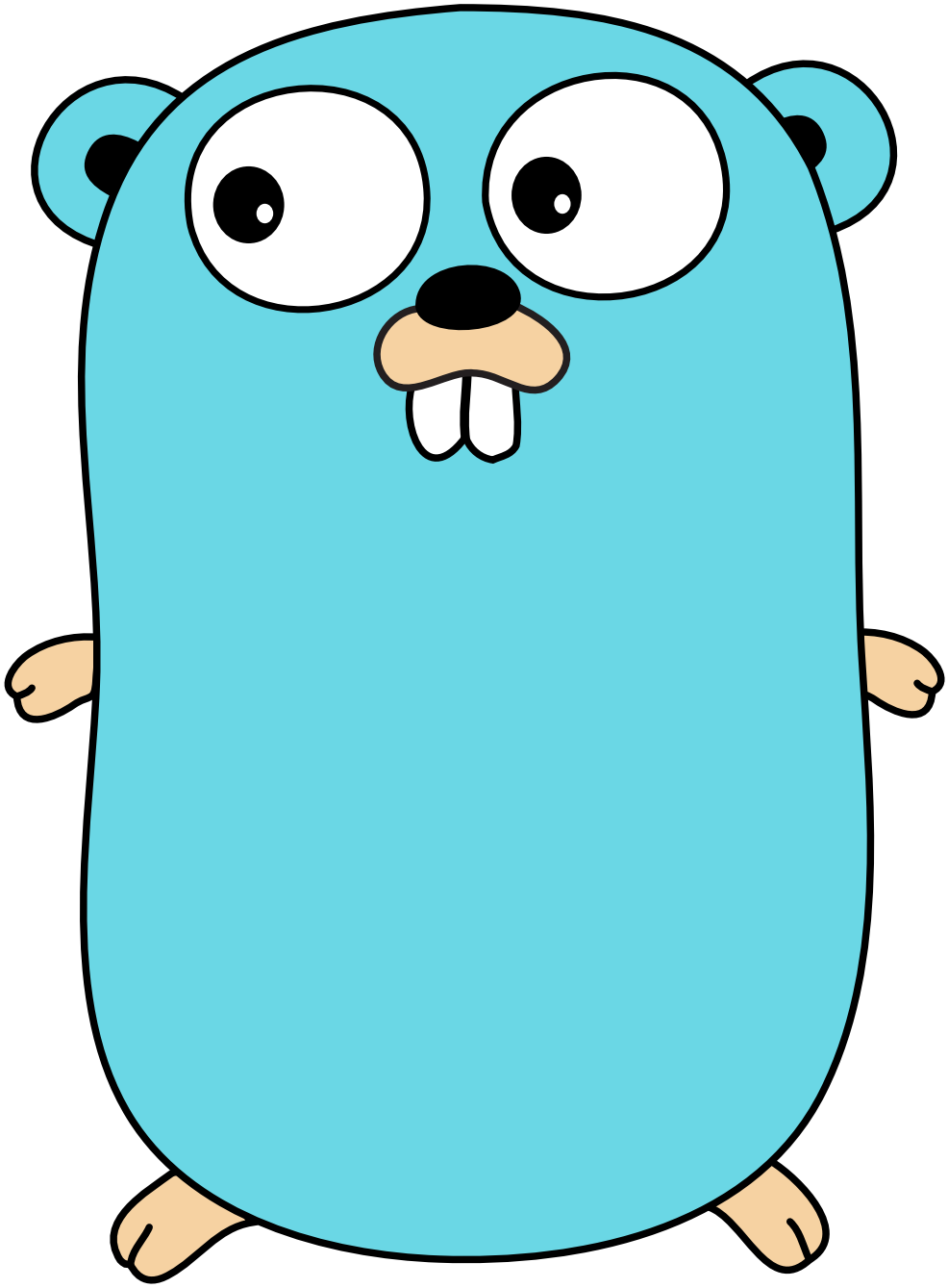
Grundlagen der Programmierung in Go



Inhaltsverzeichnis

Die Programmiersprache Go

Go (auch **Golang** genannt) ist eine statisch typisierte, kompilierte Programmiersprache, die von den Mitarbeitern Robert Griesemer, Rob Pike und Ken Thompson des Unternehmens Google Inc. entwickelt wurde. Rob Pike und Ken Thompson waren unter anderem auch maßgeblich an der Entwicklung der Programmiersprache C und des Unix-Betriebssystems beteiligt.

Die Syntax von Go orientiert sich im Wesentlichen an der Syntax der Programmiersprache C, weicht davon aber an einigen Stellen ab. So kann beispielsweise auf den Abschluss von Anweisungen durch ein Semikolon verzichtet werden und Datentypen werden bei Deklarationen hinter den Bezeichner geschrieben statt davor, um die Deklaration von Funktionstypen zu vereinfachen.

Im März 2012 wurde Version 1 freigegeben.

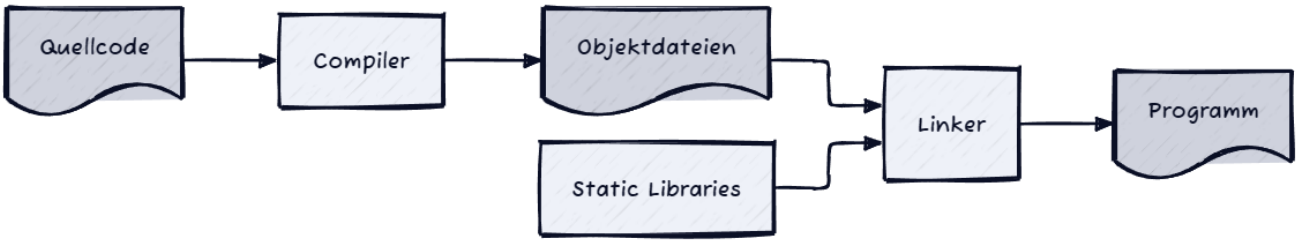
Compiler, Linker und das Go-Kommando

Der Go-Quellcode eines Projektes besteht aus Textdateien mit der Endung `.go` und ggf. `go.mod` und `go.sum` zur Beschreibung eines Moduls. Jedes Go-Projekt ist ein Modul. Ein Modul besteht wiederum aus Packages. Alle Tools im Go-SDK werden über das Go-Kommando aufgerufen.

```
# Einzelne Quellcode-Datei als Programm ausführen
go run hello.go

# Go-Projekt im aktuellen Ordner als Programm ausführen
go run hello.go

# Baut ein Go-Projekt aus angegebenen Quellcode-Dateien
go build hello.go

# Baut ein ganzes Go-Projekt im aktuellen Odner
go build

# Quellcode formartieren
go fmt
```

Wenn ein Projekt mit `go build` gebaut wird, dann wird intern zuerst mit dem **Compiler** jedes Package in eine **Objektdatei** übersetzt und anschließend der **Linker** aufgerufen um aus allen **Objektdateien** und verwendeten Bibliotheken ein ausführbares Programm zu erzeugen.

