
    var a = [5]string{"Alpha", "Beta", "Gamma", "Delta", "Epsilon"}

    // Creating a slice from the array
    var s []string = a[1:4]

    fmt.Println("Array a = ", a)
    fmt.Println("Slice s = ", s)
}
```

```
Array a =  [Alpha Beta Gamma Delta Epsilon]
Slice s =  [Beta Gamma Delta]
```

```go
package main
import "fmt"

func main() {
    a := [5]string{"C", "C++", "Java", "Python", "Go"}

    slice1 := a[1:4]
    slice2 := a[:3]
    slice3 := a[2:]
    slice4 := a[:]

    fmt.Println("Array a = ", a)
    fmt.Println("slice1 = ", slice1)
    fmt.Println("slice2 = ", slice2)
    fmt.Println("slice3 = ", slice3)
    fmt.Println("slice4 = ", slice4)
}
```

```
# Output
Array a =  [C C++ Java Python Go]
slice1 =  [C++ Java Python]
slice2 =  [C C++ Java]
slice3 =  [Java Python Go]
slice4 =  [C C++ Java Python Go]
```

3. Creating a slice from another slice

A slice can also be created by slicing an existing slice.

```go
package main
import "fmt"

func main() {
    cities := []string{"New York", "London", "Chicago", "Beijing", "Delhi", "Mumbai", "Bangalore", "Hyderabad", "Hong Kong"}

    asianCities := cities[3:]
    indianCities := asianCities[1:5]

    fmt.Println("cities = ", cities)
    fmt.Println("asianCities = ", asianCities)
    fmt.Println("indianCities = ", indianCities)
}
```

```
# Output
cities =  [New York London Chicago Beijing Delhi Mumbai Bangalore Hyderabad Hong Kong]
asianCities =  [Beijing Delhi Mumbai Bangalore Hyderabad Hong Kong]
indianCities =  [Delhi Mumbai Bangalore Hyderabad]
```

Working With Collections

MAPS

Introduction to Map

A map is an unordered collection of key-value pairs. It maps keys to values. The keys are unique within a map while the values may not be.

The map data structure is used for fast lookups, retrieval, and deletion of data based on keys. It is one of the most used data structures in computer science.

Declaring a map

A map is declared using the following syntax -

```
var m map[KeyType]ValueType
```

For example, Here is how you can declare a map of `string` keys to `int` values -

```
var m map[string]int
```

The zero value of a map is `nil` . A `nil` map has no keys. Moreover, any attempt to add keys to a `nil` map will result in a runtime error.

1. Initializing a map using the built-in `make()` function

You can initialize a map using the built-in `make()` function. You just need to pass the type of the map to the `make()` function as in the example below. The function will return an initialized and ready to use map -

```
// Initializing a map using the built-in make() function
var m = make(map[string]int)
```

```go
package main
import "fmt"

func main() {
    var m = make(map[string]int)

    fmt.Println(m)

    if m == nil {
        fmt.Println("m is nil")
    } else {
        fmt.Println("m is not nil")
    }

    // make() function returns an initialized and ready to use map.
    // Since it is initialized, you can add new keys to it.
    m["one hundred"] = 100
    fmt.Println(m)
}
```

```
# Output
map[]
m is not nil
map[one hundred:100]
```

2. Initializing a map using a map literal

A map literal is a very convenient way to initialize a map with some data. You just need to pass the key-value pairs separated by colon inside curly braces like this -

```go
var m = map[string]int{
    "one": 1,
    "two": 2,
    "three": 3,
}
```

Note that the last trailing comma is necessary, otherwise, you'll get a compiler error.

