Teil 3 Go Programmierung

Mohammadimohammadi, Nima

Agenda Teil 3

1) Binärzahlen
2) Arrays
3) Slices

Binärzahlen (uint)

- Eine Binärzahl **x** besteht aus **n** Bits

$$Bit = x_i \in \{0, 1\} \qquad x_i = \begin{cases} 1 & Bit\ aktiviert \\ 0 & Bit\ deaktiviert \end{cases} \qquad x = x_{n-1} x_{n-2} \dots x_1 x_0$$

$$Dezimalwert(x) = x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0 \qquad \sum_{i=0}^{n-1} x_i \cdot 2^i$$

Beispiel n = 8 `(uint8, byte)`

$$0_7 0_6 0_5 0_4 0_3 0_2 0_1 0_0$$

= 0 * 2^7 + 0 * 2^6 + … + 0 * 2^1 + 0 * 2^0
= 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0
= 0

Beispiel n = 8 `(uint8, byte)`

$$0_7 1_6 0_5 0_4 0_3 0_2 1_1 0_0$$

= 0 + 2^6 + 0 + 0 + 0 + 0 + 2^1 + 0
= 64 + 2 = 66

Binärzahlen Beispiel

$$x_i = 1 \ : 2^i$$
$$x_i = 0 \ : 0$$

$$x_{n-1}\, x_{n-2} \cdots x_2\, x_1\, x_0$$

was ist mit negativen zahlen?

0000 0101
= 5

101
= 5

0000 1111
= 15

1000 0000
= 128

1111 1111
= 255

0000 1010
= 10

1000 1010
= 138

10 1010
= 42

1001 1001
= 203

0001 1111
= 31

Negative Binärzahlen (int)

- Eine Binärzahl **x** besteht aus **n** Bits

$$Bit = x_i \in \{0, 1\} \qquad x_i = \begin{cases} 1 & Bit\ aktiviert \\ 0 & Bit\ deaktiviert \end{cases} \qquad x = x_{n-1} x_{n-2} \dots x_1 x_0$$

$$Dezimalwert(x) = x_{n-1} \cdot -2^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

N-1 Bit wird negativ betrachtet

Beispiel n = 8 `int8`

$$1_7 0_6 0_5 0_4 0_3 0_2 0_1 1_0$$

$-2^7 + 0 + 0 + 0 + 0 + 0 + 0 + 2^0 = -128 + 1 = -127$

Binärzahlen Beispiel

$$x_i = 1 \ : 2^i$$
$$x_i = 0 \ : 0$$

$$x_{n-1}\, x_{n-2} \cdots x_2\, x_1\, x_0$$

Ausnahme: $x_{n-1} = -2^{n-1}$

1000 0101 (uint8)
= 132

1101 (int4)
= -3

1111 1111 (int8)
= -1

1000 0000 (int32)
= 128

1111 1111(uint64)
= 255

0000 1010 (uint8)
= 10

1000 1010
= Nicht Möglich

10 1010 (int6)
= -22

1001 1001
= Nicht Möglich

0001 1111
= 31

Warum Umrundung bei Overflows

Beispiel:

-	0000		1111 1111 (uint8) = 255
-	0000	+	0000 0001 (uint8) = 1
-	0001		0000 0000 (uint8) = 0

-	0000		0111 1111 (int8) = 127
-	0000	+	0000 1010 (int8) = 10
-	0000		1000 1001 (int8) = -119

Exkurs: Dezimalzahl zu Binär

Beispiel: 204

204	/	2 =	102	R.0
102	/	2 =	51	R.0
51	/	2 =	25	R.1
25	/	2 =	12	R.1
12	/	2 =	6	R.0
6	/	2 =	3	R.0
3	/	2 =	1	R.1
1	/	2 =	0	R.1

= 11001100
= 1 1 0 0 1 1 0 0
= 2^7 + 2^6 + 0 + 0 + 2^3 + 2^2 + 0 + 0
= 128 + 64 + 0 + 0 + 8 + 4 + 0 + 0
= 204

Exkurs: Dezimalzahl zu Binär

-304 (int16)