Der Go-Compiler erzeugt nativen Machinen-Code für eine bestimmte Prozessor-Architektur und der **Linker** erstellt ausführbare Dateien im nativen Format für ein bestimmtes Betriebssystem. Die Kombination von Prozessor-Architektur und Betriebssystem wird auch **Plattform** genannt. Ein Go-Programm muss für jede **Plattform** neu gebaut werden.

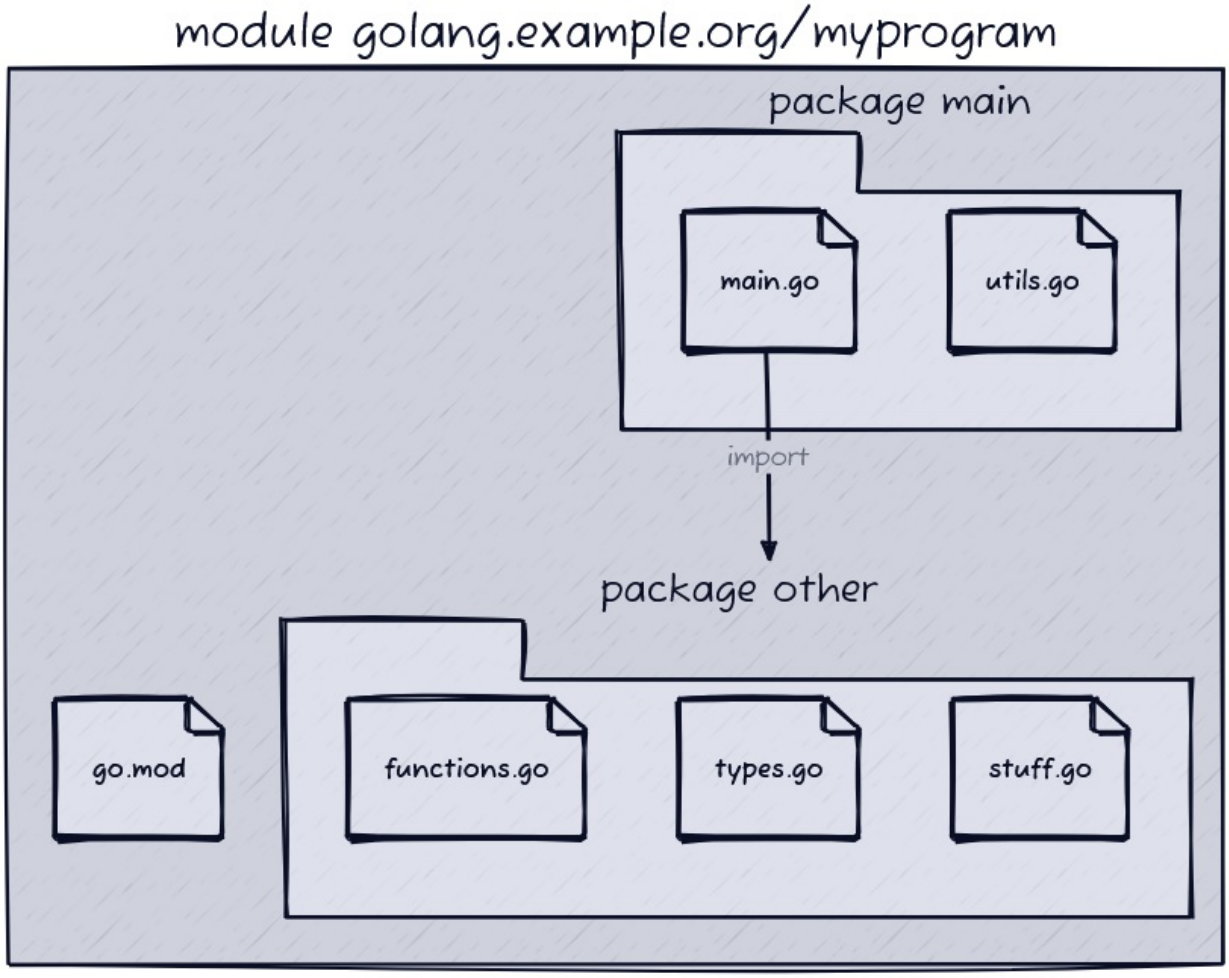


Packages und Imports

Alle Go-Quellcode-Dateien in einem Ordner müssen zum gleichen Package gehören. Sollen Funktionen, Konstanten, Variablen oder Typen aus einem anderen Package verwendet werden müssen diese durch Großschreibung exportiert und in das Ziel-Package importiert werden.

Gemäß Konvention entspricht der Paketname dem letzten Element des Importpfades. Zum Beispiel beinhaltet das Package `"math/rand"` Dateien, die mit der Anweisung `package rand` beginnen.

```
package main

import "fmt"
```



Die Main-Funktion

Jedes Go-Programm benötigt eine Funktion mit dem Namen `main` innerhalb des Packages `main`. Die Main-Funktion wird beim Start vom Betriebssystem ausgeführt und bildet den Eintrittspunkt in das Programm.

```
package main

import "fmt"

func main() {

}
```

Funktionsaufrufe

Aufrufe von Funktionen erfolgen über den Namen der Funktion und Klammern (`( )`). Innerhalb der Klammern können, durch Komma (`,`) getrennt, ein oder mehrere Parameter übergeben berden. Eine Funktion kann außerdem einen oder mehrere Werte zurückgeben.

