Primitive Data Types

Overview

Declaring Variable

Basic Types, Operators and Type Conversion

Constants, Iota

Pointers

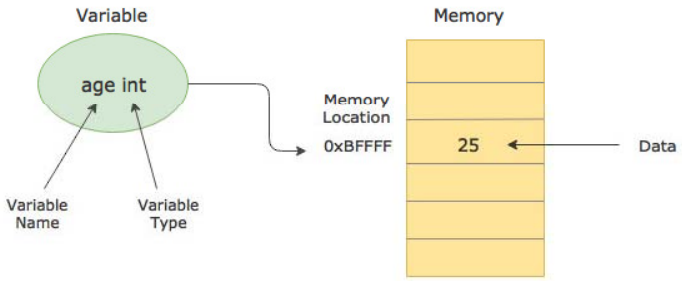
Working with Primitive Data Types

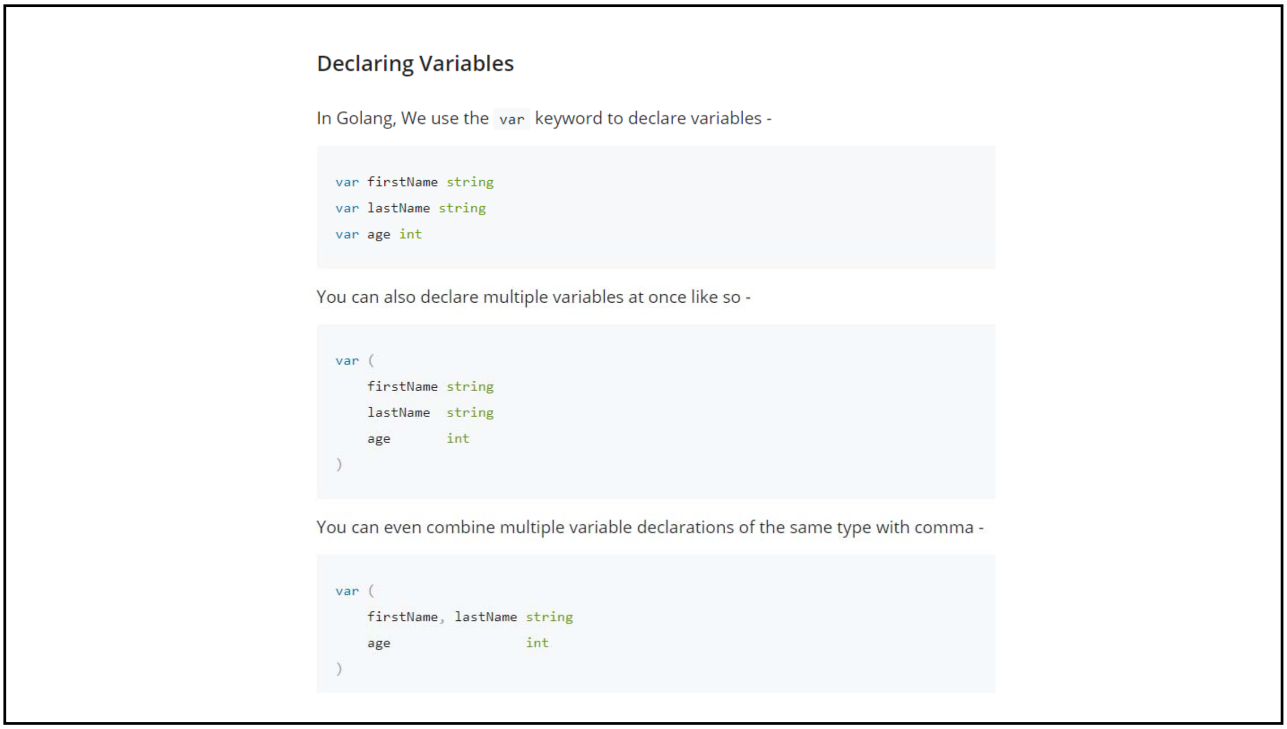
DECLARING VARIABLE

Introduction to Variables

Every program needs to store some data/information in memory. The data is stored in memory at a particular memory location.

A variable is just a convenient name given to a memory location where the data is stored. Apart from a name, every variable also has an associated type.





Declaring Variables

In Golang, We use the `var` keyword to declare variables -

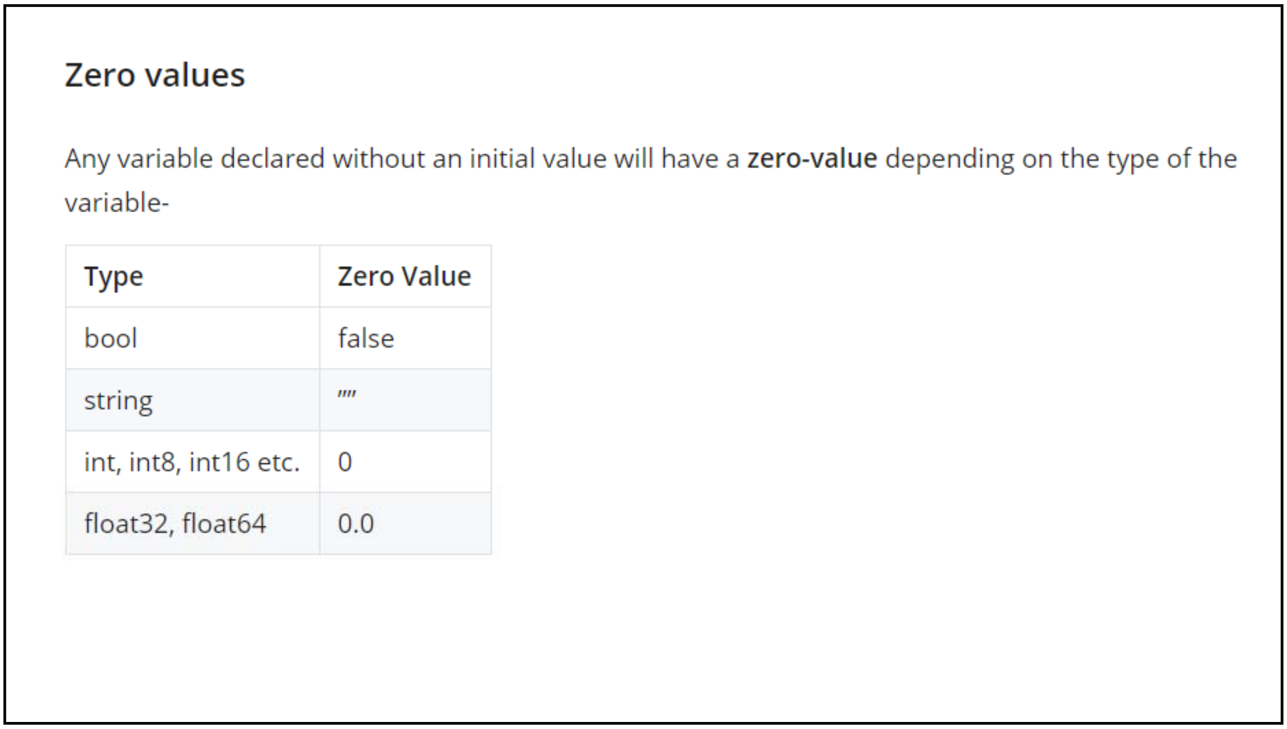
```
var firstName string
var lastName string
var age int
```

You can also declare multiple variables at once like so -

```
var (
    firstName string
    lastName  string
    age       int
)
```

You can even combine multiple variable declarations of the same type with comma -

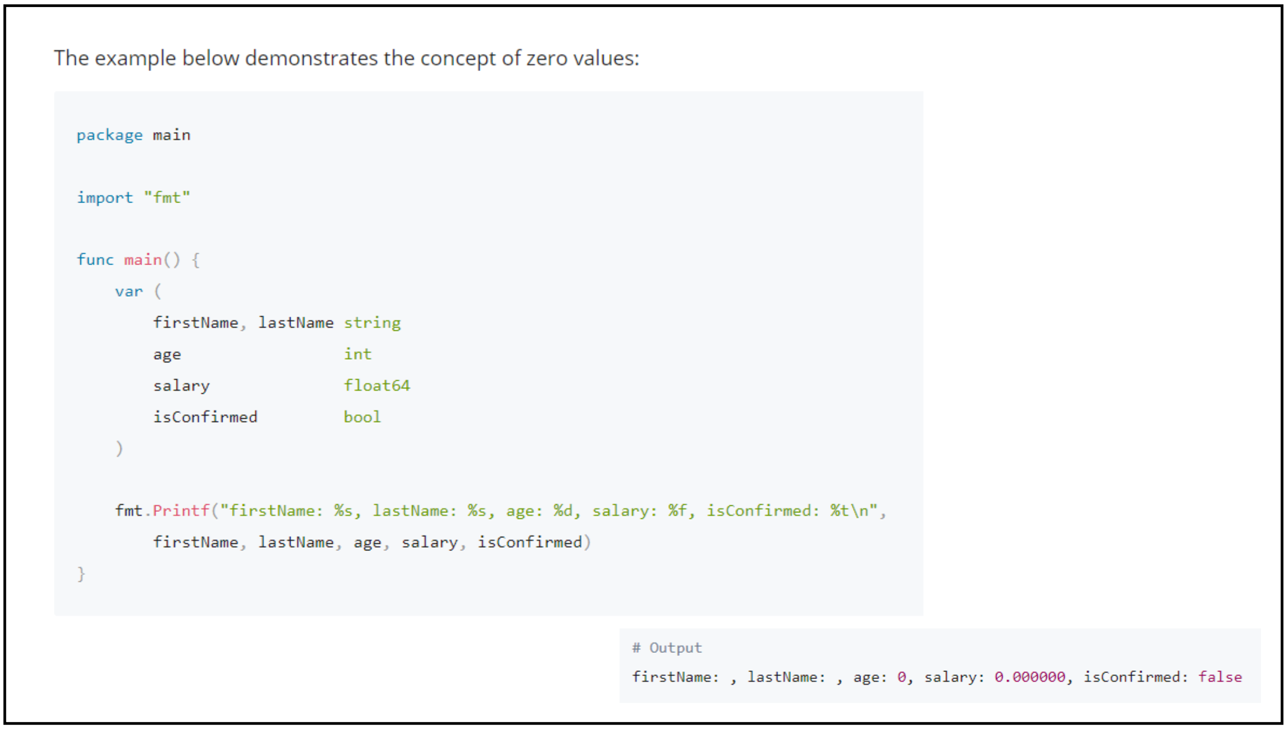
```
var (
    firstName, lastName string
    age                 int
)
```



Zero values

Any variable declared without an initial value will have a **zero-value** depending on the type of the variable-

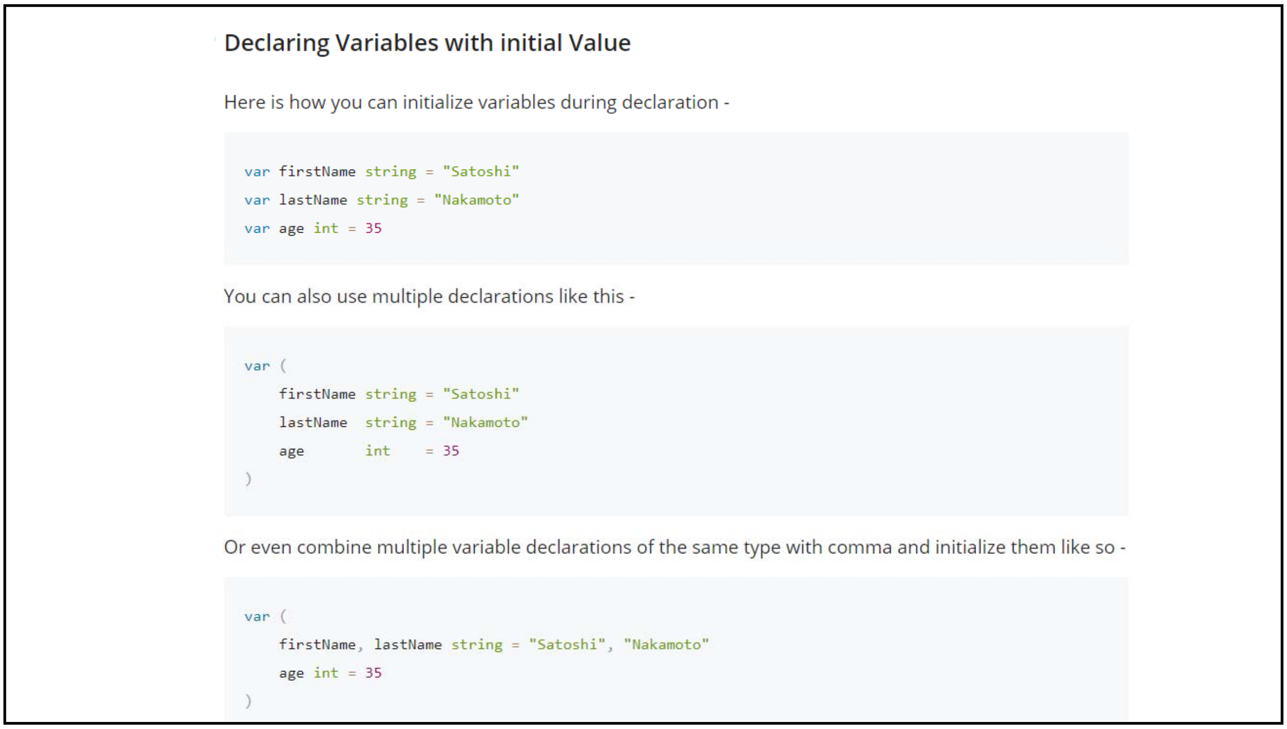
Type	Zero Value
bool	false
string	""
int, int8, int16 etc.	0
float32, float64	0.0