```go
package main
import "fmt"

func main() {
    var m = map[string]int{
        "one":   1,
        "two":   2,
        "three": 3,
        "four":  4,
        "five":  5, // Comma is necessary
    }

    fmt.Println(m)
}
```

```
# Output
map[one:1 two:2 three:3 four:4 five:5]
```

The following example initializes a map using the `make()` function and adds some new items to it -

```go
package main
import "fmt"

func main() {
    // Initializing a map
    var tinderMatch = make(map[string]string)

    // Adding keys to a map
    tinderMatch["Rajeev"] = "Angelina" // Assigns the value "Angelina" to the key "Rajeev"
    tinderMatch["James"] = "Sophia"
    tinderMatch["David"] = "Emma"

    fmt.Println(tinderMatch)

    /*
      Adding a key that already exists will simply override
      the existing key with the new value
    */
    tinderMatch["Rajeev"] = "Jennifer"
    fmt.Println(tinderMatch)
}
```

```
# Output
map[Rajeev:Angelina James:Sophia David:Emma]
map[Rajeev:Jennifer James:Sophia David:Emma]
```

You can retrieve the value assigned to a key in a map using the syntax `m[key]`. If the key exists in the map, you'll get the assigned value. Otherwise, you'll get the zero value of the map's value type.

Let's check out an example to understand this -

```
package main
import "fmt"

func main() {
    var personMobileNo = map[string]string{
        "John":  "+33-8273658526",
        "Steve": "+1-8579822345",
        "David": "+44-9462834443",
    }

    var mobileNo = personMobileNo["Steve"]
    fmt.Println("Steve's Mobile No : ", mobileNo)

    // If a key doesn't exist in the map, we get the zero value of the value type
    mobileNo = personMobileNo["Jack"]
    fmt.Println("Jack's Mobile No : ", mobileNo)
}
```

```
# Output
Steve's Mobile No :  +1-8579822345
Jack's Mobile No :
```

Working With Collections

STRUCTS

Introduction to Struct

A `struct` is a user-defined type that contains a collection of named fields/properties. It is used to group related data together to form a single unit. Any real-world entity that has a set of properties can be represented using a struct.

Defining a struct type

You can define a new `struct` type like this -

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

The `type` keyword introduces a new type. It is followed by the name of the type (`Person`) and the keyword `struct` to indicate that we're defining a `struct` . The struct contains a list of `fields` inside the curly braces. Each field has a name and a type.

Note that, you can collapse fields of the same type like this:

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

Declaring a variable of a struct type

Just like other data types, you can declare a variable of a `struct` type like this -

```
// Declares a variable of type 'Person'
var p Person // All the struct fields are initialized with their zero value
```

Initializing a struct

You can initialize a variable of a `struct` type using a struct literal like so -

```
// Initialize a struct by supplying the value of all the struct fields.
var p = Person{"Rajeev", "Singh", 26}
```

Note that you need to pass the field values in the same order in which they are declared in the `struct`. Also, you can't initialize only a subset of fields with the above syntax -

```
var p = Person{"Rajeev"} // Compiler Error: too few values in struct initializer
```

Naming fields while initializing a struct

Go also supports the `name: value` syntax for initializing a struct (the order of fields is irrelevant when using this syntax).

```
var p = Person{FirstName: "Rajeev", LastName: "Singh", Age: 25}
```

You can separate multiple fields by a new line for better readability (the trailing comma is mandatory in this case) -

```
var p = Person{
    FirstName: "John",
    LastName:  "Snow",
    Age:       45,
}
```

Accessing fields of a struct

```
type Car struct {
    Name, Model, Color string
    WeightInKg         float64
}

func main() {
    c := Car{
        Name:       "Ferrari",
        Model:      "GTC4",
        Color:      "Red",
        WeightInKg: 1920,
    }

    // Accessing struct fields using the dot operator
    fmt.Println("Car Name: ", c.Name)
    fmt.Println("Car Color: ", c.Color)

    // Assigning a new value to a struct field
    c.Color = "Black"
    fmt.Println("Car: ", c)
```

```
# Output
Car Name:  Ferrari
Car Color:  Red
Car:  {Ferrari GTC4 Black 1920}
```

Summary

- Arrays
- Slices
- Maps
- Structs