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

Funktionsaufrufe können ineinander verschachtelt werden. Dabei wird der Rückgabewert des inneren Funktionsaufrufs als Parameter des äußeren Funktionsaufrufs verwendet.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Wurzel von 36 ist", math.Sqrt(36))
}
```

Variablen und Konstanten

Variablen können mit dem Schlüsselwort `var` als **globale Variablen** innhalb eines Packages oder als **lokale Variablen** innerhalb einer Funktion deklariert werden.

Auf **globale Variablen** kann von jeder Stelle des Programms zugegriffen werden und sie existieren solange wie das Programm läuft. Soll eine **globale Variable** auch außerhalb eines Packages für andere Packages sichtbar sein, muss diese durch Großschreibung exportiert werden.

Lokale Variablen sind nur innerhalb des Blocks in welchem sie deklariert wurden und dessen Unterblöcken gültig (**Block Scope**).

Bezeichner für Variablen können aus Groß- und Kleinbuchstaben (Umlaute sind möglich, sollten aber vermieden werden), Ziffern und Unterstrichen zusammengesetzt sein, dürfen allerdings nicht mit einer Ziffer beginnen. Außerdem darf kein Schlüsselwort als Bezeichner verwendet werden.

Mit dem Schlüsselwort `const` statt `var` können außerdem benannte Konstanten deklariert werden. Der Wert einer benannten Konstante muss ein bei der Kompilierung feststehender Wert sein und kann bei der Ausführung nicht mehr geändert werden.

```
package main

import "fmt"

// globale Variablen
var a, b, c int

func main() {
    // lokale Variablen
    var d int = 5
    var e = 7
    fmt.Println(a, b, c, d, e)
}
```

Mit dem Schlüsselwort `iota` lassen sich Konstanten mit fortlaufenden Nummern erzeugen.

```
package main

// globale Konstanten mit fortlaufender Nummer
const (
    Null = iota
    Eins
    Zwei
)
```

Datentypen

Variablen, Konstanten und Parameter besitzen in Go immer einen spezifischen Datentyp.

Datentyp	Länge	Beschreibung	Wertebereich	Nullwert
`bool`	-	Boolesche Wert	`true` oder `false`	`false`
`byte`	8 Bit	Byte (alias für `uint8`)	0 bis 255	`0`
`float32`	32 Bit	Kommazahl	1.401298e-45 bis 3.402823e+38	`0.0`
`float64`	64 Bit	Kommazahl mit doppelter Präzision	4.940656e-324 bis 1.797693e+308	`0.0`
`int`	min. 32 Bit	Ganzzahl	-9223372036854775808 bis 9223372036854775807 (bei 64 Bit)	`0`
`int8`	8 Bit	Ganzzahl (8 Bit)	-128 bis 127	`0`
`int16`	16 Bit	Ganzzahl (16 Bit)	-32768 bis 32767	`0`
`int32`	32 Bit	Ganzzahl (32 Bit)	-2147483648 bis 2147483647	`0`
`int64`	64 Bit	Ganzzahl (64 Bit)	-9223372036854775808 bis 9223372036854775807	`0`
`rune`	32 Bit	Unicode-Zeichen (alias für `int32`)	`'\u0000'` bis `'\uFFFF'`	`'\u0000'`
`string`	-	Zeichenkette	-	`""`
`uint`	min. 32 Bit	Ganzzahl ohne Vorzeichen (+/-)	0 bis 18446744073709551615 (bei 64 Bit)	`0`
`uint8`	8 Bit	Ganzzahl (8 Bit) ohne Vorzeichen (+/-)	0 bis 255	`0`
`uint16`	16 Bit	Ganzzahl (16 Bit) ohne Vorzeichen (+/-)	0 bis 65535	`0`
`uint32`	32 Bit	Ganzzahl (32 Bit) ohne Vorzeichen (+/-)	0 bis 4294967295	`0`
`uint64`	64 Bit	Ganzzahl (64 Bit) ohne Vorzeichen (+/-)	0 bis 18446744073709551615	`0`

Zahlensysteme

Name	Basis	Ziffern	Schreibweise in Go
Binär/Dual	2	`0`, `1`	`0b00000001`
Oktal	8	`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`	`077` (gekennzeichnet durch führende Null) oder `0o77`
Dezimal	10	`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`	`123`
Hexadezimal	16	`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`	`0x32`

Typumwandlung

Der Ausdruck `T(v)` wandelt den Wert `v` in den Typ `T` um. Im Gegensatz zu C bedarf es für die Zuweisung von Werten unterschiedlichen Typs immer einer expliziten Typumwandlung. Auf diese Weise lasen sich allerdings nur gleichartige Typen umwandeln, eine Umwandlung von `string` zu `int` oder `string` zu `bool` erfordert eine explizite Konvertierung. Das Package `strconv` stellt dafür u.a. Funktionen zur Verfügung.

```
package main

func main() {
    var a uint32
    var b int64 = 4762
    var c = uint8(55)

    a = uint32(b)
}
```

Formatierte Ausgabe

Das Package `fmt` enthält Funktionen für die Ausgabe auf die Konsole.

```
func fmt.Print(a ...any) (n int, err error)
func fmt.Printf(format string, a ...any) (n int, err error)
func fmt.Println(a ...any) (n int, err error)
```

Die Funktion `Printf` verwenden einen Formatstring mit sogenannten **Formatierungsverben** um Ausgaben zu formatieren.

```go
package main

import "fmt"

func main() {
    var num float64 = 3.14
    var str = "hallo"

    fmt.Printf("Werte: %s, %f\n", str, num)
}
```

Mit Flags lässt sich die Ausgabe der Verben weiter anpssen. So kann beispielsweise mit `%.2f` eine Kommazahl immer mit zwei Nachkommastellen ausgegeben werden oder mit `%-10s` ein String am Ende mit Leerzeichen "auffüllen" um immer mindestens 10 Zeichen lang zu sein.

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    var str = "Eulersche Zahl"

    fmt.Printf("%-20s: %.4f\n", str, math.E)
    fmt.Printf("%-20s: %#016b\n", "Max int16", math.MaxInt16)
}
```

Formatierungsverben

Verb	Beschreibung
%%	Ausgabe des %-Zeichens
%b	Ganzzahl im Binärsystem; `%#b` gibt `0b`-Prefix mit aus
%c	Unicode-Zeichen
%d	Ganzzahl im Dezimalsystem; Beispiel: `%08d` gibt immer 8 Stellen mit führenden Nullen aus
%e	Kommazahl in wissenschaftlicher Notation