The example below demonstrates the concept of zero values:

```go
package main

import "fmt"

func main() {
    var (
        firstName, lastName string
        age                 int
        salary              float64
        isConfirmed         bool
    )

    fmt.Printf("firstName: %s, lastName: %s, age: %d, salary: %f, isConfirmed: %t\n",
        firstName, lastName, age, salary, isConfirmed)
}
```

```
# Output
firstName: , lastName: , age: 0, salary: 0.000000, isConfirmed: false
```



Declaring Variables with initial Value

Here is how you can initialize variables during declaration -

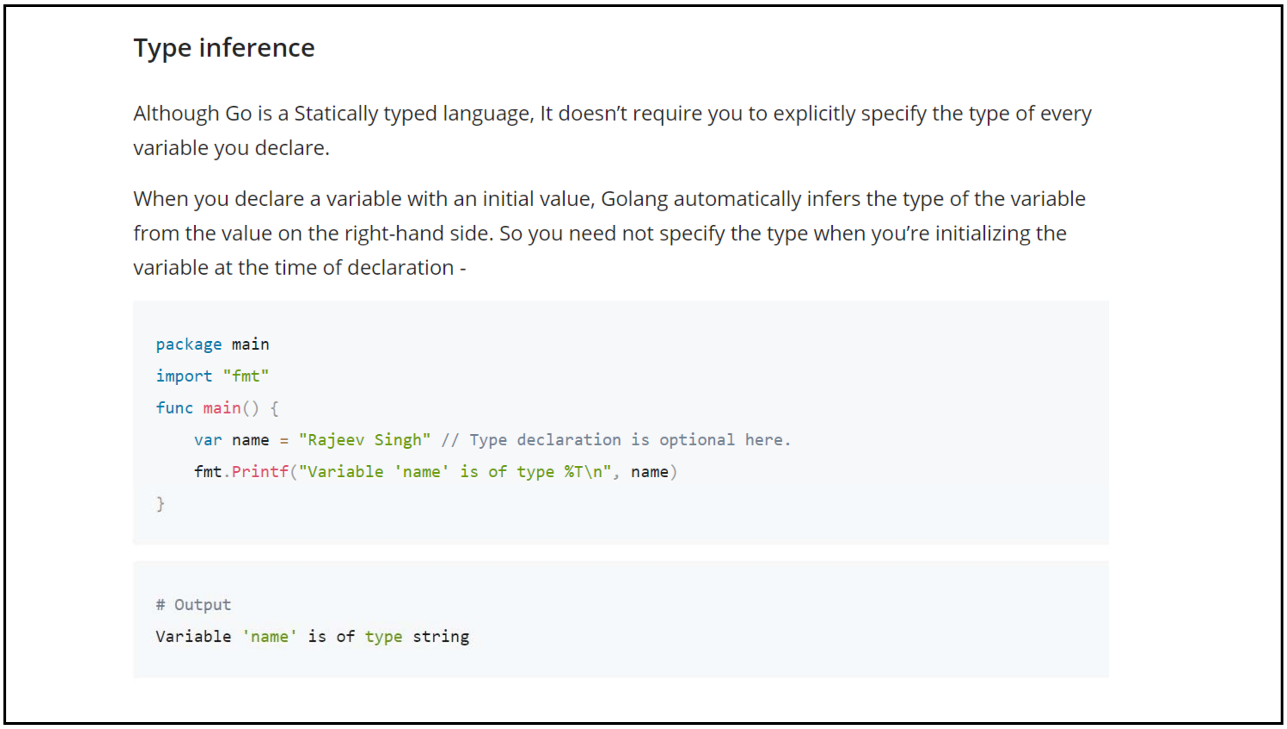
```go
var firstName string = "Satoshi"
var lastName string = "Nakamoto"
var age int = 35
```

You can also use multiple declarations like this -

```go
var (
    firstName string = "Satoshi"
    lastName  string = "Nakamoto"
    age       int    = 35
)
```

Or even combine multiple variable declarations of the same type with comma and initialize them like so -

```go
var (
    firstName, lastName string = "Satoshi", "Nakamoto"
    age int = 35
)
```



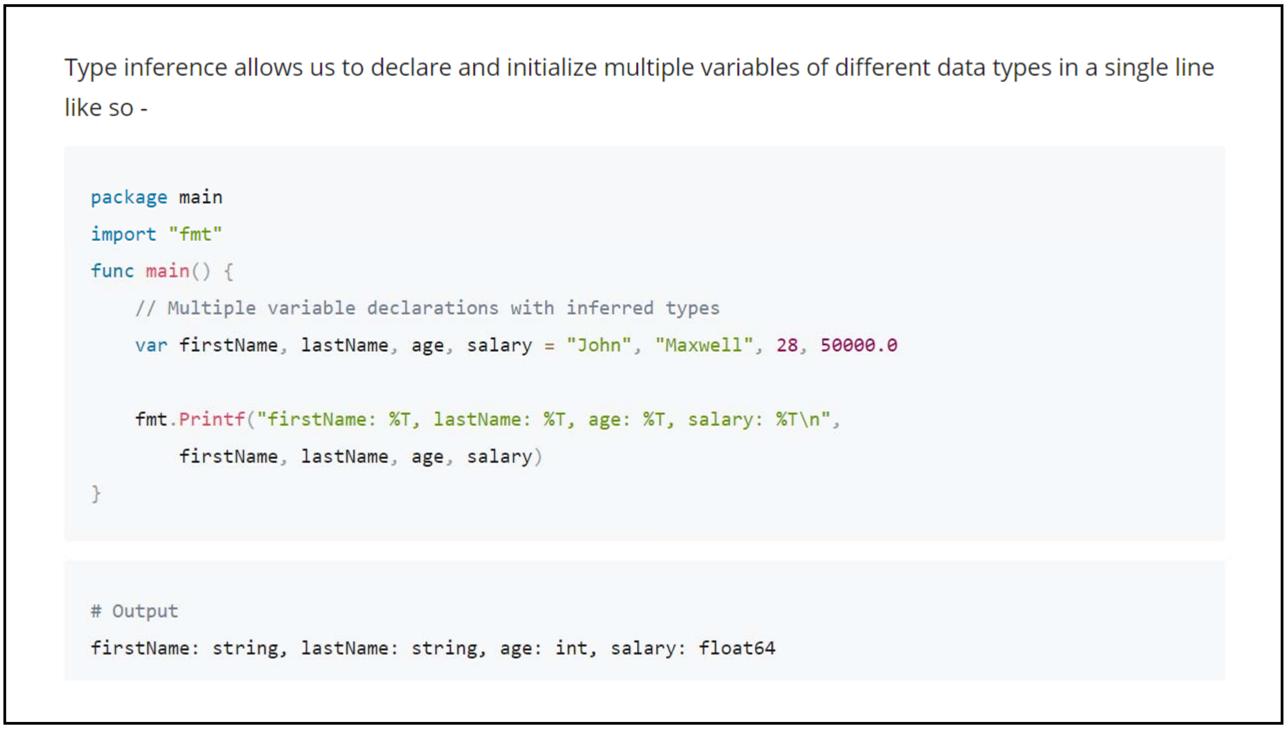
Type inference

Although Go is a Statically typed language, It doesn’t require you to explicitly specify the type of every variable you declare.

When you declare a variable with an initial value, Golang automatically infers the type of the variable from the value on the right-hand side. So you need not specify the type when you’re initializing the variable at the time of declaration -

```
package main
import "fmt"
func main() {
    var name = "Rajeev Singh" // Type declaration is optional here.
    fmt.Printf("Variable 'name' is of type %T\n", name)
}
```

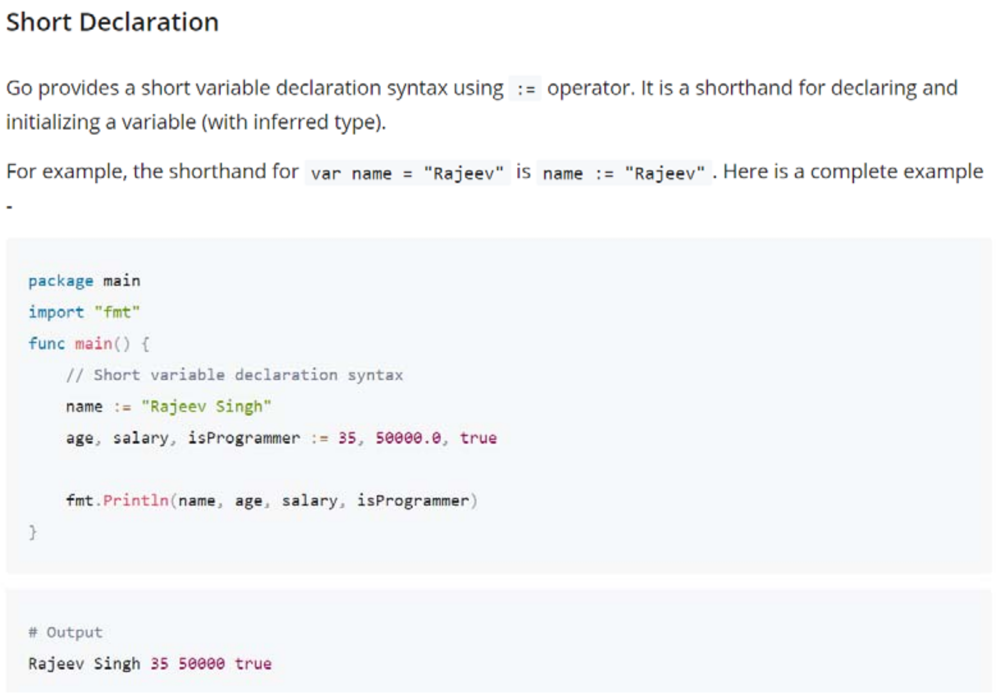
```
# Output
Variable 'name' is of type string
```



Type inference allows us to declare and initialize multiple variables of different data types in a single line like so -

```
package main
import "fmt"
func main() {
    // Multiple variable declarations with inferred types
    var firstName, lastName, age, salary = "John", "Maxwell", 28, 50000.0

    fmt.Printf("firstName: %T, lastName: %T, age: %T, salary: %T\n",
        firstName, lastName, age, salary)
}
```

```
# Output
firstName: string, lastName: string, age: int, salary: float64
```



Short Declaration

Go provides a short variable declaration syntax using `:=` operator. It is a shorthand for declaring and initializing a variable (with inferred type).