1) 304

2)

304	/	2 =	152	R.0
152	/	2 =	76	R.0
76	/	2 =	38	R.0
38	/	2 =	19	R.0
19	/	2 =	9	R.1
9	/	2 =	4	R.1
4	/	2 =	2	R.0
2	/	2 =	1	R.0
1	/	2 =	0	R.1

3)
```
= 0000 0001 0011 0000
= 1111 1110 1100 1111
```

4)
```
      1111 1110 1100 1111
+     0000 0000 0000 0001
=     1111 1110 1101 0000
```

1) Vorzeichen entfernen
2) Berechne Bits
3) Binärzahl invertieren
4) Binärzahl + 1

Exkurs: Basis

- Eine Zahl **x** in einer bestimmten Basis **B**, hat **n** Stellen

$$x = x_{n-1} x_{n-2} \dots x_1 x_0$$

$$Stelle = x_i \in \{0, 1, \dots, Basis - 1\}$$

$$Dezimalwert(x) = x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \dots + x_1 \cdot B^1 + x_0 \cdot B^0$$

Beispiel: (Basis 5)

- $0_7 0_6 0_5 0_4 4_3 0_2 0_1 0_0$

$0 + 0 + 0 + 0 + 4 * 5^3 + 0 + 0 + 0$
$= 2500$

Beispiel: (Basis 2)

- $0_7 0_6 0_5 0_4 0_3 1_2 1_1 0_0$

$0 + 0 + 0 + 0 + 0 + 1 * 2^2 + 1 * 2^1$
$4 + 2 = 6$

Exkurs: Basis

Beispiel: (Basis 10)

– $0_7 0_6 0_5 0_4 4_3 0_2 0_1 0_0$

0 + 0 + 0 + 0 + 4 * 10³ + 0 + 0 + 0
= 4000

Beispiel: (Basis 16)

– $0_7 0_6 0_5 0_4 B_3 A_2 1_1 0_0$

11*16³ + 10 * 16² + 1 * 16¹ + 0 = 47632

A: 10
B: 11
C: 12
D: 13
E: 14
F: 15

Binäre Operation

- Zahlen sind als Binärzahlen abgespeichert
- Rein Binär zu Arbeiten kann sehr viel Speicher sparen
- Binär zu Arbeiten ist die performanteste Operation einer CPU

```
Binäre Literale  Basis 2 :  0b0000 // 0b
Hex    Literale  Basis 16:  0xAFFE // 0x
Oct    Literale  Basis 8 :  0o70   // 0o
```

Binäre Operationen

Binäres OR : x | y
Binäres AND : x & y
Binäres XOR : x ^ y
Binäres NOT : ^ x

A	B	NOT A	NOT B
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Beispiel: Binäre Operation

```
    0000 0001          0000 0001          0000 0001
&   0001 1001      |   0001 1001      ^   0001 1001
-------------------------------------------------
    0000 0001          0001 1001          0001 1000
```

```
^   0000 0001
-------------
    1111 1110
```

Binäres Shiften

Binäres Shift Rechts : `x >> y`
Binäres Shift Links : `x << y`

- Shifte um y stellen

```
0001 0001 >> 2  = 0000 0100
0000 1001 << 4  = 1001 0000
```

Beispiele:

```
fmt.Println(0b00001111 << 4)
// 1111 0000
fmt.Println(0b00001111 >> 2)
// 0000 0011
fmt.Println(0b00001111 & 2)
// 2
fmt.Println(8|4)
// 12 == 8 + 4
fmt.Println(8|15)
// 15
```

Beispiel Binär Operation

```
const (
    READ_ROLE    = 1        // 0001
    WRITE_ROLE   = 1 << 1   // 0010
    UPDATE_ROLE  = 1 << 2   // 0100
    DELETE_ROLE  = 1 << 3   // 1000
)

myProfile := READ_ROLE | WRITE_ROLE | DELETE_ROLE // 1011

if (0 != (myProfile & UPDATE_ROLE)) {
    fmt.Println("Profile has update permission")
}else{
    fmt.Println("Access Denied")
}
```

Einführungs von Arrays

Definition: Eine Sequenz von Daten eines Datentyps `T`, welche `N` Datensätze enthält.