Verb	Beschreibung
%E	Kommazahl in wissenschaftlicher Notation mit großem E
%f	Kommazahl; Beispiel: %.2f gibt immer zwei Nachkommastellen aus
%o	Ganzzahl im Oktalsystem; %#o gibt 0-Prefix mit aus
%O	Ganzzahl im Oktalsystem mit 0o-Prefix
%p	Pointer im Hexadezimalsystem
%q	String in Anführungszeichen, Byte-Slice als String in Anführungszeichen oder Unicode-Zeichen in Hochkommata
%s	String oder Byte-Slice als String; Beispiel: %10s oder %-10s füllt String mit Leerzeichen auf, wenn kürzer als 10
%t	Boolescher Wert
%T	Ausgabe des Datentyps
%U	Unicode notation eines Zeichens
%v	Ausgabe im Standard-Format (enspricht dem was mit Print ausgegeben wird); %+v gibt die Feldnamen bei struct aus und %#v gibt die Go-Syntax repräsentation eines wertes aus
%x	Ganzzahl oder Byte-Array/Slice im Hexadezimalsystem und Kleinbuchstaben; %#x gibt 0x-Prefix mit aus
%X	Ganzzahl oder Byte-Array/Slice im Hexadezimalsystem und Großbuchstagben; %#X gibt 0X-Prefix mit aus

Rechen- und Zuweisungsoperatoren

Operator	Beispiel	Beschreibung
+	x + y	Addition
-	x - y	Subtraktion
*	x * y	Multiplikation
/	x / y	Division
%	x % y	Modulo (Rest der ganzzahligen Division)
=	x = y	Zuweisung
+=	a += b	Additionszuweisung (a = a + b)
-=	a -= b	Subtraktionszuweisung (a = a - b)
*=	a *= b	Multiplikationszuweisung (a = a * b)
/=	a /= b	Divisionszuweisung (a = a / b)
%=	a %= b	Modulozuweisung (a = a % b)
++	a++	Inkrement (a = a + 1)
--	a--	Dekrement (a = a - 1)

Bitweise Operatoren

Operator	Beispiel	Beschreibung
&	x & y	AND
|	x | y	OR
^	x ^ y	XOR
^	^x	NOT (invertiert alle Bits)
<<	x << 8	Bit-Verschiebung nach Links
>>	x >> 8	Bit-Verschiebung nach Rechts

Package `math`

Das `math`-Package enthält Konstanten und mathematische Funktionen.

```
math.E  // Eulersche Zahl
math.Pi // Kreiszahl
func math.Ceil(x float64) float64
func math.Cos(x float64) float64  // Kosinus
func math.Floor(x float64) float64
func math.Log(x float64) float64
func math.Log10(x float64) float64
func math.Log2(x float64) float64
func math.Max(x, y float64) float64
func math.Min(x, y float64) float64
func math.Pow(x, y float64) float64
func math.Pow10(n int) float64
func math.Round(x float64) float64
func math.Sin(x float64) float64  // Sinus
func math.Sqrt(x float64) float64 // Quadratwurzel
func math.Tan(x float64) float64  // Tangens
func math.Trunc(x float64) float64
```

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    var r float64 = 7
    var flaeche = math.Pow(r, 2) * math.Pi
    fmt.Println("Kreisfläche", flaeche)
}
```

Außerdem sind in dem Unterpackage `math/rand` Funktionen zur Erzeugung von Zufallszahlen zu finden.

```
func rand.Float64() float64
func rand.Int() int
func rand.Intn(n int) int
```

Schleifen

Go kennt ausschließlich **kopfgesteuerte Schleifen**. Alle Schleifenvarianten verwenden das Schlüsselwort `for`.

Endlosschleife

Eine Endlosschleife lässt sich mit dem Schlüsselwort `break` beenden, läuft aber ansonsten endlos weiter.

```
package main

func main() {
    for {

    }
}
```

Schleifen mit Bedingung

Die Bedingung im Kopf wird geprüft und anschließend der Schleifenrumpf solange wiederholt bis die Bedingung nicht mehr erfüllt ist.

```
package main

import "fmt"

func main() {
    var a int = 5
    for a > 0 {
        fmt.Println(a)
        a--
    }
}
```

Zählschleifen

Bei der Zählschleifen wird zuerst im Kopf ein Initialwert gesetzt, eine Bedingung für die Wiederhohlung des Schleifenrumpfs festgelegt und eine Operation welche nach jedem Durchlauf ausgeführt angegeben.

```go
package main

import "fmt"

func main() {
    for i := 0; i < 10; i++ {
        fmt.Println(i)
    }
}
```

Schleifensteuerung

Mit dem Schlüsselwort `break` kann eine Schleife komplett abgebrochen werden und mit dem Schlüsselwort `continue` wird der aktuelle Durchlauf der Schleife beendet und zum nächsten Durchlauf gesprungen.

```go
package main

func main() {
    var num int
    for num < 10 {
        if num > 8 {
            break // gesamte Schleife beenden
        } else {
            num++
            continue // zur nächsten Interation springen
        }
    }
}
```

If-Else-Verzweigungen

Die `if`-Verzweigung führ Code nur aus wenn eine bestimmte Bedingung erfüllt ist. Klammern um die Bedingung sind nicht notwendig, aber die `{ }` müssen gesetzt werden.

Mit `else` lässt sich ein alternativer Pfad angeben welcher ausgeführt wird wenn die Bedingung nicht erfüllt ist. Außerdem lassen sich mit `else if` mehrere Bedingungen nacheinander prüfen und es kann jeweils unterschiedlicher Code ausgeführt werden. In einem solchen Fall könnte aber die `switch`-Anweisung besser geeignet sein.

```
package main

func main() {
    var a int = 5
    if a > 0 {

    } else if a > 100 {

    } else {

    }
}
```

Genau wie bei der Zählschleife mit `for` kann auch die `if`-Verzweigung mit einer kurzen Initial-Anweisung beginnen, die vor der `if`-Bedingung ausgeführt wird.

Variablen, die in dieser Anweisung deklariert werden sind nur bis zum Ende des `if`-Blocks aber auch innerhalb des `else`-Blocks sichtbar.