For example, the shorthand for `var name = "Rajeev"` is `name := "Rajeev"`. Here is a complete example -

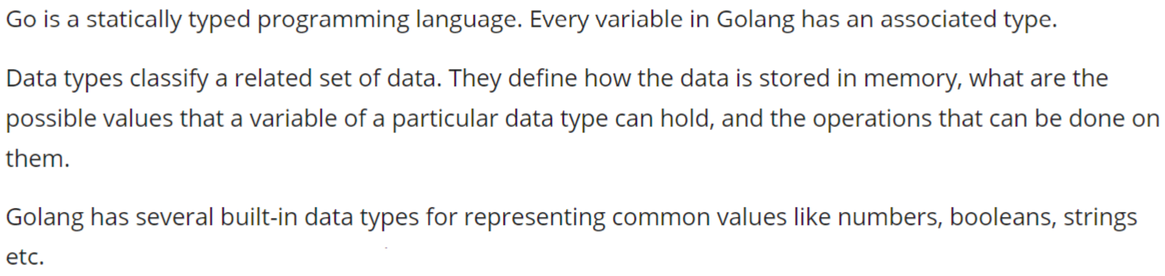
```go
package main
import "fmt"
func main() {
    // Short variable declaration syntax
    name := "Rajeev Singh"
    age, salary, isProgrammer := 35, 50000.0, true

    fmt.Println(name, age, salary, isProgrammer)
}
```

```
# Output
Rajeev Singh 35 50000 true
```

Basic Types

BASIC TYPES, OPERATORS AND TYPE CONVERSION

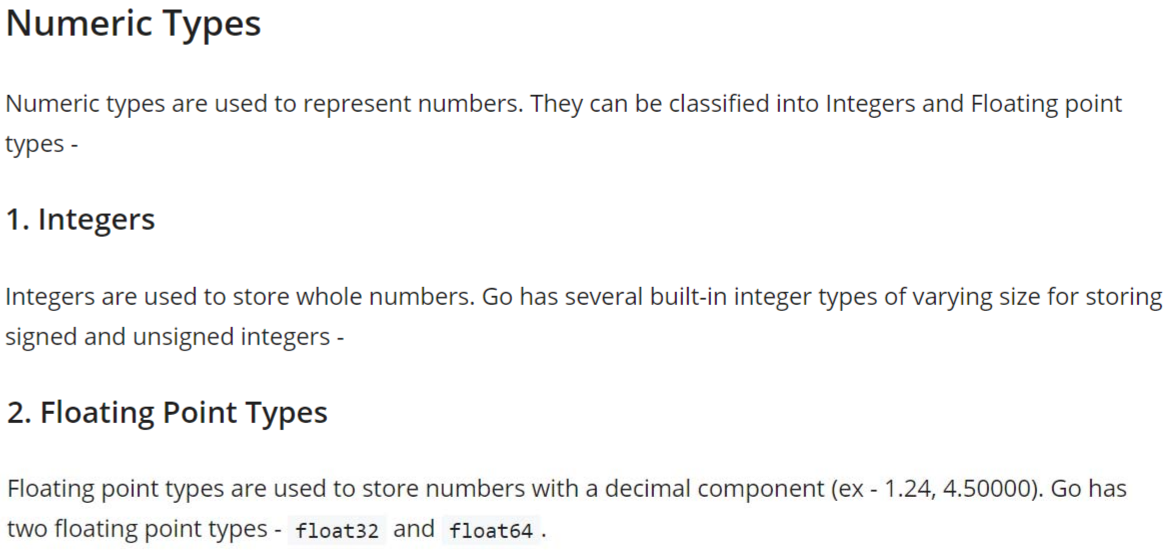
Introduction to Basic Types



Go is a statically typed programming language. Every variable in Golang has an associated type.

Data types classify a related set of data. They define how the data is stored in memory, what are the possible values that a variable of a particular data type can hold, and the operations that can be done on them.

Golang has several built-in data types for representing common values like numbers, booleans, strings etc.



Numeric Types

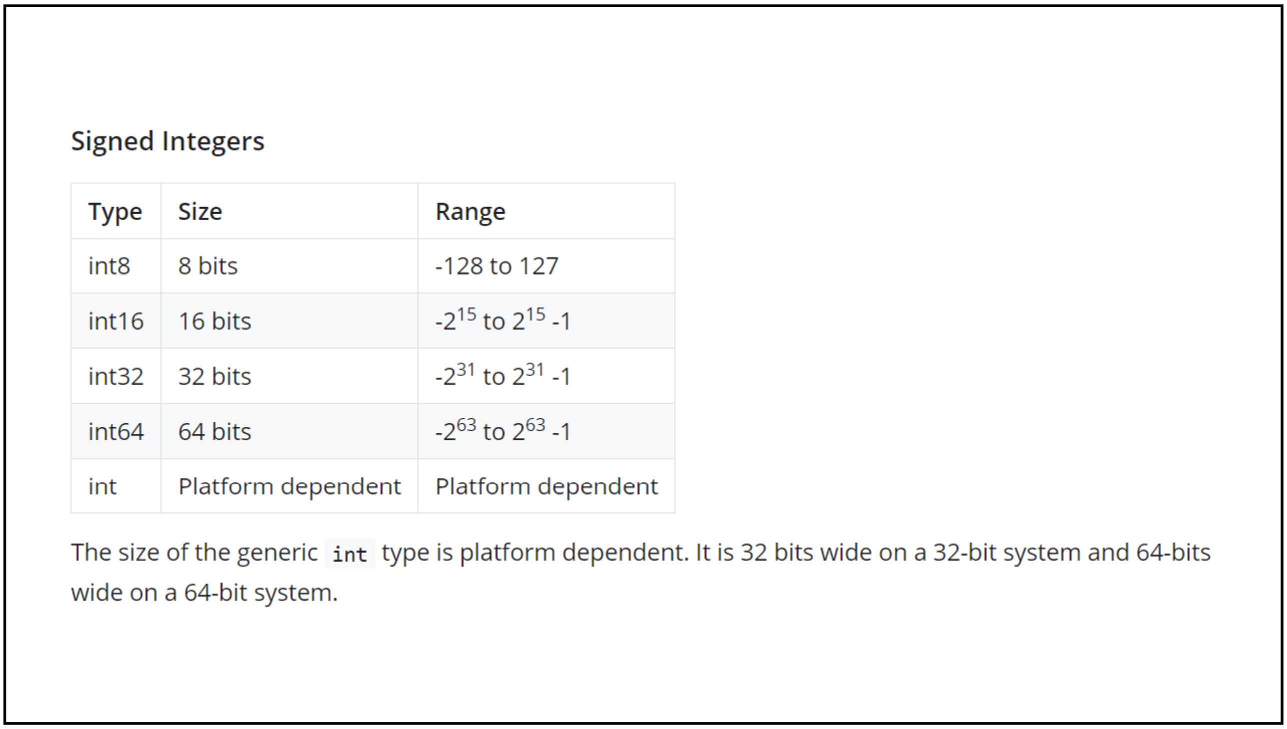
Numeric types are used to represent numbers. They can be classified into Integers and Floating point types -

1. Integers

Integers are used to store whole numbers. Go has several built-in integer types of varying size for storing signed and unsigned integers -

2. Floating Point Types

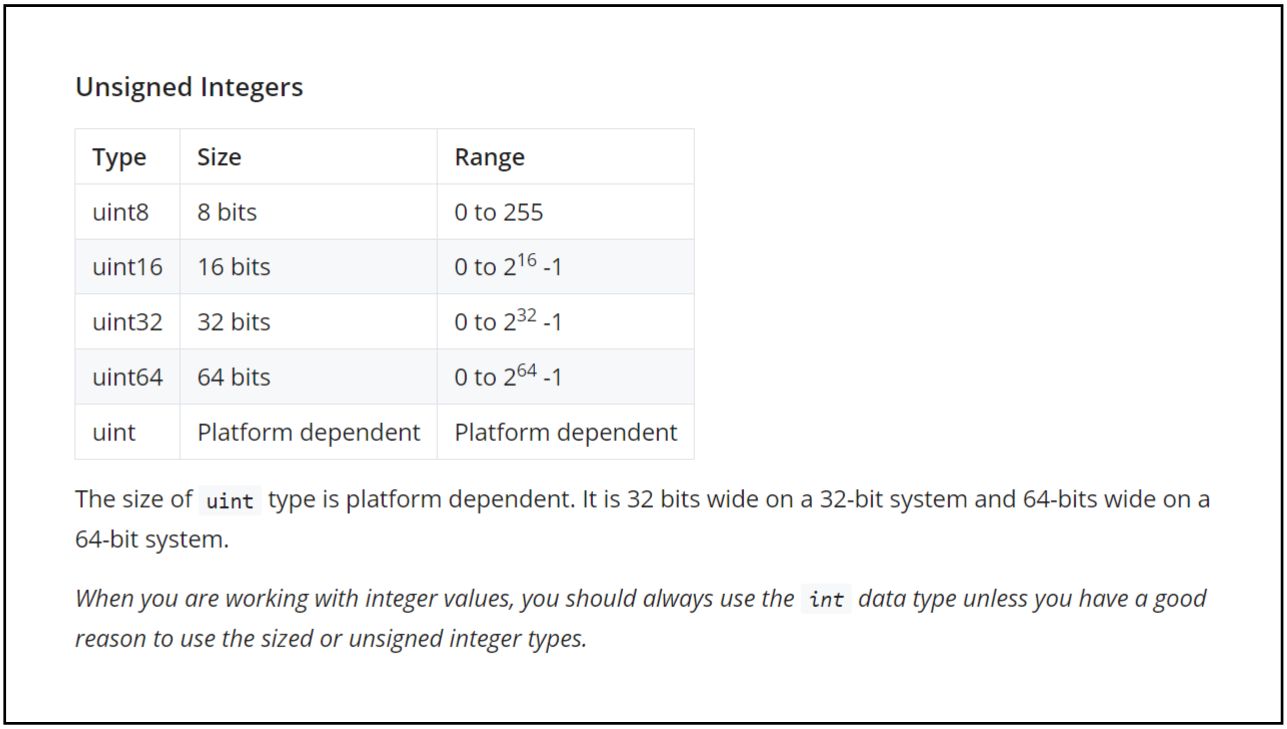
Floating point types are used to store numbers with a decimal component (ex - 1.24, 4.50000). Go has two floating point types - `float32` and `float64` .



Signed Integers

Type	Size	Range
int8	8 bits	-128 to 127
int16	16 bits	-2^{15} to 2^{15} -1
int32	32 bits	-2^{31} to 2^{31} -1
int64	64 bits	-2^{63} to 2^{63} -1
int	Platform dependent	Platform dependent

The size of the generic `int` type is platform dependent. It is 32 bits wide on a 32-bit system and 64-bits wide on a 64-bit system.

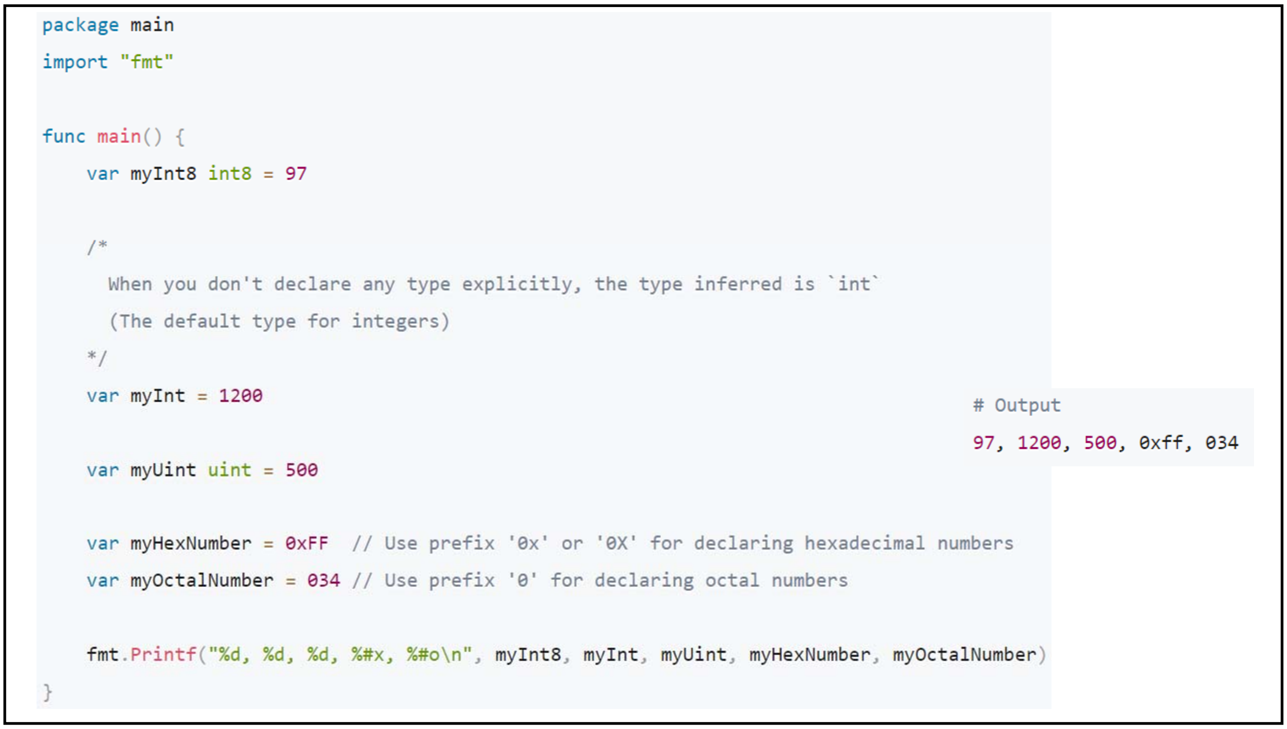


Unsigned Integers

Type	Size	Range
uint8	8 bits	0 to 255
uint16	16 bits	0 to 2^{16} -1
uint32	32 bits	0 to 2^{32} -1
uint64	64 bits	0 to 2^{64} -1
uint	Platform dependent	Platform dependent