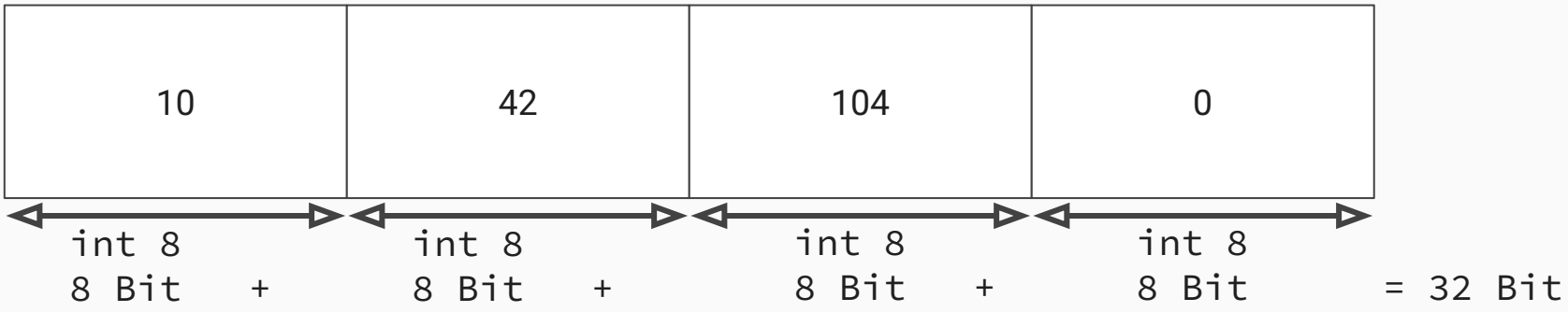
- N : Anzahl der Datensätze
- T: Datentyp aller Datensätze

Beispiel: n = 4, T: noch nicht erteilt

Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4

Beispiel von Arrays

Beispiel: n = 4, T: int8



- Datentypen Auswahl sind bei Arrays wichtig
- So klein wie möglich wählen um RAM zu sparen

Index eines Array

Beispiel: n = 4, T: int8

10	42	104	0

Als **Index** ist eine ganzzahlige Positionsangabe, die verwendet wird, um auf ein bestimmtes Element innerhalb eines Arrays zuzugreifen. Der Index ermöglicht das **Lesen (Zugreifen)** und **Schreiben (Verändern)** von Werten an einer bestimmten Stelle im Array.

$$Index \in \{0, \ldots, n-1\}$$

- Index = 0: greift auf die 10 (erstes Element)
- Index = 3: greift auf die 0 (letztes Element)

Verboten : Index < 0 oder Index >= n

Arrays in Go

Syntax:

```
var <name> [N]<type>
```

- `N` : Anzahl Elemente
- `<type>`: Element Typen

- Alle **`N`** Elemente sind in der Oben Genannten Syntax per **Default** initialisiert worden.
- `Zahlentypen = 0,  bool = false,  string = ""`
- Arrays sind unveränderlich, bedeutet Arrays wachsen nicht und können nicht verkleinert werden.
- Nach der Deklaration hat ein Array, N plätze bis das Array freigegeben wurde

Arrays in Go mit Initialisierung

Syntax:

```
    var <name> ([N]<type>)|(= [N]<type>{<wert1>,<wert2>,... })
```

Alternative:

```
        <name> := [N]<type>{<wert1>,<wert2>,... }
```

- `N` : Anzahl Elemente
- `<type>`: Element Typen

Die Anzahl der Werte zur Initialisierung muss kleiner gleich N sein.