```
package main

import "os"

func main() {
    if err := os.Remove("loeschmich.txt"); err != nil {
        panic(err)
    }
}
```

Vergleichsoperatoren und logische Verknüpfungen

Operator	Beispiel	Beschreibung
==	x == y	Ist x gleich y?
!=	x != y	Ist x ungleich y?
<	x < y	Ist x kleiner als y?
<=	x <= y	ist x kleiner oder gleich y
>	x > y	Ist x größer als y?
>=	x >= y	Ist x größer oder gleich y?
&&	a && b	AND
||	a || b	OR
!	!true ist false	NOT

```go
package main

import "fmt"

func main() {
    var x = 5
    var truth bool = x < 7 && x > 2 || x * x == 25
    if truth {
        fmt.Println(truth)
    }
}
```

Switch-Case-Verzweigungen

Mehrfachverzweigungen lassen sich, alternativ zu `if else`-Blöcken auch mit dem Schlüsselwort `switch` realisieren.

Ein `case` beendet den `switch`-Block automatisch, außer er endet mit einer `fallthrough`-Anweisung.

`switch`-Verzweigungen werten die cases von oben nach unten aus und brechen ab, sobald ein `case` zutrifft.

Trifft kein `case` zu kann ein `default` ausgeführt werden.

```
package main

import "fmt"

func main() {
    var a = 5
    switch a {
        case 1:
            fmt.Println("Eins")
        case 2, 3, 4:
            fmt.Println("2, 3 oder 4")
        case 5:
            fallthrough
        case 6:
            fmt.Println("5 oder 6")
        default:
            fmt.Println("> 6")
    }
    switch {
        case a <= 5:
            fmt.Println("<= 5")
        case a > 5:
            fmt.Println("> 5")
    }
}
```

Arrays

Der Typ `[n]T` ist ein Array von `n` Werten des Typs `T`. Die Länge eines Arrays ist Teil seines Typs, die Größe eines Arrays kann also nicht verändert werden. Mit dem Index-Operator (`[n]`) kann auf die einzelnen Stellen des Arrays zugegriffen werden. Array-Indizes beginnen bei `0`.

```
package main

import "fmt"

func main() {
    var array [10]int = [10]int{7}

    array[1] = 5
    array[4] = 7

    fmt.Println(array[0], array[1], array[4])
}
```

Bei einem Array-Literal kann statt der Länge auch `...` angegeben werden. Dadurch erhält das Array die Länge welche sich aus der Anzahl der Elemente im Literal ergibt.

Die Funktion `len` ermitteln die Anzahl der Elemente eines Array zur Laufzeit. Das kann z.B. in einer Zählschleife genutzt werden um nacheinander auf alle Elemente zuzugreifen.

```
package main

import "fmt"

func main() {
    var array = [...]int{7, 9, 3, 4, 6}

    fmt.Println("Länge:", len(array))

    for index := 0; index < len(array); index++ {
        fmt.Println(array[index])
    }
}
```

Mehrdimensionale Arrays

Arrays können auch mehr als eine Dimension haben. Damit lässt sich z.B. eine Matrix realisieren.

```
package main

func main() {
    var array [3][3]byte = [3][3]byte{
        {0, 1, 1},
        {1, 0, 1},
        {1, 1, 0},
    }
}
```

Slices

Ein Array hat eine fixe Größe. Ein Slice dagegen hat eine dynamische Größe und bietet flexiblen Zugriff auf die Elemente eines Arrays. Der Typ `[]T` ist ein Slice mit Elementen des Typs `T`. In der Praxis kommen Slices wesentlich häufiger zum Einsatz als Arrays. Mit dem Slice-Operator `:` innehalb des Index-Operators (`[]`) lässt sich ein Slice auf einen Teil eines Arrays erzeugen. Wird debei weder Anfangs- noch Endindex angegeben, umfasst das Slice das gesamte Array.

Ein Slice speichert keine Daten, es beschreibt nur einen Abschnitt eines dahinterliegenden Arrays. Ändert man die Elemente eines Slices, ändert man damit die jeweiligen Elemente des dahinterliegenden Arrays. Andere Slices, die auf dem selben Array operieren, werden diese Änderungen ebenfalls sehen.

Ein Slice-Literal ist wie ein Array-Literal ohne Angabe der Länge.

```
package main

func main() {
    var array [5]int = [5]int{1, 2, 3, 4, 5}
    var slice1 []int = array[:]
    var slice2 []int = array[1:3]
    var slice3 []int = []int{6, 7, 9}
}
```

Slices dynamisch erzeugen

Neben der Erzeugung durch ein Slice-Literal, lassen sich Slices auch dynamisch mit der Funktion `make` erzeugen. Dabei muss eine Länge und es kann außerdem eine Kapazität angegeben werden. Es wird dann ein neues Array im Hintergund erzeugt, dessen größe der Kapazität des Slices entspricht. Mit der Funktion `cap` lässt sich im nachhinein die Kapazität eines Slices ermitteln. Die Funktion `len` ermittelt genau wie bei Arrays die Länge, also die Anzahl der Elemente eines Slices.

```
package main

func main() {
    var slice []int = make([]int, 0, 10)
}
```

Slices erweitern

Mit der Funktion `append` kann ein zusätzliches Element an ein vorhandenes Slice angehängt werden. Übertrifft dabei die Länge des Slices seine Kapazität, ist also das Array im Hintergrund zu klein, wird automatisch ein neues größeres Array im Hintergrund angelegt und der gesamte Inhalt des alten Arrays kopiert.

```
package main

func main() {
    var slice []int = make([]int, 1, 1)
    slice[0] = 5
    slice = append(slice, 7)
}
```

Package `slices`

Das Package `slices` enthält einige nützliche Hilfsfunktionen zum einfügen oder entfernen von Elementen aus Slices.

```
func slices.Concat[S ~[]E, E any](slices ...S) S
func slices.Contains[S ~[]E, E comparable](s S, v E) bool
func slices.Delete[S ~[]E, E any](s S, i, j int) S
func slices.Insert[S ~[]E, E any](s S, i int, v ...E) S
```

Range-Schleife

Mit dem Schlüsselwort `range` läuft eine Schleife über alle Elemente eines Arrays oder Slices. Die erste Variable enthält den Index und die zweite Variable den Wert am aktuellen Index. Wird der Index nicht benötigt kann als Variablenname _ verwendet werden, dadurch wird die Variable ignoriert.

