

Pointer in Go – Ausführliche Erklärung, Aufgaben & Lösungen

Dieses Dokument dient als Lern- und Nachschlagewerk für das Thema Pointer in der Programmiersprache Go. Es kombiniert Theorie, ausführliche Aufgabenstellungen und kommentierte Lösungen, um ein tiefes Verständnis für den Umgang mit Speicheradressen zu entwickeln.

1. Was sind Pointer?

Ein Pointer ist eine Variable, die nicht direkt einen Wert speichert, sondern die Speicheradresse einer anderen Variable. Dadurch kann direkt auf den ursprünglichen Speicher zugegriffen werden. In Go sind Pointer explizit und sicher gestaltet, es gibt keine Pointer-Arithmetik wie in C.

- Jede Variable liegt an einer bestimmten Adresse im Speicher
- Der Operator `&` liefert die Adresse einer Variable
- Der Operator `*` dereferenziert einen Pointer und greift auf den Wert zu
- Ein nil-Pointer zeigt auf keine gültige Speicheradresse

Pointer werden in Go vor allem verwendet, um Funktionen zu ermöglichen, Werte zu verändern, oder um große Datenstrukturen effizient weiterzugeben.

2. Aufgabenstellung

- 1 Aufgabe 1 – Wert über Pointer verändern:
Schreibe eine Funktion, die einen int-Wert über einen Pointer verdoppelt. Berücksichtige dabei den Fall, dass der Pointer nil ist.
- 2 Aufgabe 2 – Swap mit Pointern:
Implementiere eine Funktion, die zwei int-Variablen mithilfe von Pointern vertauscht. Ohne Pointer darf die Aufgabe nicht lösbar sein.
- 3 Aufgabe 3 – Struct Pointer:
Definiere ein Struct Person und erhöhe das Alter der Person über einen Pointer.
- 4 Aufgabe 4 – Speicher reservieren:
Erzeuge eine neue Person auf dem Heap und gib einen Pointer auf diese zurück.
- 5 Aufgabe 5 – Slice manipulieren:
Verdopple alle Werte eines Slices innerhalb einer Funktion.
- 6 Aufgabe 6 – Sichere Dereferenzierung:
Implementiere eine Funktion, die überprüft, ob ein Pointer nil ist, bevor er verwendet wird.
- 7 Aufgabe 7 – Pointer auf Pointer:
Ändere einen Wert über einen Pointer auf einen Pointer.
- 8 Aufgabe 8 – Pointer erzeugen:
Gib einen Pointer auf eine neu erzeugte int-Variable zurück.

3. Lösungen mit Erklärung der Schlüsselkonzepte

DoubleViaPointer

```
func DoubleViaPointer(p *int) {
if p == nil {
return
}
*p *= 2
}
```

Erklärung: Der Pointer p wird zuerst auf nil geprüft. Mit *p wird der gespeicherte Wert dereferenziert und direkt im Speicher verändert.

Swap

```
func Swap(a, b *int) {
if a == nil || b == nil {
return
}
temp := *a
*a = *b
*b = temp
}
```

Erklärung: Beide Pointer zeigen auf Speicheradressen. Durch Dereferenzierung (*a, *b) werden die Originalwerte getauscht, nicht nur Kopien.

Birthday

```
func Birthday(p *Person) {
if p == nil {
return
}
p.Alter++
}
```

Erklärung: Structs werden über Pointer verändert. Go erlaubt den vereinfachten Zugriff p.Alter anstatt (*p).Alter.

NewPerson

```
func NewPerson(name string, alter int) *Person {
return &Person{Name: name, Alter: alter}
}
```

Erklärung: Mit &Person; {...} wird Speicher reserviert und ein Pointer auf das Struct zurückgegeben.

DoubleAll

```
func DoubleAll(s []int) {
for i := range s {
s[i] *= 2
}
}
```

Erklärung: Slices enthalten intern bereits einen Pointer auf ein Array. Änderungen wirken sich daher direkt auf das Original aus.

SafeDeref

```
func SafeDeref(p *int) (int, bool) {
if p == nil {
return 0, false
}}
```

```
}
```

```
return *p, true
```

```
}
```

Erklärung: Durch die nil-Prüfung wird eine Panic verhindert. Der bool-Wert signalisiert, ob der Zugriff erfolgreich war.

SetThroughDoublePointer

```
func SetThroughDoublePointer(pp **int, v int) {
```

```
if pp == nil || *pp == nil {
```

```
return
```

```
}
```

```
**pp = v
```

```
}
```

Erklärung: `**pp` dereferenziert zwei Pointer-Ebenen und greift auf den ursprünglichen int-Wert zu.

IntPtr

```
func IntPtr(v int) *int {
```

```
x := v
```

```
return &x;
```

```
}
```

Erklärung: `x` ist eine lokale Variable, deren Adresse zurückgegeben wird. Go sorgt automatisch dafür, dass sie auf dem Heap liegt (Escape Analysis).