Beispiel:

```
arr :=   [3]int8{1, 2, 3}   => [ 1 2 3 ]
arr :=   [2]int8{1, 2, 5}   => // FEHLER !
arr := [100]bool{true}      => [ true false false ...
```

Fazit Array Deklarationen

1) `var <name> [N]<type>`
Mit Angabe des Typ und alles per default initialisiert

2) `var <name> [N]<type> = [N]<type>{<wert1>,<wert2>,... }`
Mit Angabe des Typs und selbst initialisiert

3) `var <name> = [N]<type>{<wert1>,<wert2>,... }`
Typ wird abgeleitet und selbst initialisiert

4) `<name> := [N]<type>{<wert1>,<wert2>,... }`
Typ wird abgeleitet und selbst initialisiert

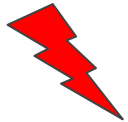
4.1) `<name> := [...]<type>{<wert1>,<wert2>,... }`
N wird automatisch ermittelt anhand der Anzahl der Initialisierung und wie 4)

Beispiel Array Deklaration

```
var arr [100]int8
```

- array mit 100 elemente jeweils 0

```
const arr [1000]int8
```

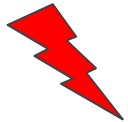


- `const` Arrays gibt es nicht

```
var arr [10]int32{-30, 20 }
```

- diese Syntax gibt es nicht

```
var arr [2]bool = {true}
```



- Initialisierung Typ nicht angegeben, [2]bool{...}

```
var arr [0]bool
```

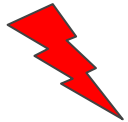
- Geht, aber nicht sinnvoll

Beispiel Array Deklaration

```
var x = [100]int8
```

- {} fehlen , [100]int8{}

```
const y = [100]byte{}
```

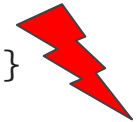


- const Arrays gibt es nicht, mit var gültig

```
var arr = [1]int32{2025}
```

- Gültig

```
var x [2]bool = [1]bool{true}
```



- Initialisierung Typ unterscheidet sich

```
var arr = [...]bool{}
```

- Geht, aber nicht sinnvoll

Mit Arrays Arbeiten

- Um ein Array lesend oder schreibend zuzugreifen brauchen wir ein Index

  $Index \in \{0, \ldots, n-1\}$

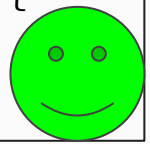
```
var arr  = [...]int32{42, 3, 110}

Schreibender Zugriff:

arr[0] = 10
arr[0]++ // arr[0] = arr[0] + 1

Lesender Zugriff:

fmt.Println(arr)    // [ 11 3 110 ]
fmt.Println(arr[0]) // 11
```

Beispiel Array

```
for i := 0 ; i < 100; i++ {
    fmt.Println(i + 1)
}
```

```
for i := 0 ; i < 100; i++ {
    fmt.Println(arr[0])
}
```

```
for i := 0 ; i < 100; i++ {
    fmt.Println(arr[i])
}
```



```
var arr  = [100]int32{}

for i := 0 ; i < 100; i++ {
    arr[i] = i + 1
}
```

- Finden wir die Schleifen gut?
  - 100 hardcoded in der Schleife
  - nicht flexibel

Iteration über Arrays

1. Ansatz: hardcoded in der Schleife

```
var arr1  = [100]int32{}
var arr2  = [500]int32{}

for i := 0 ; i < 100; i++ {
    fmt.Println(arr1[i])
}

for i := 0 ; i < 500; i++ {
    fmt.Println(arr2[i])
}
```

2. Ansatz: mit len(...)

len(...) : Liefert Anzahl der Elemente vom Argument

```
var arr1  = [100]int32{}
var arr2  = [500]int32{}

for i := 0 ; i < len(arr1); i++ {
    fmt.Println(arr1[i])
}

for i := 0 ; i < len(arr2); i++ {
    fmt.Println(arr2[i])
}
```

Iteration über Arrays

3. Ansatz For Range

```
Syntax:

for <index>, <value> := range <collection> {
     <Code>
}
```

<index>: Index Variable liefert in jede Iteration
den aktuellen index

<value>: Value Variable, liefert in jede Iteration
den aktuellen Value

<collection> : Sammlung
(z.B. Array, String, Map, Slices, uvw.)
keine Primitiven Typen(int, float, bool)

```
3. Ansatz: mit Range Ausdrücke

var arr1  = [100]int32{}
var arr2  = [500]int32{}

for index, value := range arr1 {
    fmt.Println(value)
}

for _, value := range arr2 {
    fmt.Println(value)
}
```