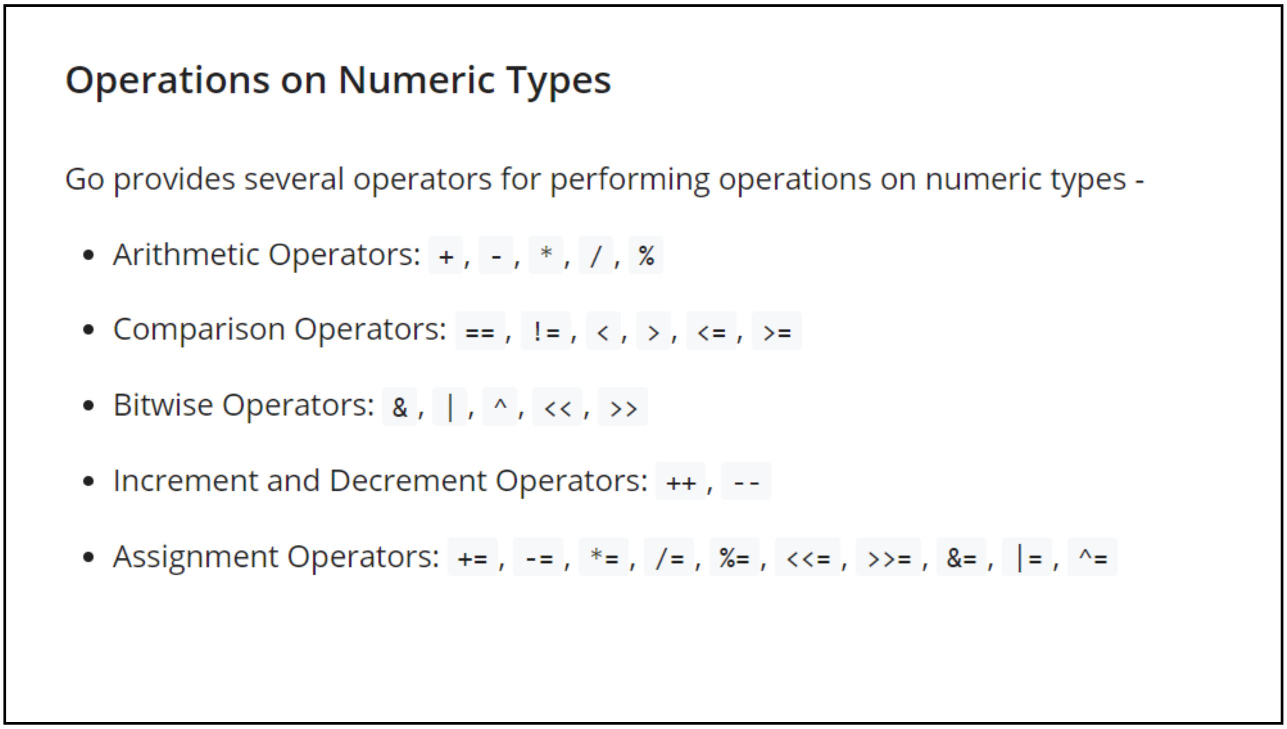
The size of `uint` type is platform dependent. It is 32 bits wide on a 32-bit system and 64-bits wide on a 64-bit system.

When you are working with integer values, you should always use the `int` *data type unless you have a good reason to use the sized or unsigned integer types.*



```go
package main
import "fmt"

func main() {
    var myInt8 int8 = 97

    /*
      When you don't declare any type explicitly, the type inferred is `int`
      (The default type for integers)
    */
    var myInt = 1200

    var myUint uint = 500

    var myHexNumber = 0xFF  // Use prefix '0x' or '0X' for declaring hexadecimal numbers
    var myOctalNumber = 034 // Use prefix '0' for declaring octal numbers

    fmt.Printf("%d, %d, %d, %#x, %#o\n", myInt8, myInt, myUint, myHexNumber, myOctalNumber)
}
```

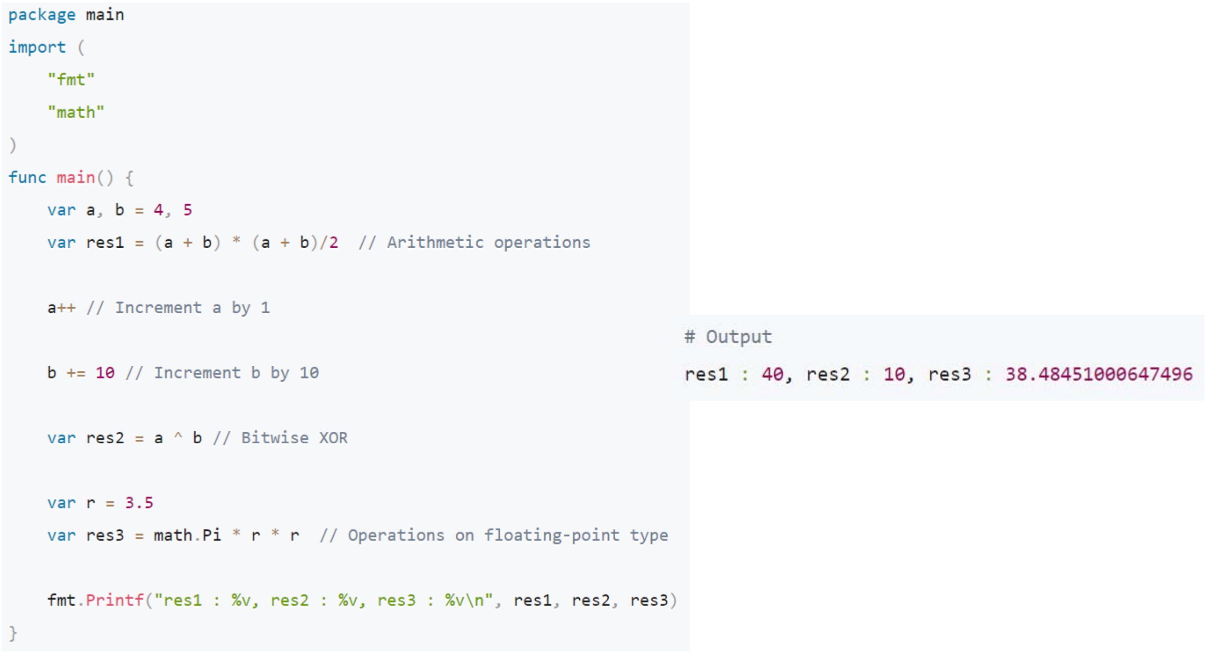
```
# Output
97, 1200, 500, 0xff, 034
```



Operations on Numeric Types

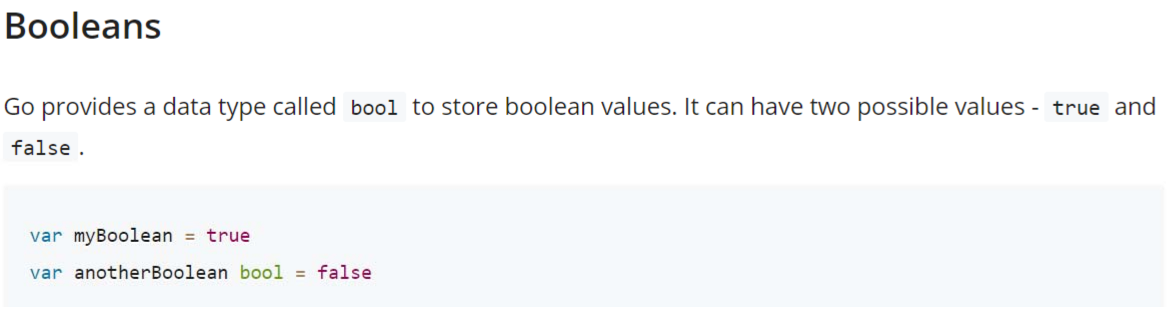
Go provides several operators for performing operations on numeric types -

- Arithmetic Operators: `+`, `-`, `*`, `/`, `%`
- Comparison Operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Bitwise Operators: `&`, `|`, `^`, `<<`, `>>`
- Increment and Decrement Operators: `++`, `--`
- Assignment Operators: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`



```
package main
import (
    "fmt"
    "math"
)
func main() {
    var a, b = 4, 5
    var res1 = (a + b) * (a + b)/2  // Arithmetic operations

    a++ // Increment a by 1

    b += 10 // Increment b by 10

    var res2 = a ^ b // Bitwise XOR

    var r = 3.5
    var res3 = math.Pi * r * r  // Operations on floating-point type

    fmt.Printf("res1 : %v, res2 : %v, res3 : %v\n", res1, res2, res3)
}
```

```
# Output
res1 : 40, res2 : 10, res3 : 38.48451000647496
```



Booleans

Go provides a data type called `bool` to store boolean values. It can have two possible values - `true` and `false`.

```
var myBoolean = true
var anotherBoolean bool = false
```

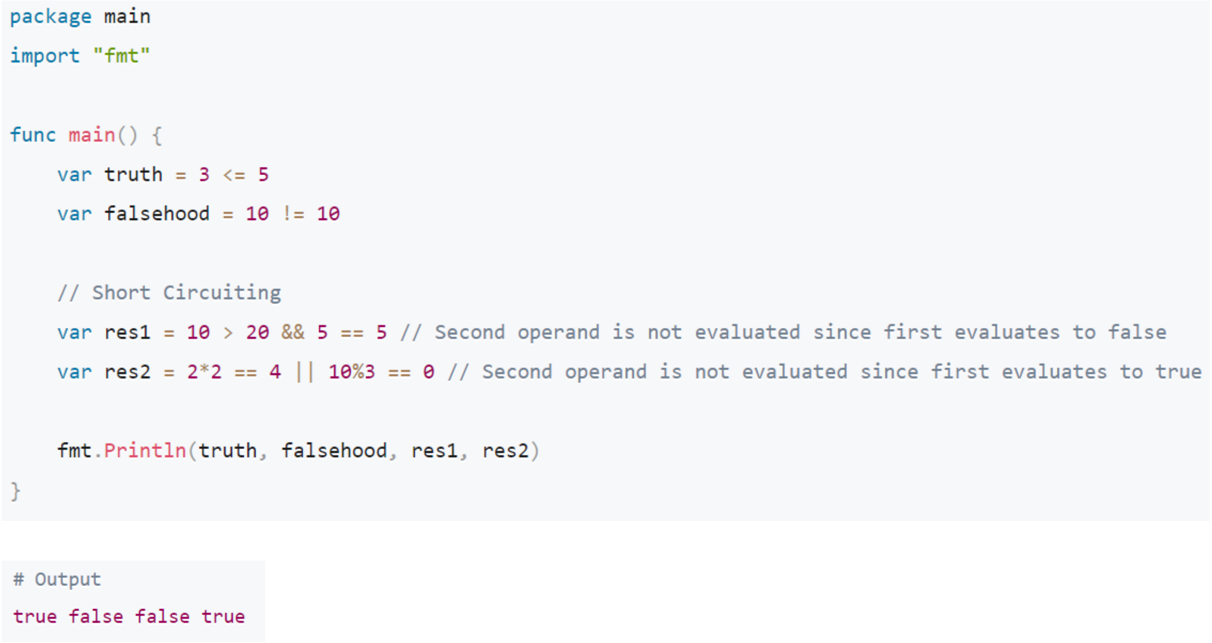
Operations on Boolean Types

You can use the following operators on boolean types -

Logical Operators:

