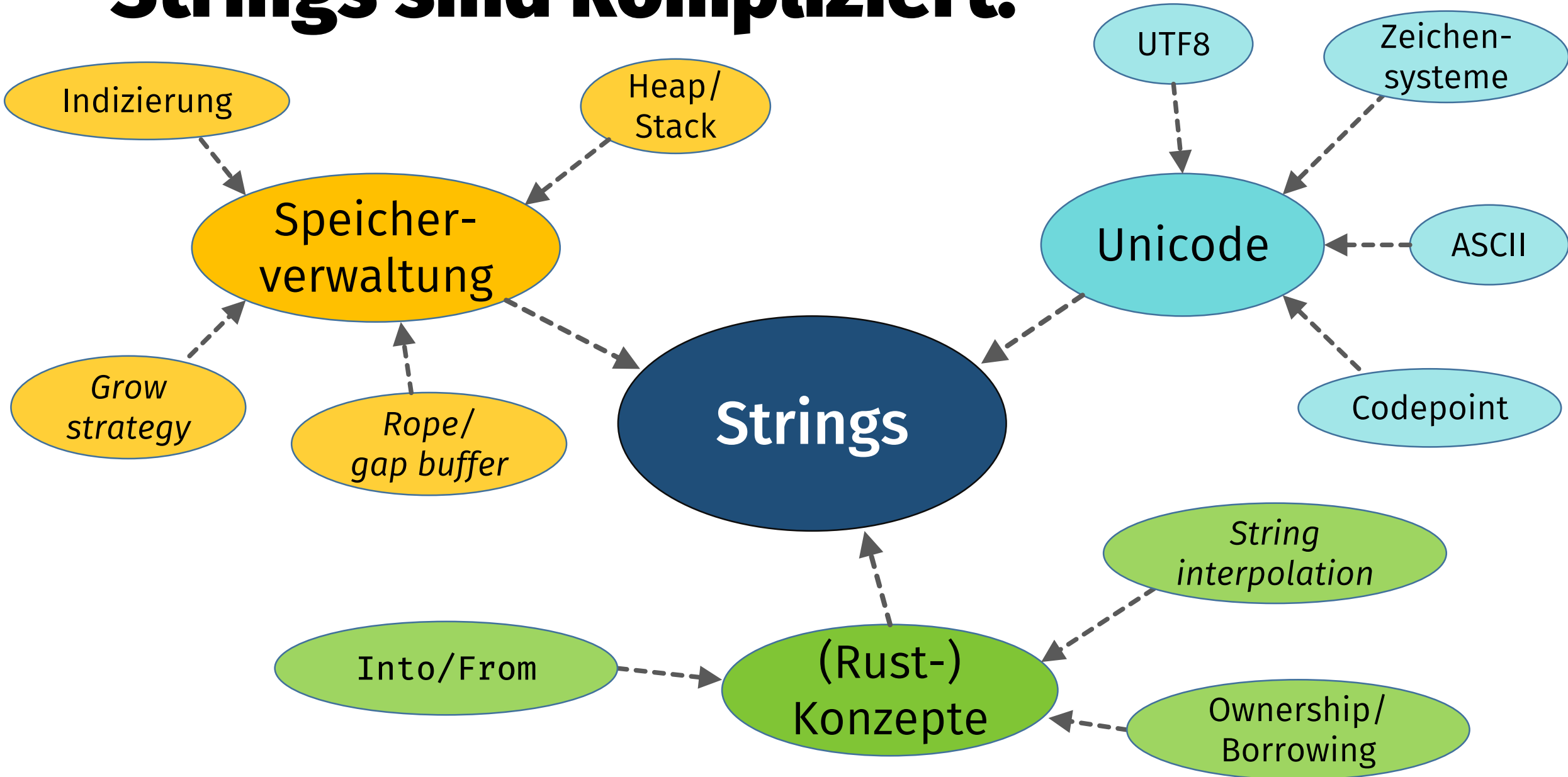


4.