Range Ausdrücke

Noch Paar Worte zu Range Ausdrücke

- range wird in for-Schleifen verwendet.
- Liefert pro Iteration ein oder zwei Werte: **Index** und **Wert**.
- Wird genutzt für: **Arrays, Slices, Strings**

```
nums := []int{10, 20, 30}

for i, v := range nums {

    fmt.Printf("Index: %d -> Wert: %d\n",i, v)

}
```

Array Zuweisungen

Zuweisungen mit Array Variablen wird ein tatsächlicher **copy by value** gemacht.

```
x := [5]int32{0,1,2,3,4}
y := [5]int32{}

y = x
```

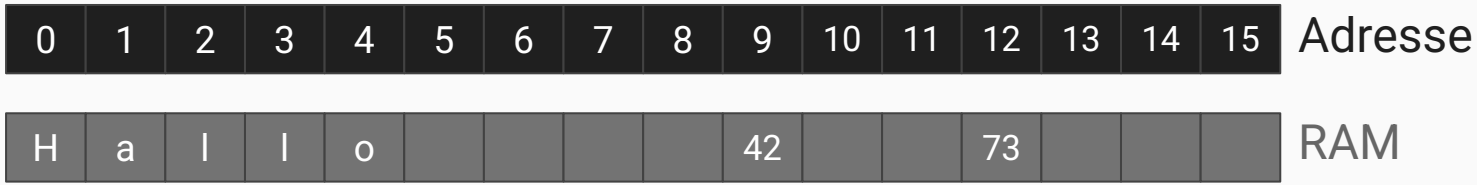
Bei dieser Zuweisung von y = x wird ein tatsächlicher Kopie gemacht. Somit erhalten beide Arrays zwei getrennte Speicherbereiche.

Gegensatz zum **copy by value** ist das **copy by reference**.

RAM <-> Adresse

Eine Adresse ist eine Zahl, die angibt, wo im Arbeitsspeicher (RAM) ein Wert gespeichert ist.

```
var str = "Hallo"
var x = 42
...
var y = 73
```

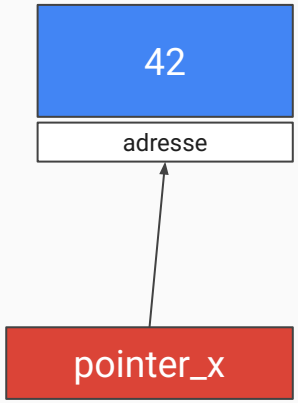


Was sind Pointer?

Wenn wir von Pointer reden, ist das nichts anderes als eine Variable die eine Adresse enthält.

- Vertiefung folgt !

```
&<var> : Adressen Operator
*<var> : Dereferenzierung

var x int32 = 42
var pointer_x = &x

fmt.Println(pointer_x) // 0xc0000101c0
fmt.Println(*pointer_x) // 42

*pointer = 5

fmt.Println(x) // 5
```



Arrays in Go

Arrays haben ein wichtige Einschränkung: Ihre Größe ist **fix**.

- Ein Array kann nach der Deklaration nicht wachsen oder schrumpfen.

Wie können wir dieses Problem umgehen?

1) Array größer machen als nötig
    a) Nachteil:
        i) Ungenutzter Speicher
        ii) Wie groß wählen wir die größe?

Mit Arrays allein werden wir nicht glücklich!
=> Wie können wir dynamische Arrays erstellen

Was sind Slices?

Ein Slice ist eine flexible Version eines Arrays.

- Es kann wachsen oder schrumpfen.
- Es sieht fast genauso aus wie ein Array.
- Es ist wie ein Ausschnitt (Slice) aus einem größeren Array.

Intern merkt sich ein Slice:

- Wo die Daten im Speicher anfangen.(Pointer)
- Wie viele Elemente es aktuell enthält (Länge).
- Wie viel Platz insgesamt noch frei ist (Kapazität).