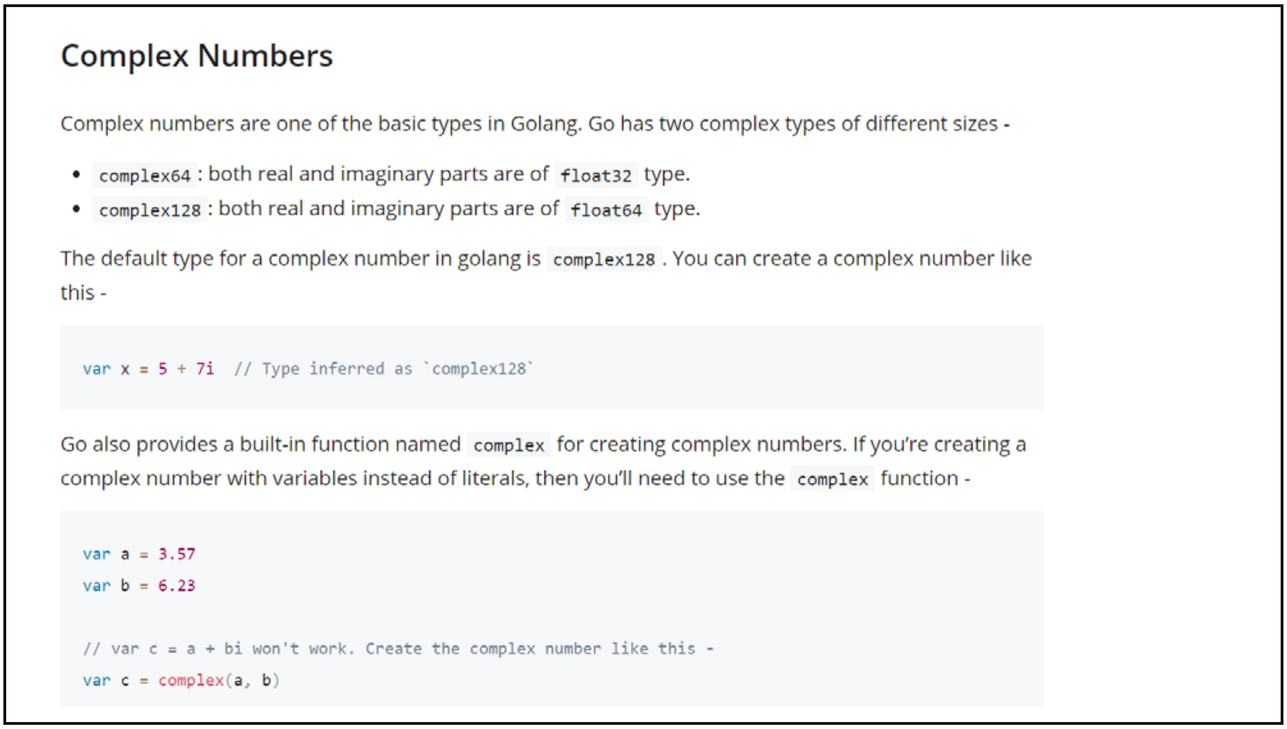
- `&&` (logical conjunction, “and”)
- `||` (logical disjunction, “or”)
- `!` (logical negation)

Equality and Inequality: `==` , `!=`



```go
package main
import "fmt"

func main() {
    var truth = 3 <= 5
    var falsehood = 10 != 10

    // Short Circuiting
    var res1 = 10 > 20 && 5 == 5 // Second operand is not evaluated since first evaluates to false
    var res2 = 2*2 == 4 || 10%3 == 0 // Second operand is not evaluated since first evaluates to true

    fmt.Println(truth, falsehood, res1, res2)
}
```

```
# Output
true false false true
```



Complex Numbers

Complex numbers are one of the basic types in Golang. Go has two complex types of different sizes -

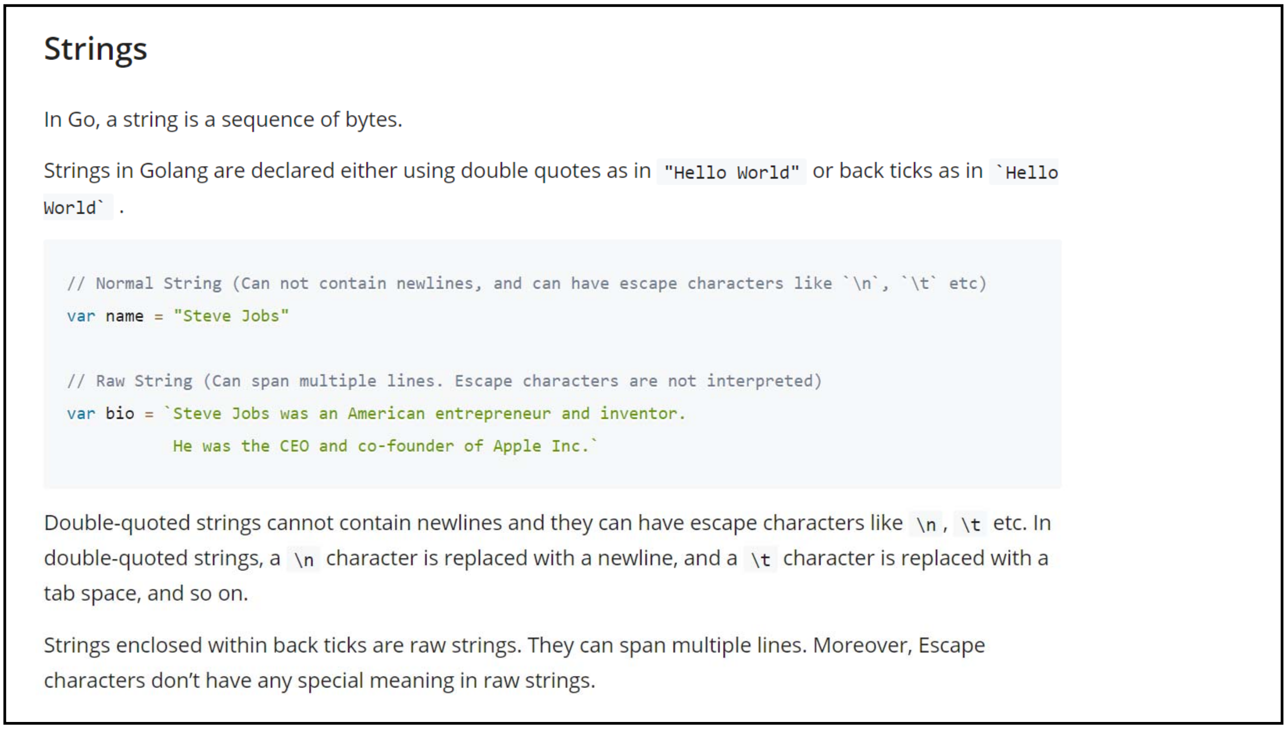
- `complex64` : both real and imaginary parts are of `float32` type.
- `complex128` : both real and imaginary parts are of `float64` type.

The default type for a complex number in golang is `complex128` . You can create a complex number like this -

```
var x = 5 + 7i  // Type inferred as `complex128`
```

Go also provides a built-in function named `complex` for creating complex numbers. If you're creating a complex number with variables instead of literals, then you'll need to use the `complex` function -

```
var a = 3.57
var b = 6.23

// var c = a + bi won't work. Create the complex number like this -
var c = complex(a, b)
```



Strings

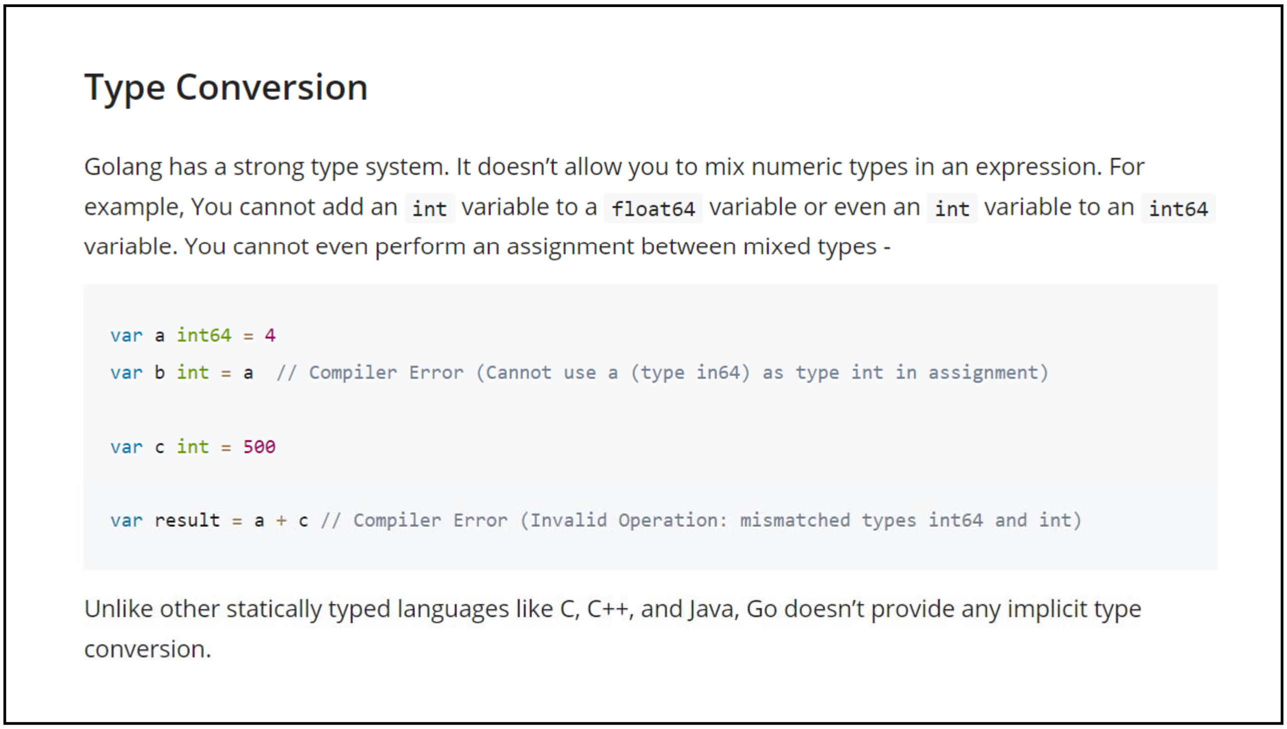
In Go, a string is a sequence of bytes.

Strings in Golang are declared either using double quotes as in `"Hello World"` or back ticks as in `` `Hello World` `` .

```
// Normal String (Can not contain newlines, and can have escape characters like `\n`, `\t` etc)
var name = "Steve Jobs"

// Raw String (Can span multiple lines. Escape characters are not interpreted)
var bio = `Steve Jobs was an American entrepreneur and inventor.
           He was the CEO and co-founder of Apple Inc.`
```

Double-quoted strings cannot contain newlines and they can have escape characters like `\n` , `\t` etc. In double-quoted strings, a `\n` character is replaced with a newline, and a `\t` character is replaced with a tab space, and so on.

Strings enclosed within back ticks are raw strings. They can span multiple lines. Moreover, Escape characters don't have any special meaning in raw strings.