```
package main

import "fmt"

func main() {
    var array [5]int = [5]int{4, 7, 8, 3, 1}
    for index, value := range array {
        fmt.Println(index, value)
    }
}
```

Wenn mit `range`-Schlüsselwort eine Schleife über eine Zeichenkette läuft, dann über die einzelnen Runen der Zeichenkette.

```
package main

import "fmt"

func main() {
    for _, char := range "Trüffelöl" {
        fmt.Printf("%c\n", char)
    }
}
```

Zeichenketten

Zeichenkettenliterale werden in Go mit " oder ` erzeugt. Außerdem wird ' für Literale einzelner Zeichen verwendet. Ein `string` in Go ist immutable, das bedeutet das, im gegensatz zu Arrays und Slices, einzelne Zeichen nicht geändert werden können. Wird ein `string` verändert, ensteht immer ein neuer `string` welcher ebenfalls wieder immutable ist.

Mit dem +-Operator können zwei Strings verbunden werden und mit den Vergleichsoperatoren lässt sich der Inhalt zweier Strings vergleichen.

```
package main

func main() {
    var str1 = "Hello" + " " + "world"
    var str2 = `Hello
                world`
    var char = 'H'
}
```

Escape-Sequenzen

Mit **Escape-Sequenzen** können u.a. *nicht-druckbare Zeichen* in Zeichen- und Zeichenkettenliteralen angegeben werden.

Escape-Sequenz	Beschreibung
\\	Backslash
\'	Hochkomma (nur in Zeichen-Literalen)
\"	Anführungszeichen (nur in String-Literalen)
\a	akustisches Signal
\b	Backspace
\f	Form feed
\n	Zeilenumbruch
\r	Carriage return (Bestandteil des Windows-Zeilenumbruchs "\r\n")
\t	Tabulator (horizontal)

Escape-Sequenz	Beschreibung
`\v`	Tabulator (vertikal)
`\x00`	Byte-Wert im Hexadezimalsystem mit zwei Stellen (1 Byte)
`\u0000`	Unicode-Zeichen im Hexadezimalsystem mit vier Stellen (2 Byte)
`\U00000000`	Unicode-Zeichen im Hexadezimalsystem mit acht Stellen (4 Byte)

Umwandlung von Strings in andere Datentypen

Ein `string` kann durch einen Type-Cast einfach in ein Byte-Slice (`[]byte`) umgewandelt werden und umgekehrt. Für die Umwandlung zu anderen Datentypen sind allerdings extra Funktionen notwendig. Zur Umwandlung von anderen Datentypen zu Strings kann u.a. die Funktion `Sprint` aus dem Package `fmt` verwendet werden. Für die Umwandlung von Strings zu anderen Datentypen gibt es im Package `strconv` entsprechende Funktionen.

```
func fmt.Sprint(a ...any) string
func fmt.Sprintf(format string, a ...any) string
func fmt.Sprintln(a ...any) string
func strconv.Atoi(s string) (int, error)
func strconv.ParseBool(str string) (bool, error)
func strconv.ParseFloat(s string, bitSize int) (float64, error)
func strconv.ParseInt(s string, base int, bitSize int) (i int64,
err error)
func strconv.ParseUint(s string, base int, bitSize int) (uint64,
error)
```

Package `strings`

Das Package `strings` enthält einige nützliche Hilfsfunktionen für die Verarbeitung von Strings.

```
func strings.Contains(s, substr string) bool
func strings.ContainsRune(s string, r rune) bool
func strings.Fields(s string) []string
func strings.HasPrefix(s, prefix string) bool
func strings.HasSuffix(s, suffix string) bool
func strings.Join(elems []string, sep string) string
func strings.Repeat(s string, count int) string
func strings.Replace(s, old, new string, n int) string
func strings.ReplaceAll(s, old, new string) string
func strings.Split(s, sep string) []string
func strings.SplitN(s, sep string, n int) []string
func strings.ToLower(s string) string
func strings.ToUpper(s string) string
func strings.Trim(s, cutset string) string
func strings.TrimSpace(s string) string
type strings.Builder
    func (b *Builder) String() string
    func (b *Builder) Write(p []byte) (int, error)
    func (b *Builder) WriteByte(c byte) error
    func (b *Builder) WriteRune(r rune) (int, error)
    func (b *Builder) WriteString(s string) (int, error)
```

Structs

Ein `struct` ist ein Verbund von Elementen mit unterschiedlichen Typen. Jedes Unterelement hat einen eigenen Namen. Der Zugriff auf die Elemente eines Structs erfolgt mit einem Punkt.

Ein Struct-Literal beschreibt einen neu allokierten Struct-Wert, indem eine Liste der Werte aller Elemente des Structs angegeben wird.

Man kann auch nur eine Untermenge der Elemente angeben. Das ist mit der `Name:`-Syntax möglich, die Reihenfolge der Elemente ist dabei irrelevant.

```
package main

type Adresse struct {
    Strasse    string
    Hausnummer string
    PLZ        string
}

func main() {
    var a Adresse = Adresse{
        Strasse:    "Frankenstr.",
        Hausnummer: "210",
        PLZ:        "90461",
    }

    fmt.Println(a.Strasse, a.Hausnummer)
    fmt.Println(a.PLZ)
}
```

Pointer

Ein **Pointer** (auch **Zeiger**) ist eine Variable welche auf die Speicheradresse einer anderen Variable zeigt. Der Typ `*T` ist ein Pointer auf einen Wert des Typs `T`. Der Nullwert von `*T` ist `nil`. Der `&`-Operator erzeugt einen Pointer auf seinen Operanden. Der `*`-Operator liefert den Wert auf den der Pointer verweist (**Dereferenzierung**). Im Gegensatz zu C gibt es in Go keine Pointerarithmetik.

Funktionen

Funktionen sind ein Block von Anweisungen mit einem Namen. Der Funktionsblock wird einmal definiert und dann können die Anweisungen innerhalb der Funktion über den Namen mehrfach an beliebigen Stellen immer wieder aufgerufen werden. Funktionen werden mit dem Schlüsselwort `func` erstellt. Sie können mehrere Werte als Parameter entgegenehmen. Außerdem können Funktionen in Go auch mehrere Werte zurückgeben. Bei mehr als einem Rückgabewert müssen die Typen der Werte mit Kommas (`,`) getrennt am Ende der Funktionssignatur in Klammern (`( )`) geschrieben werden. Mit dem Schlüsselwort `return` wird eine Funktion beendet und die Rückgabewerte werden festgelegt.