Slice

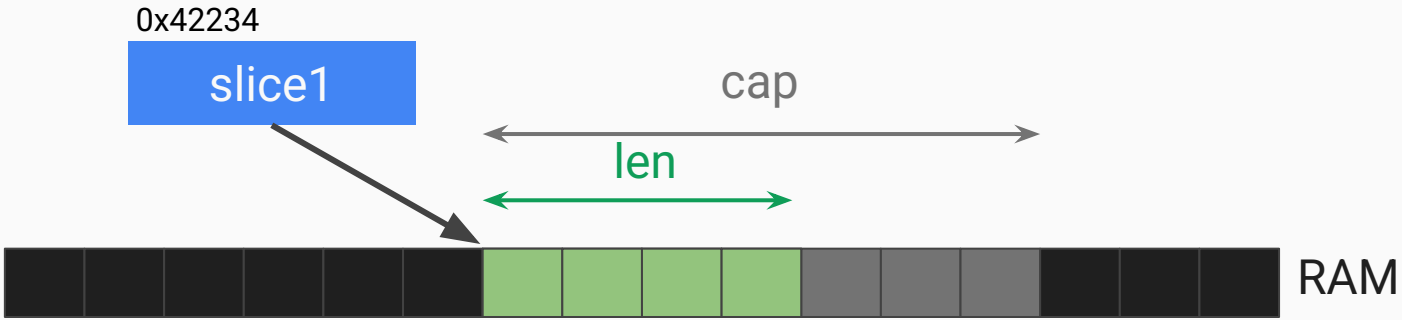
Ein Slice besteht intern aus drei Bestandteilen:

- Ein Zeiger auf ein Array
- Eine Länge (len), wie viele Elemente sind aktuell sichtbar
- Eine Kapazität (cap), wie viele Elemente ab der Startposition im Array noch genutzt werden können.

cap = 8, len = 4

"slices under the hood"

```
var slice1 = <slice deklaration>
```



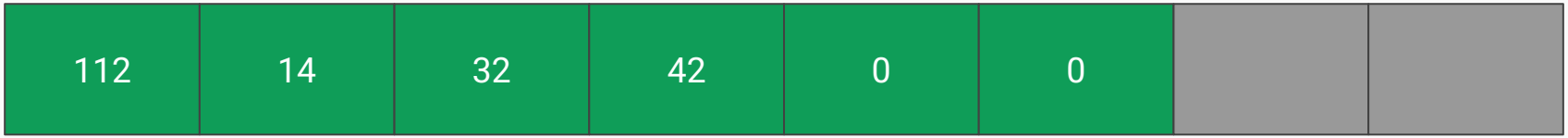
Merke: Ein Slice ist kein Array, sondern eine Referenz auf einen Teil eines Arrays – mit eigener Länge und Kapazität.

Beispiel Slice

Slice: n = 4, cap = 8 mit int8, Adresse: 17203

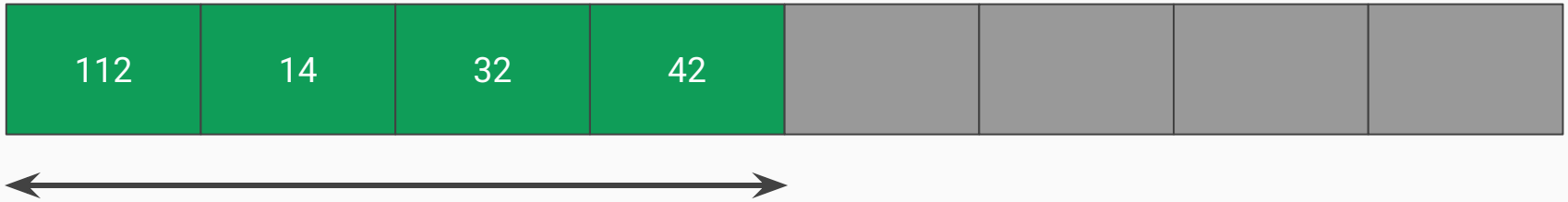
- Du hast nur Zugriff auf Index 0 - 3
- Jetzt willst du dein Slice vergrößern um 2 Stellen und weil die Kapazität noch ausreichend war, hat dein Slice effizient es vergrößert.
- Jetzt haben wir noch zwei Felder zum Arbeiten