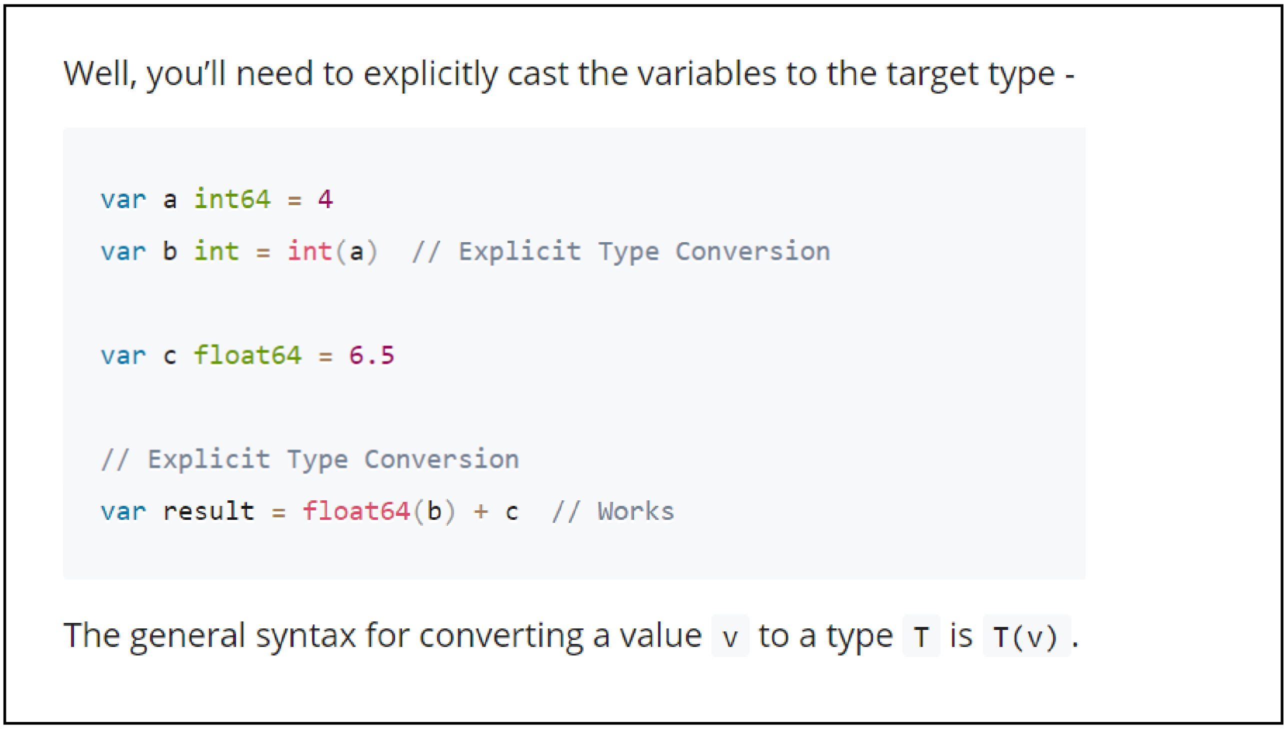


Type Conversion

Golang has a strong type system. It doesn't allow you to mix numeric types in an expression. For example, You cannot add an `int` variable to a `float64` variable or even an `int` variable to an `int64` variable. You cannot even perform an assignment between mixed types -

```
var a int64 = 4
var b int = a  // Compiler Error (Cannot use a (type in64) as type int in assignment)

var c int = 500

var result = a + c // Compiler Error (Invalid Operation: mismatched types int64 and int)
```

Unlike other statically typed languages like C, C++, and Java, Go doesn't provide any implicit type conversion.



Well, you'll need to explicitly cast the variables to the target type -

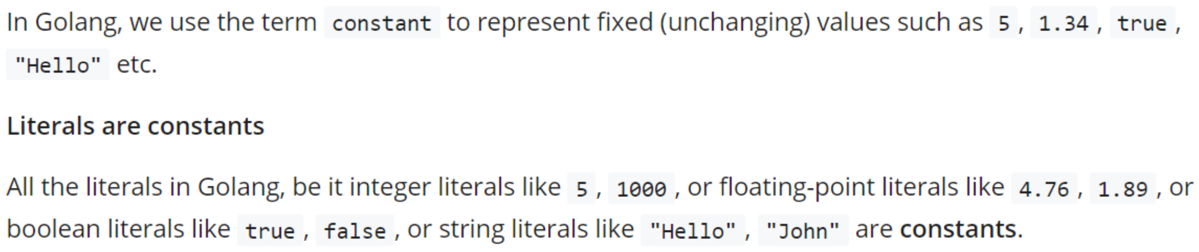
```
var a int64 = 4
var b int = int(a)  // Explicit Type Conversion

var c float64 = 6.5

// Explicit Type Conversion
var result = float64(b) + c  // Works
```

The general syntax for converting a value `v` to a type `T` is `T(v)`.

Working with Primitive Data Types

CONSTANTS, IOTA

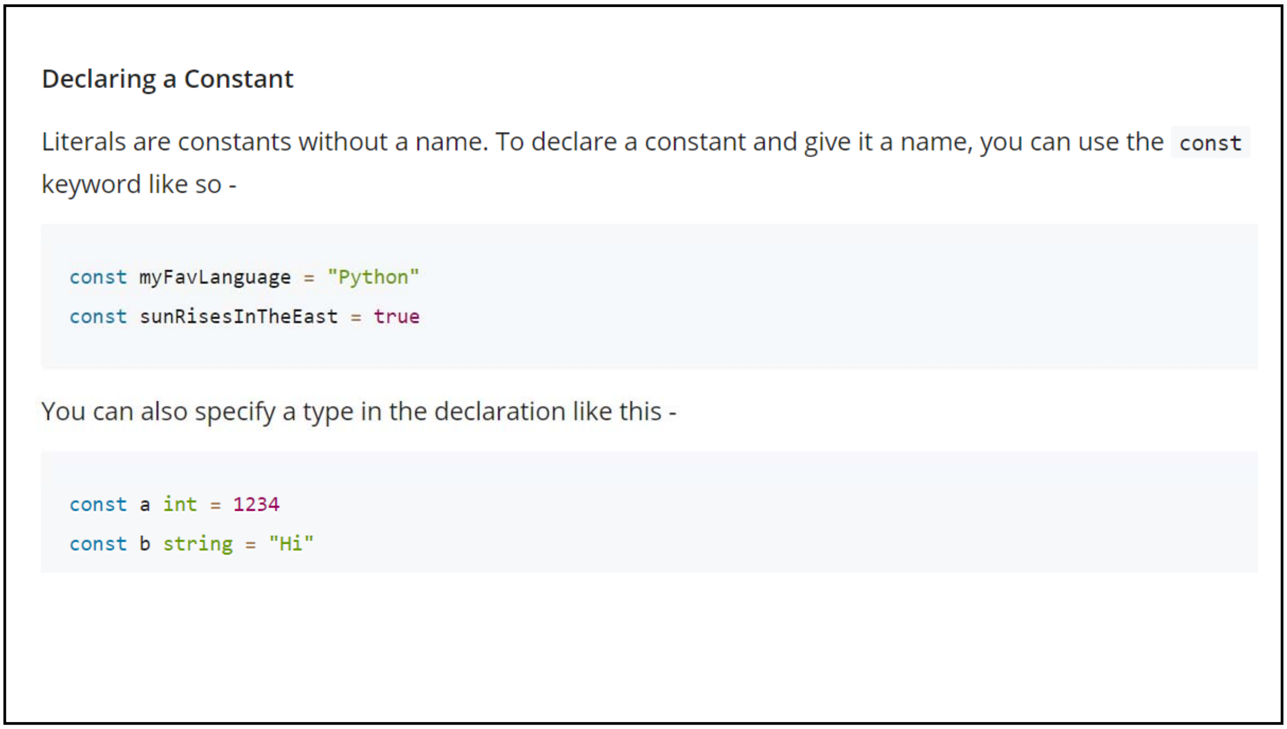
Introduction to Constants



In Golang, we use the term `constant` to represent fixed (unchanging) values such as `5` , `1.34` , `true` , `"Hello"` etc.

Literals are constants

All the literals in Golang, be it integer literals like `5` , `1000` , or floating-point literals like `4.76` , `1.89` , or boolean literals like `true` , `false` , or string literals like `"Hello"` , `"John"` are **constants**.



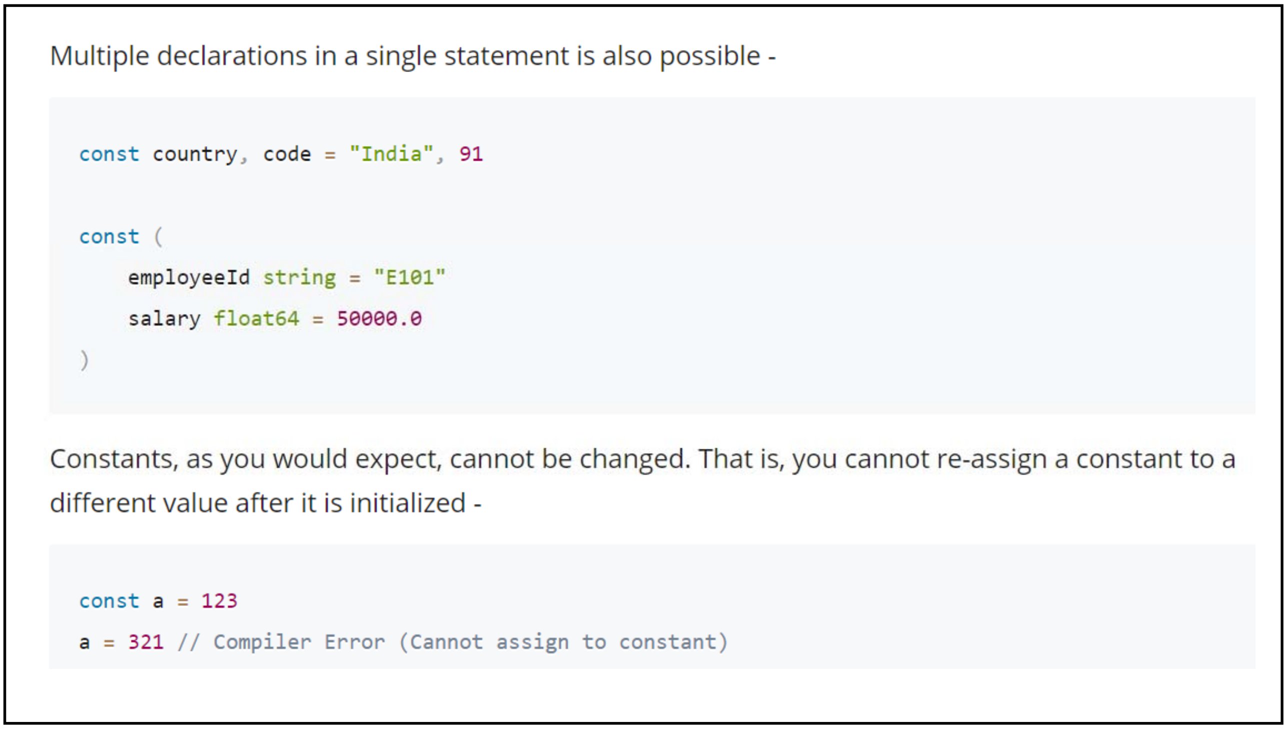
Declaring a Constant

Literals are constants without a name. To declare a constant and give it a name, you can use the `const` keyword like so -

```
const myFavLanguage = "Python"
const sunRisesInTheEast = true
```

You can also specify a type in the declaration like this -

```
const a int = 1234
const b string = "Hi"
```



Multiple declarations in a single statement is also possible -

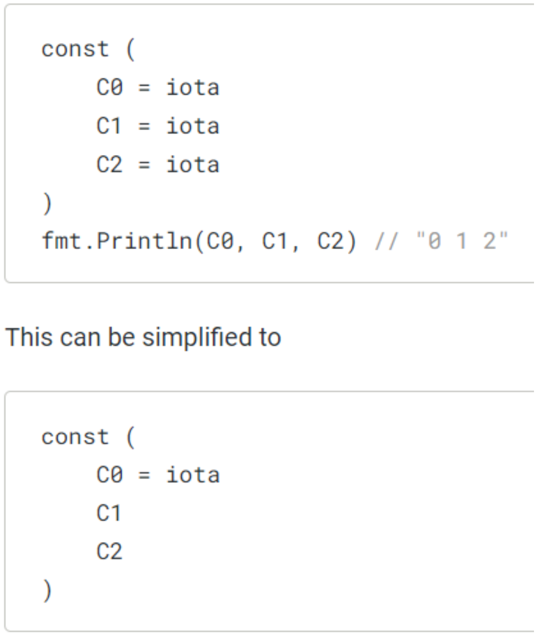
```
const country, code = "India", 91

const (
    employeeId string = "E101"
    salary float64 = 50000.0
)
```