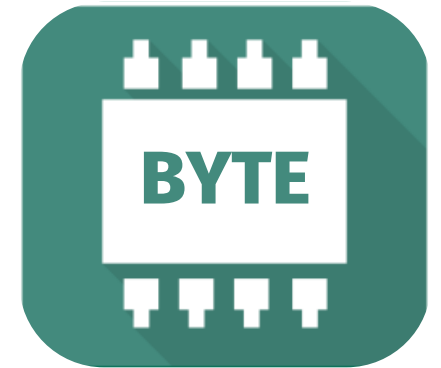
Strings

Strings sind kompliziert!



Die (stark vereinfachte) Geschichte der Zeichenkodierung

Dann hätte ich aber noch ein Bit über...



Scan-code	ASCII hex	ASCII dez	Zeichen	Scan-code	ASCII hex	ASCII dez	Zch.	Scan-code	ASCII hex	ASCII dez	Zch.	Scan-code	ASCII hex	ASCII dez	Zch.
00	0	0	NUL ^@	20	32	32	SP	40	64	64	@	0D	60	96	`
01	1	1	SOH ^A	21	33	33	!	1E	41	65	A	1E	61	97	a
02	2	2	STX ^B	22	34	34	"	30	42	66	B	30	62	98	b
03	3	3	ETX ^C	23	35	35	#	2E	43	67	C	2E	63	99	c
04	4	4	EOT ^D	24	36	36	\$	20	44	68	D	20	64	100	d
05	5	5	ENQ ^E	25	37	37	%	12	45	69	E	12	65	101	e
06	6	6	ACK ^F	26	38	38	&	21	46	70	F	21	66	102	f
07	7	7	BEL ^G	27	39	39	'	22	47	71	G	22	67	103	g
0E	08	8	BS ^H	28	40	40	(23	48	72	H	23	68	104	h
0F	09	9	TAB ^I	29	41	41)	17	49	73	I	17	69	105	i
	0A	10	LF ^J	2A	42	42	*	24	4A	74	J	24	6A	106	j
	0B	11	VT ^K	2B	43	43	+	25	4B	75	K	25	6B	107	k
	0C	12	FF ^L	2C	44	44	,	26	4C	76	L	26	6C	108	l
1C	0D	13	CR ^M	2D	45	45	-	32	4D	77	M	32	6D	109	m
	0E	14	SO ^N	2E	46	46	.	31	4E	78	N	31	6E	110	n
	0F	15	SI ^O	2F	47	47	/	18	4F	79	O	18	6F	111	o
	10	16	DLE ^P	30	48	48	0	19	50	80	P	19	70	112	p
	11	17	DC1 ^Q	31	49	49	1	10	51	81	Q	10	71	113	q
	12	18	DC2 ^R	32	50	50	2	13	52	82	R	13	72	114	r
	13	19	DC3 ^S	33	51	51	3	1F	53	83	S	1F	73	115	s
	14	20	DC4 ^T	34	52	52	4	14	54	84	T	14	74	116	t
	15	21	NAK ^U	35	53	53	5	16	55	85	U	16	75	117	u
	16	22	SYN ^V	36	54	54	6	0E	56	86	V	2F	76	118	v
	17	23	ETB ^W	37	55	55	7	08	57	87	W	11	77	119	w
	18	24	CAN ^X	38	56	56	8	09	58	88	X	2D	78	120	x
	19	25	EM ^Y	39	57	57	9	0A	59	89	Y	2C	79	121	y
	1A	26	SUB ^Z	3A	58	58	.	34	3A	58	.	1F	7A	122	z
01	1B	27	Esc ^[2B	43	43	,	33	2B	43	,	33	2B	43	,
	1C	28	FS ^\	2C	44	44	,	2C	44	68	D	2C	44	68	D
	1D	29	GS ^]												
	1E	30	RS ^^												
	1F	31	US ^^												

Hallo zusammen, ich bin ASCII und brauche 7 Bits!