Slice: n = 6, cap = 8, Adresse: 17203



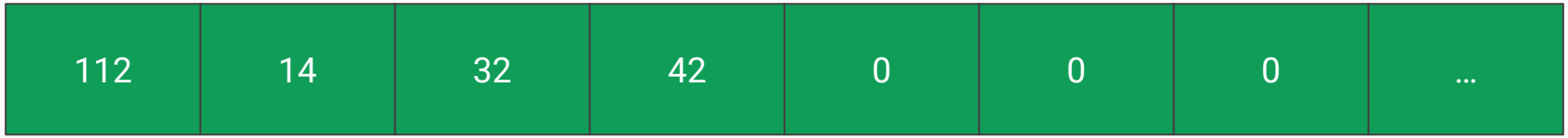
Beispiel Slice

Slice: n = 4, cap = 8 mit int8, Adresse: 17203



- Du hast nur Zugriff auf Index 0 - 3
- Jetzt willst du dein Slice vergrößern um 10 Stellen.
- Leider muss jetzt das Betriebssystem, ein neuen geeigneten Speicherbereich finden.
  => langsam, viel Aufwand

Slice: n = 14, cap = 16 mit int8, Adresse: 23422



Deklaration von Slices

1) `var <name> []<type>`
Mit Angabe des Typ und alles per default initialisiert, cap = len = 0

2) `var <name> []<type> = []<type>{<wert1>,<wert2>,... }`
Mit Angabe des Typs und selbst initialisiert, cap = len = Anzahl der Initialisierungs Werten

3) `var <name> = []<type>{<wert1>,<wert2>,... }`
Typ wird abgeleitet und selbst initialisiert, cap = len = Anzahl der Initialisierungs Werten

4) `<name> := []<type>{<wert1>,<wert2>,... }`
Typ wird abgeleitet und selbst initialisiert, cap = len = Anzahl der Initialisierungs Werten

5) `make([]<typ>, N, C):`
Liefert ein Slice mit den Typ `<typ>` und eine Länge `N` und eine Kapazität `C`

Mit Slices Arbeiten

- Um ein Slice lesend oder schreibend zuzugreifen - braucht ein Index

  $Index \in \{0, \ldots, n-1\}$

- Über ein Slice zu iterieren ist identisch wie beim Array

```
var slice  = []int32{42, 3, 110}

Schreibender Zugriff:

slice[0] = 10
slice[0]++ // slice[0] = slice[0] + 1

Lesender Zugriff:

fmt.Println(slice)    // [ 11 3 110 ]
fmt.Println(slice[0]) // 11
```

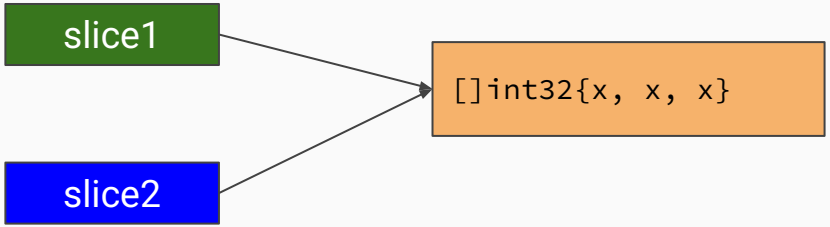
Copy by Reference

Wenn eine Zuweisung unter Referenztypen besteht, tritt das sogenannte **copy by reference**.

In diesem Beispiel entsteht kein neuer Slice, sondern `slice1` und `slice2` werden zum gleichen Objekt.