Constants, as you would expect, cannot be changed. That is, you cannot re-assign a constant to a different value after it is initialized -

```
const a = 123
a = 321 // Compiler Error (Cannot assign to constant)
```

Iota basic example

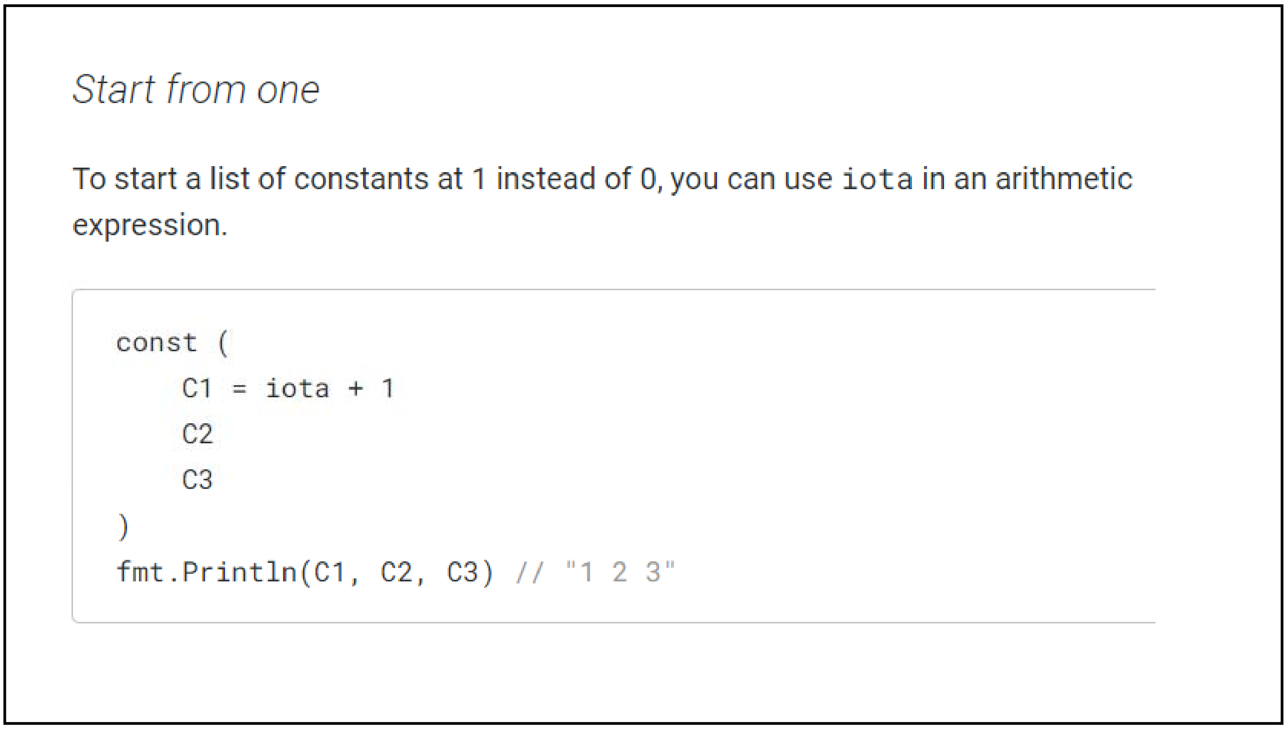
- The `iota` keyword represents successive integer constants 0, 1, 2,…
- It resets to 0 whenever the word `const` appears in the source code,
- and increments after each const specification.



```
const (
    C0 = iota
    C1 = iota
    C2 = iota
)
fmt.Println(C0, C1, C2) // "0 1 2"
```

This can be simplified to

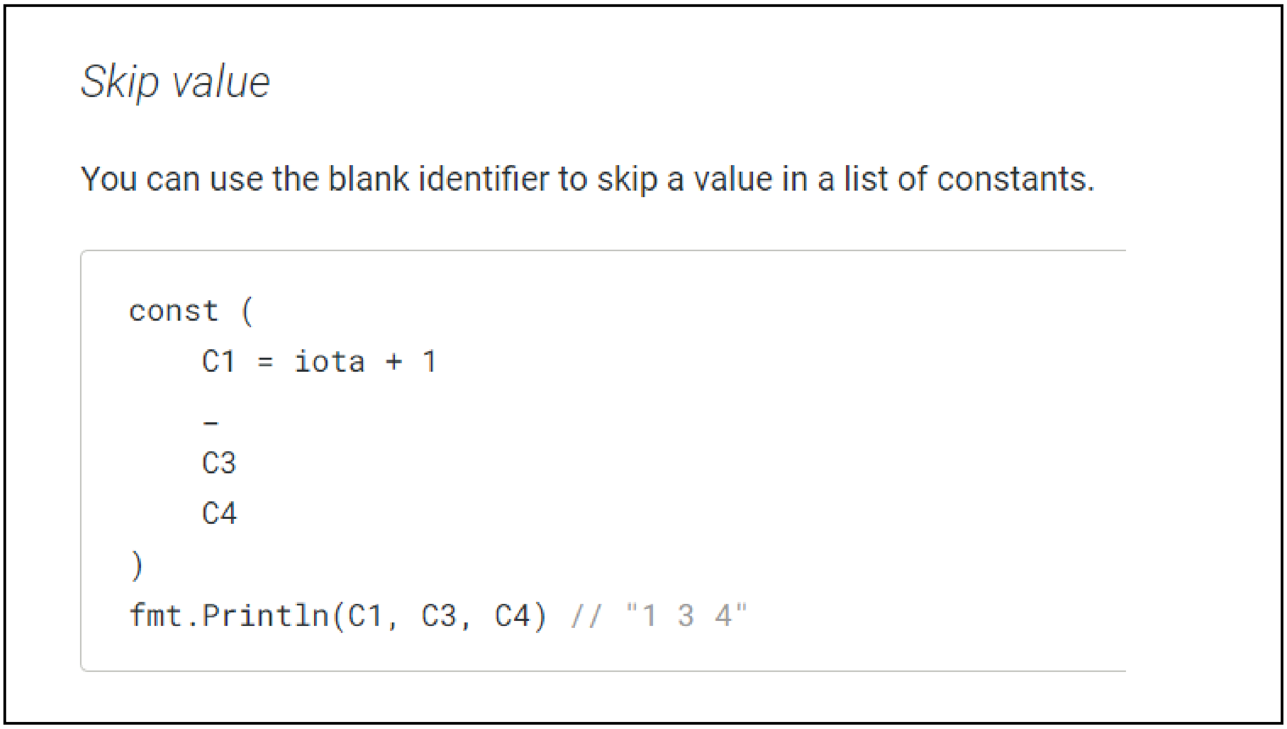
```
const (
    C0 = iota
    C1
    C2
)
```



Start from one

To start a list of constants at 1 instead of 0, you can use `iota` in an arithmetic expression.

```
const (
    C1 = iota + 1
    C2
    C3
)
fmt.Println(C1, C2, C3) // "1 2 3"
```



Skip value

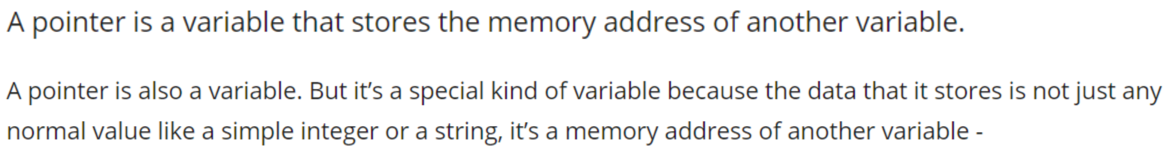
You can use the blank identifier to skip a value in a list of constants.

```
const (
    C1 = iota + 1
    _
    C3
    C4
)
fmt.Println(C1, C3, C4) // "1 3 4"
```

Working with Primitive Data Types

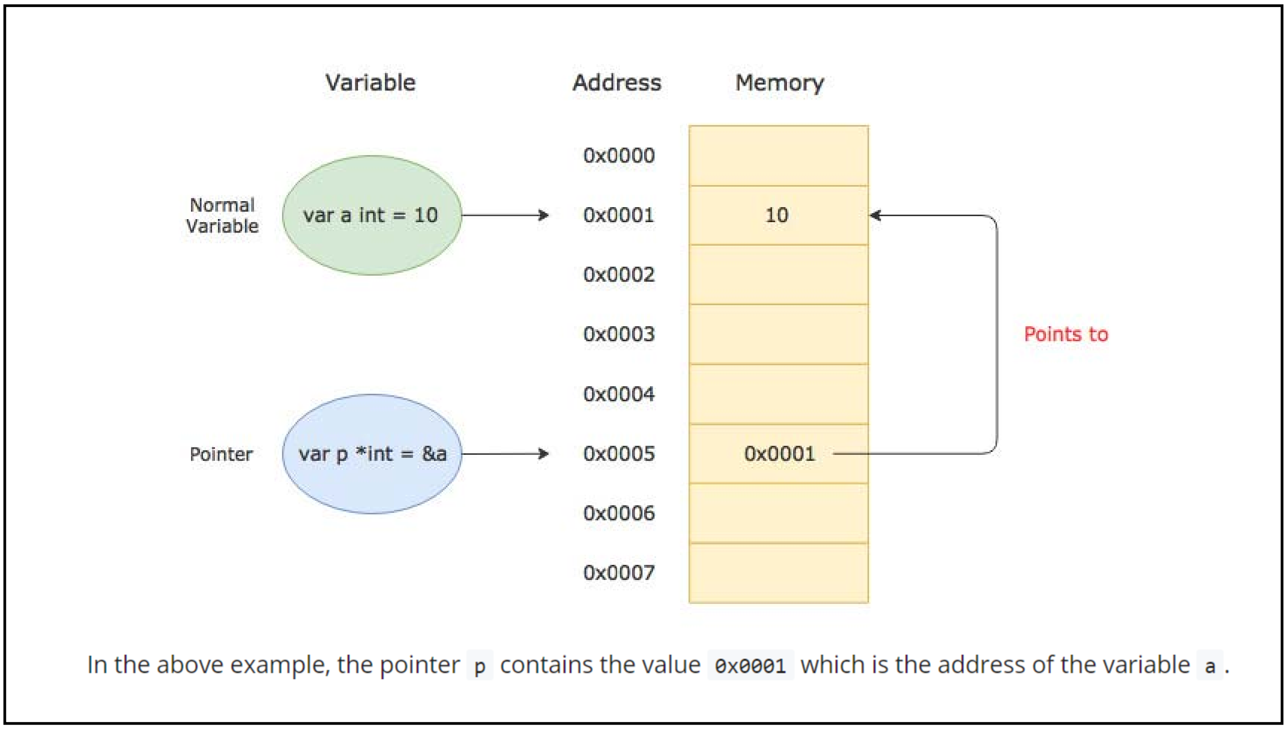
POINTERS

Introduction to Pointers

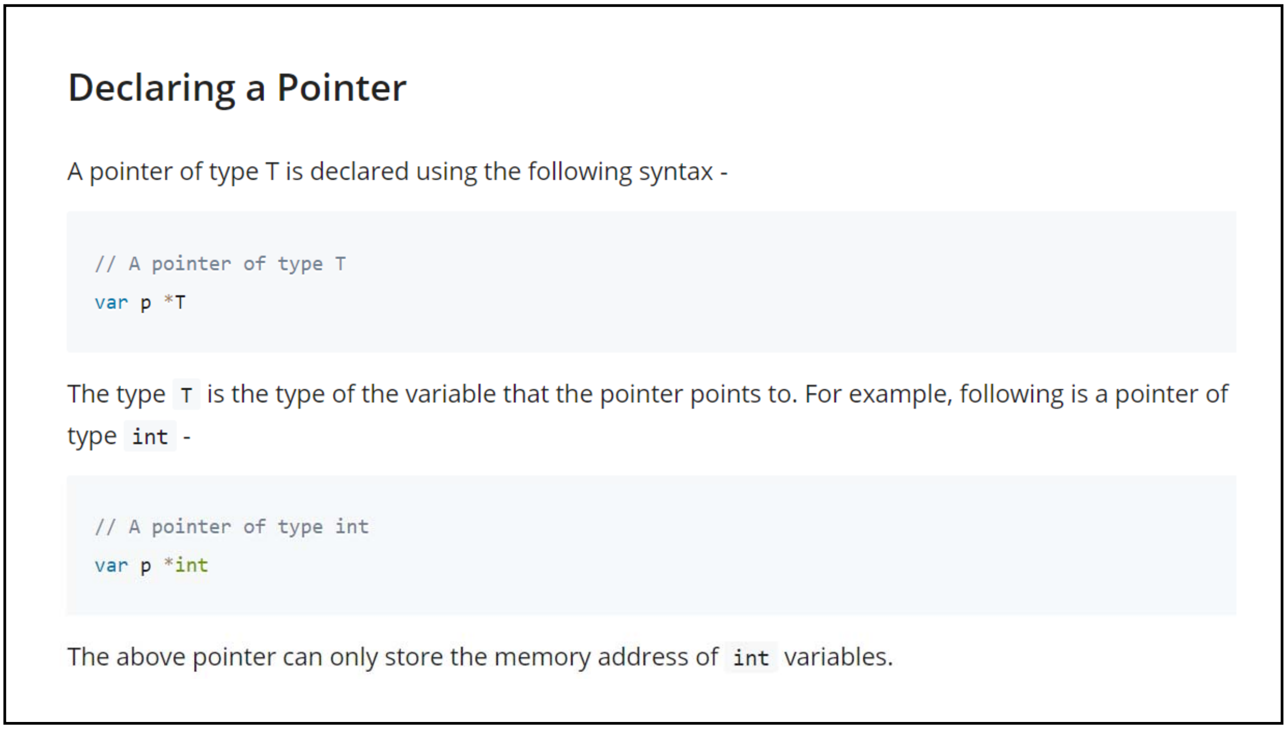


A pointer is a variable that stores the memory address of another variable.