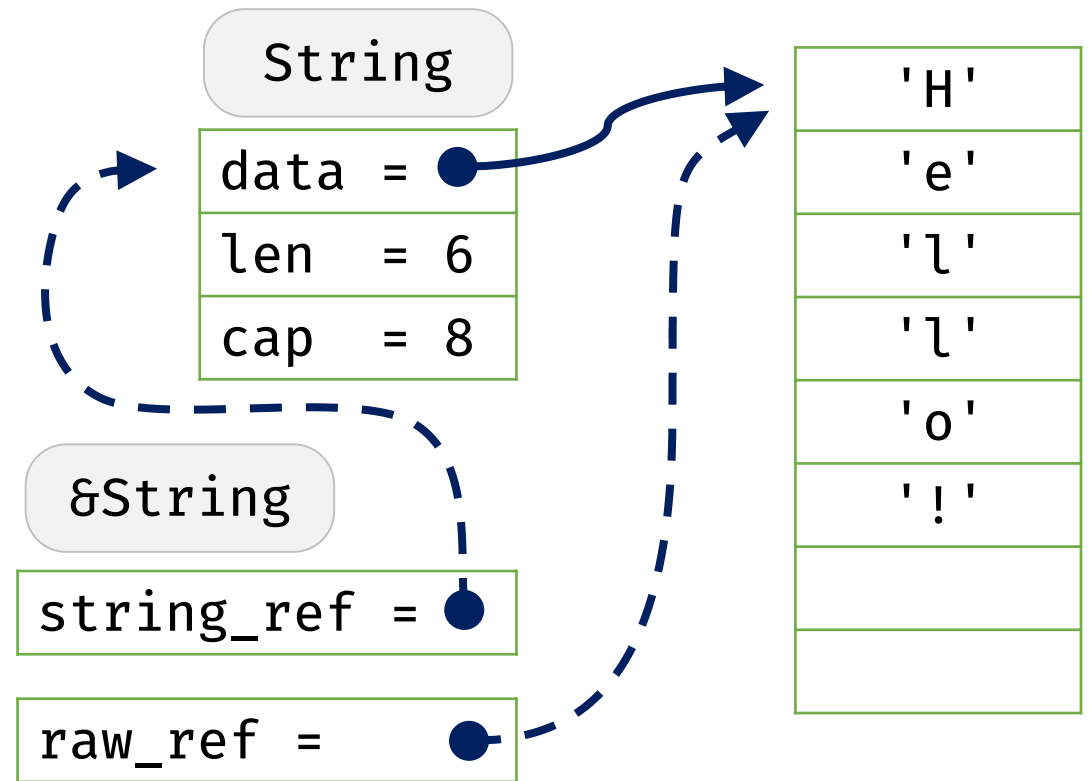
Unicode

- 32-Bit *Code Points*
 - Zeichen für quasi alle Kulturen ヲ ス ぽ
 - Nicht-Text Zeichen ☁️ ❤️ 🧑‍🎄
 - u.v.m.
- Verwaltet weiterhin Schreibrichtung, Rendering, ...
- UTF-8 Kodierung mit variabler Byte-Zahl
 - Erstes Bit ist *marker bit* („geht’s nach diesem byte noch weiter?“)
 - Kompatibel mit ASCII! (häufige Zeichen brauchen nur 1 Byte)
 - **Nachteil:** Indizierung schwierig! 😞