Wir kopieren nur die Referenz von slice1

```
var slice1 = []int32{42, 3, 110}
var slice2 = slice1

slice2[0]++

fmt.Println(slice1[0] == slice2[0])
```



Slices Vergrößern/Verkleinern

1. Mit append(): Werte hinzufügen oder anderes Slice anhängen
    a. **Kann** ein neuen Slice erstellen abhängig vom Kapazität
        i. Bekommt ein neuen Bereich
        ii. Bleibt im selben Bereich

2. Slice mit make(): neu anlegen und kopieren
    a. Erstellt **immer** ein neuen Slice, somit **neuer eigener Bereich**

Append

- append(s, v) : Füge an Slice s ein neuen Wert V an
  Beispiel:

```
x := []int{1, 2, 3}
x = append(x, 4)   // [1 2 3 4]
```

- append(s, x, y, z, ...): Füge an Slice s mehrere Werte an
  Beispiel:

```
x = append(x, 5, 6, 7) // [1 2 3 4 5 6 7]
```

- append(s, otherSlice...)
  Beispiel:

```
t := []int{8, 9}
x = append(x, t...) // [1 2 3 4 5 6 7 8 9]
```

Mit Make arbeiten

`make([]<typ>, N, C):`

Liefert ein Slice mit den Typ <typ> und eine Länge N und eine Kapazität C und alle Elemente sind Default initialisiert.

```
old := []int{1, 2, 3}
newSlice := make([]int, len(old) + 2, 10) // mehr Kapazität

for i, value := range old {
     newSlice[i] = value
}

// Alternative:  copy(dst, src) dst: ziel_slice, src: quelle_slice
// copy(newSlice,old)

new slice[3] = 4
new slice[4] = 5

fmt.Println(newSlice) // [ 1 2 3 4 5 ]
```

Slice Operator

Der Slice-Operator liefert von einem Array/Slice einen neuen Slice von den Bereich start und end

- Syntax:
  `s[<start>:<end>]`
  Mit max:
  `s[start:end:max]`

Teilnehmer:

<start>: Das Element am start-Index ist das erste Element im neuen Slice.
<end> : Der neue Slice enthält die Elemente bis zum end-Index, aber schließt diesen selbst nicht mit ein.
<max> Ein Index im ursprünglichen Slice, der das Ende der Kapazität des neuen Slices festlegt.
s: Array/Slice.

Regel: 0 <= start <= end <= max <= cap(s)

Beispiel Slice Operator

```
x := []int{1, 2, 3, 4}
copy(x[1:4], x)
fmt.Println(x)
```

```
src := []int{9, 8, 7}
dst := make([]int, 0)
n := copy(dst, src)
fmt.Println(n, dst)
```

```
a := []int{1, 2, 3}
b := append(a[0:1], 99)
fmt.Println("a:", a)
```

```
x := []int{10, 20, 30, 40, 50}
y := x[1:3]
fmt.Println(y, cap(y))

y = append(y, 99)
fmt.Println(x)
```

```
x := []int{100, 200, 300, 400, 500}
a := x[1:4:4]
append(a, 450)

fmt.Println(a)
```

Beispiel Slice Operator

```
x := [2]int32{0,1}
x_slice := x[0:2]
fmt.Println(x_slice)
```

```
x := []int32{0,1,2,3,4,5}
x_slice := x[0:]
fmt.Println(x_slice)
```

```
x := []int32{0,1,2,3,4,5}
x_slice := x[4:]
fmt.Println(x_slice)
```

```
x := [100]int32{0,1,2,3,4,5}
x_slice := x[:]
fmt.Println(x_slice)
fmt.Println(cap(x_slice))
fmt.Println(len(x_slice))
```

```
a := [100]int{10, 20, 30, 40, 50}
s := a[1:4:5]
fmt.Println(s)
fmt.Println(cap(s))
fmt.Println(len(s))
```

Demo Append und Slicing

https://github.com/nmm-4/learn-go/tree/main/ueb03/vl/main.go

Fazit

1. append()
   a. **Kann** ein neuen Slice erstellen, abhängig der Kapazität
      i. Bekommt ein neuen Bereich
      ii. Bleibt im selben Bereich

2. make()
   a. Erstellt **immer** ein neuen Slice, somit **neuer eigener Bereich**

3. Slice Operator
   a. Referenziert den angegebenen Bereich, **kein eigenen Bereich**