A pointer is also a variable. But it’s a special kind of variable because the data that it stores is not just any normal value like a simple integer or a string, it’s a memory address of another variable -



In the above example, the pointer `p` contains the value `0x0001` which is the address of the variable `a` .



Declaring a Pointer

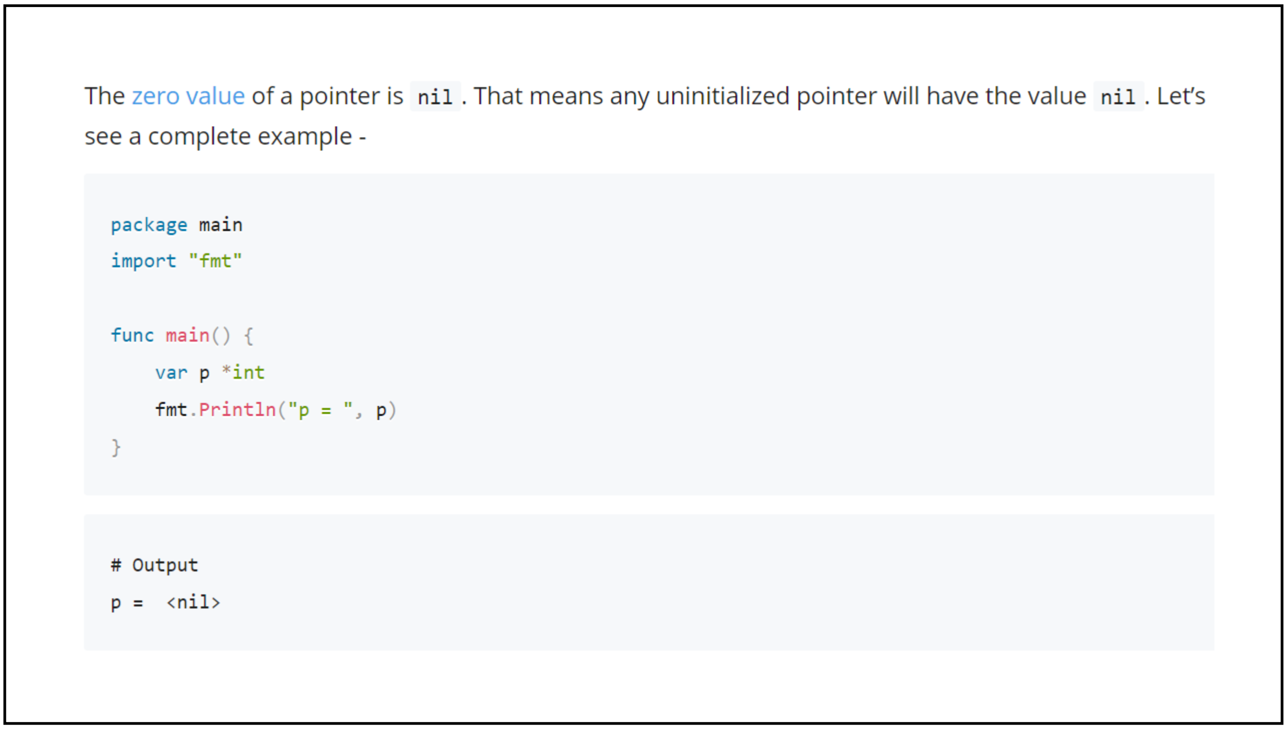
A pointer of type T is declared using the following syntax -

```
// A pointer of type T
var p *T
```

The type `T` is the type of the variable that the pointer points to. For example, following is a pointer of type `int` -

```
// A pointer of type int
var p *int
```

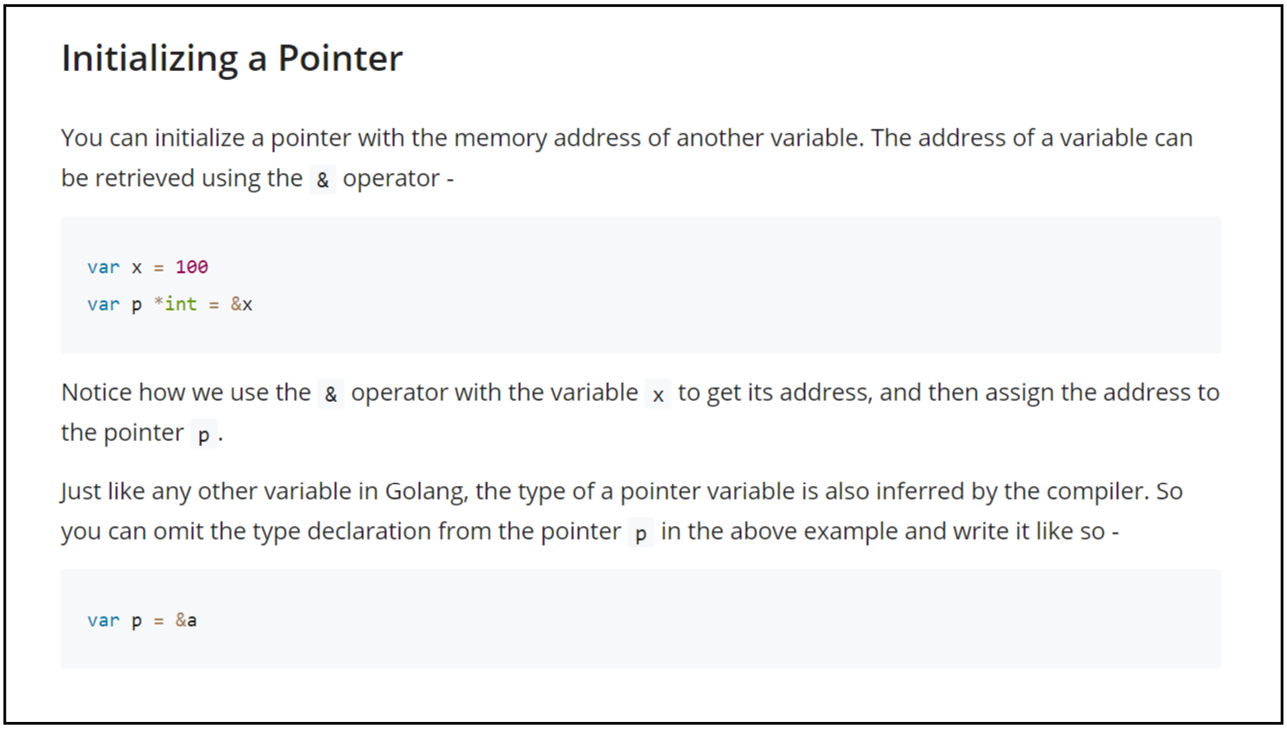
The above pointer can only store the memory address of `int` variables.



The zero value of a pointer is `nil` . That means any uninitialized pointer will have the value `nil` . Let's see a complete example -

```
package main
import "fmt"

func main() {
    var p *int
    fmt.Println("p = ", p)
}
```

```
# Output
p =  <nil>
```



Initializing a Pointer

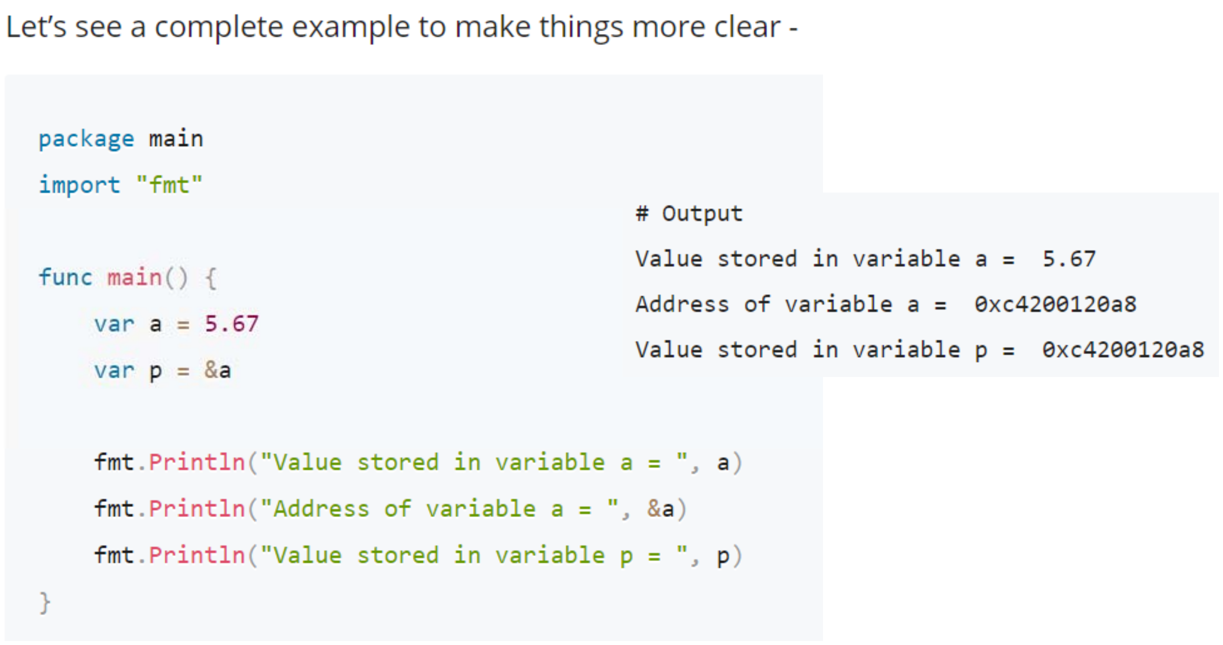
You can initialize a pointer with the memory address of another variable. The address of a variable can be retrieved using the `&` operator -

```
var x = 100
var p *int = &x
```

Notice how we use the `&` operator with the variable `x` to get its address, and then assign the address to the pointer `p` .

Just like any other variable in Golang, the type of a pointer variable is also inferred by the compiler. So you can omit the type declaration from the pointer `p` in the above example and write it like so -

```
var p = &a
```



Let’s see a complete example to make things more clear -

```go
package main
import "fmt"

func main() {
    var a = 5.67
    var p = &a

    fmt.Println("Value stored in variable a = ", a)
    fmt.Println("Address of variable a = ", &a)
    fmt.Println("Value stored in variable p = ", p)
}
```

```
# Output
Value stored in variable a =  5.67
Address of variable a =  0xc4200120a8
Value stored in variable p =  0xc4200120a8
```

Summary

Declaring Variable

Basic Types, Operators and Type Conversion

Constants, Iota

Pointers