```
func Function(a, b, c int, str string) (int, int, int) {
    return c, b, a
}
```

Alternativ können Rückgabewerte auch benannt werden und Werte so auch ohne `return` zugewiesen werden. Jede Funktion die mindestens einen Rückgabewert hat muss auch ein `return` haben.

```
func Function(a, b, c int) (x, y, z int) {
    x, y, z = c, b, a
    return
}
```

Mit dem Schlüsselwort `defer` lassen sich Funktionsaufrufe innerhalb einer Funktion an deren Ende ausführen. Die Aufrufe werden auch in einem Fehlerfall noch ausgeführt und ermöglichen so ein Cleanup durchzuführen.

Call-by-Reference

Beim Aufruf einer Funktion werden die übergebenen Variablen kopiert, weshalb sich alle Änderungen an Parametern innerhalb der Funktion nicht auf die übergebenen Variablen auswirken. Dieses Konzept wird **Call-by-Value** genannt. Die Parameter einer Funktion können allerdings auch als Pointer deklariert werden wodurch ein schreibender Zugriff auf die übergebenen Variablen aus der Funktion heraus möglich wird. Dieses Konzept wird auch **Call-by-Reference** genannt.

Variable Argumentlisten (Varargs)

Mit `...` vor dem Typ eines Parameters einer Funktion ist es möglich Funktionen zu erstellen, welche eine beliebige Anzahl von Argumenten des gleiche Typs entgegen nehmen wie es z.B. bei den Print-Funktionen aus dem `fmt`-Package der Fall ist. Der Parameter wird dadurch zu einem Slice. Auch beim Aufruf einer solchen Funktion können mit `...` die Werte aus einem Slice einzeln als Argumente der Funktion übergeben werden.

```
package main

import (
    "fmt"
    "strings"
)

func multiRepeat(repeat int, texts ...string) {
    for _, text := range texts {
        fmt.Println(strings.Repeat(text, repeat))
    }
}

func main() {
    multiRepeat(5, "a", "b", "c")
    var args []string = []string{"Cat", "Dog", "Pig"}
    multiRepeat(3, args...)
}
```

Fehlerbehandlung

Fehler werden in Go durch das `error`-Interface representiert. Eine Funktion oder Methode bei der es beim Aufruf zu einem Fehler kommen kann, liefert als letzten bzw. einzigen Rückgabewert ein Objekt vom Typ `error`. Wenn dieser Fehlerwert nicht `nil` ist, gab es einen Fehler und es sollte im Code darauf reagiert werden. Mit der Funktion `Is` aus dem `errors`-Package kann geprüft werden um welchen Fehler es sich handelt.

```
package main

import (
 "errors"
 "fmt"
 "io/fs"
 "os"
)

var ErrNotFound = errors.New("not found")

func SomeFunction(fname string) error {
    if _, err := os.Open(fname); err != nil {
        if errors.Is(err, fs.ErrNotExist) {
            return fmt.Errorf("open %s: %w", fname, err)
        } else {
            return err
        }
    }
}
```

Eigene `error`-Objekte können mit der Funktion `New` aus dem `errors`-Package oder der Funktion `Errorf` aus dem `fmt`-Package erzeugt werden.

```
func errors.Is(err, target error) bool
func errors.New(text string) error
func errors.Unwrap(err error) error
func fmt.Errorf(format string, a ...any) error
```

Benutzereingaben auf der Konsole

Das Package `fmt` enthält Funktionen um Benutzereigaben von der Konsole in ein Variablen einzulesen. Die Funktion `Scanf` verwendet genau wie `Printf` einen Formatstring mit Platzhaltern um das Format der Eignaben bestimmen zu können.

```
func fmt.Scan(a ...any) (n int, err error)
func fmt.Scanf(format string, a ...any) (n int, err error)
func fmt.Scanln(a ...any) (n int, err error)
```

Die Ziel-Variable muss als Pointer angegeben werden.

```go
package main

import (
    "fmt"
)

func main() {
    var zahl int
    fmt.Scan(&zahl)
    fmt.Println(zahl)
}
```

Ein- und Ausgabe in Dateien

Eine Konsolenanwendung besitzt drei Ein- bzw. Ausgabekanäle: Standardeingabe (0 – **stdin**), (1 – **stdout**) und die Standardfehlerausgabe (2 – **stderr**). Diese werden als geöffnete Dateien durch die globalen Variablen `Stdin`, `Stdout` sowie `Stderr` aus dem Package `os` repräsentiert.

Dateien aus dem Dateisystem können mit den Funktionen `Open` oder `Create` aus dem Package `os` geöffnet werden. Offene Dateien sollten nach ihrer Verwendung mit der Methode `Close` wieder geschlossen werden.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    var (
        file   *os.File
        err    error
        buffer []byte = make([]byte, 1024)
        count  int
    )

    file, err = os.Open("text.txt")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    count, err = file.Read(buffer)
    if err != nil {
        panic(err)
    }

    fmt.Printf("%d bytes loaded\n\n", count)
    fmt.Println(string(buffer))
}
```

Das Package `os` enthält Funktionen und den Typ `File` für den Umgang mit Dateien. Von den Print- und Scan-Funktionen existieren auch Varianten mit dem Prefix *F* welche auf beliebige Dateien

angewendet werden können. Außerdem enthält das Package `bufio` Typen zum zeilenweise Einlesen von Text.

```
func fmt.Fprint(w io.Writer, a ...any) (n int, err error)
func fmt.Fprintf(w io.Writer, format string, a ...any) (n int,
err error)
func fmt.Fprintln(w io.Writer, a ...any) (n int, err error)
func fmt.Fscan(r io.Reader, a ...any) (n int, err error)
func fmt.Fscanf(r io.Reader, format string, a ...any) (n int, err
error)
func fmt.Fscanln(r io.Reader, a ...any) (n int, err error)
func os.ReadFile(name string) ([]byte, error)
func os.Create(name string) (*File, error)
func os.Open(name string) (*File, error)
type os.File
    func (f *File) Close() error
    func (f *File) Read(b []byte) (n int, err error)
    func (f *File) Write(b []byte) (n int, err error)
    func (f *File) WriteString(s string) (n int, err error)
func bufio.NewScanner(r io.Reader) *Scanner
type bufio.Scanner
    func (s *Scanner) Bytes() []byte
    func (s *Scanner) Scan() bool
    func (s *Scanner) Text() string
```

Kommandozeilenargumente

Über `os.Args` kann auf die dem Programm ei Start übergebenen Kommandozeilenargumente als Slice zugegriffen werden.

```go
package main

import (
    "fmt"
    "os"
)

/*
 * Aufruf: "demo.exe einz zwei"
 * Ausgabe: "Exe: demo.exe, First arg: eins"
 */
func main() {
    fmt.Printf("Exe: %s, ", os.Args[0])
    fmt.Printf("First arg: %s", os.Args[1])
}
```

Exit-Codes

Wird ein Go-Programm normal beendet, wird der Exit-Code `0` an das Betriebssystem übergeben. Mit der Funktion `os.Exit` kann das Programm jeder Zeit bendet werden und ein Exit-Code angegeben werden. Das Betriebssystem interpretiert alle Werte ungleich `0` als Fehlercodes.

```go
package main

import (
    "os"
)

func main() {
    os.Exit(42)
}
```

Datum und Uhrzeit

Ein Unix-Timestamp ist eine Möglichkeit Datum und Uhrzeit als `int` zu speichern. Die Zahl repräsentiert die Anzahl der vergangenen Sekunden seit dem 1. Januar 1970, 00:00 Uhr UTC. Das Startdatum wird auch als *"The Epoch"* bezeichnet.

Das Package `time` enthält Typen und Funktionen zum Umgang mit Zeitwerten.

```
func time.Sleep(d Duration)
type time.Duration
type time.Month
type time.Time
    func Now() Time
    func Parse(layout, value string) (Time, error)
    func (t Time) Format(layout string) string
    func (t Time) Local() Time
    func (t Time) Sub(u Time) Duration
```

Damit lassen sich u.a. Datums- und Zweitwerte in einem bestimmten Format ausgeben sowie Zeitdifferenzen berechnen.

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    var mondlandung, _ = time.Parse("2006-01-02 15:04:05 MST",
        "1969-07-24 16:50:35 UTC")
    fmt.Print("Die Mondlandung war am ")
    fmt.Print(mondlandung.Local().Format("02.01.2006"))
    fmt.Print(" um ", mondlandung.Local().Format(time.TimeOnly))
    var diffInTagen = time.Now().Sub(mondlandung).Hours() / 24
    fmt.Printf(" vor %.0f Jahren", diffInTagen/365)
}
```

Module

Mit dem Befehl `go mod init example.com/mymodule` des Go-Tools wird ein Verzeichnis zum Wurzelverzeicnis eines neuen Go-Moduls indem die Datei `go.mod` erstellt wird. Als Name für ein Modul wird meißtens die URL des Quellcode-Repositories im Internet (ohne `https://` bzw. `https://`) verwendet, oft nach dem Muster `github.com/<Benutzername>/<Repository>` (GitHub ist eine populäre Online-Platform für das Versionsverwaltungssystem Git).

Der Befehl `go mod edit -require=example.com/mylib@v1.0.0` fügt eine Abhängigkeit zu einem Modul (im Beispiel "example.com/mylib") mit der entsprechenden Versionsnummer (im Beispiel "v1.0.0") durch einen Eintrag in der Datei `go.mod` hinzu.

Mit dem Befehl `go mod download` lädt alle Abhängigkeiten in den Modul-Cache herunter. Falls eine Abhängigkeit nicht heruntergeladen werden soll sondern stattdessen aus dem lokalen Dateisystem verwendet werden soll, so kann mittels `go mod edit -replace=example.com/mylib=../mylib` ein lokale Pfad als Ersatz angegeben werden.

Mit den Befehlen `go mod edit -dropreplace=example.com/mylib` oder `go mod edit -droprequire=example.com/mylib` können einzelne Einträge wieder aus der Datei `go.mod` entfernt werden. Der Befehl `go mod tidy` bereinigt die Datei `go.mod` und entfernt u.a. nicht-verwendete Abhängigkeiten.

Am häufigsten wird der Befehl `go get -u golang.org/x/sys/windows` verwendet, der sowohl einen Eintrag in der Datei `go.mod` mit der neuesten Version hinzufügt als auch die Abhängigkeit in den Modul-Cache herunterlädt. Ein Ergebnis könnte die folgende `go.mod`-Datei sein.

```
module example.com/mymodule

go 1.23

require (
    example.com/mylib v1.0.0
    golang.org/x/sys v0.0.0-20220224120231-95c6836cb0e7
)

replace example.com/mylib => ../mylib
```

Links

- The Go Programming Language (https://go.dev/)
- Go by Example (https://gobyexample.com/)
- Go with Visual Studio Code (https://code.visualstudio.com/docs/languages/go)
- GoLand von JetBrains: Mehr als nur eine Go IDE (https://www.jetbrains.com/de-de/go/)
- Variable (Programmierung) - Wikipedia (https://de.wikipedia.org/wiki/Variable_(Programmierung))
- Typumwandlung - Wikipedia (https://de.wikipedia.org/wiki/Typumwandlung)
- Strukturierte Programmierung - Wikipedia (https://de.wikipedia.org/wiki/Strukturierte_Programmierung)
- Prozedurale Programmierung - Wikipedia (https://de.wikipedia.org/wiki/Prozedurale_Programmierung)
- Kontrollstruktur - Wikipedia (https://de.wikipedia.org/wiki/Kontrollstruktur)
- Feld (Datentyp) - Wikipedia (https://de.wikipedia.org/wiki/Feld_(Datentyp))
- Zeiger (Informatik) - Wikipedia (https://de.wikipedia.org/wiki/Zeiger_(Informatik))
- Go Standard Library (https://pkg.go.dev/std)
- Printf - Wikipedia (https://de.wikipedia.org/wiki/Printf)
- Unixzeit - Wikipedia (https://de.wikipedia.org/wiki/Unixzeit)