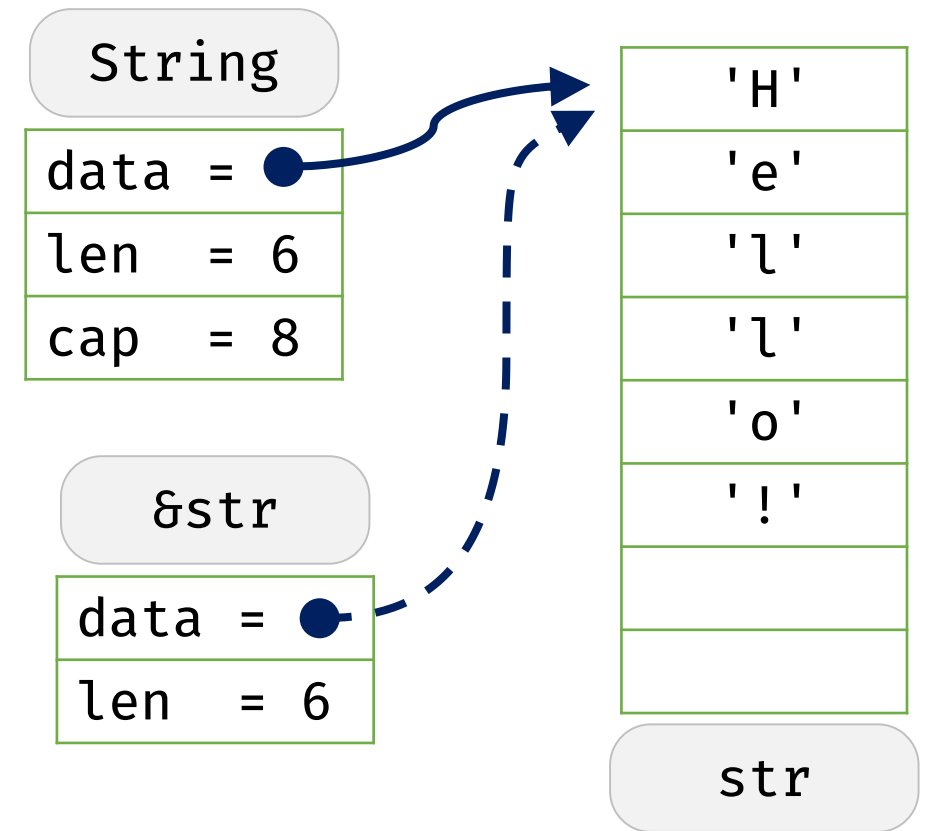
Strings in Rust

- Schon gesehen: **String**
 - Aus der Standardbibliothek
 - Garantiertes UTF-8!
 - *owned* string → besitzt Buffer
 - Wenn zerstört: gibt Buffer frei
 - *Nicht Copy* → *Move Semantics*
- Einen String ausborgen...
 - Eventuell **&String**? → Doppelte Referenz, setzt Heap-Buffer vorraus 😞
 - Referenz direkt auf den Stringbuffer? → Aber woher wissen wir die Länge?



Borrowed String

- Die Lösung: **&str** bzw. **str**
 - Ebenfalls UTF-8 Garantie
 - **str** ist ein sog. *unsized type*
 - Referenz auf *unsized types* enthält Länge
 - Später mehr zu dem Thema 😊



- Was ist eigentlich mit Stringliteralen? `let s = "wo lebe ich?";`
 - In Executable gespeichert (`.data` oder `.rodata`) → *nicht* Heap!
 - Lebt „immer“ (zur Laufzeit des Programms)

String Lifetime Quiz

```
let owned_a = "cheese".to_string(); // : String
let borrowed_a = owned_a.as_str(); // : &str

let borrowed_b = { // : &str
    let owned_b = "hi".to_string(); // : String
    owned_b.as_str() // error: `owned_b` does not live long enough
};

let borrowed_c = "ferris"; // what lifetime?
```

`&'static str`

- Spezielle Lifetime: **'static**
 - Lebt für „immer“

Konvertierung zwischen Stringarten

&str

Borrowed String. Ausgeborgt von:

- Literal ('static lifetime)
- **String**

```
// if possible: into()  
let a: String = "hi".into();  
  
fn takes_string(s: String) {}  
takes_string("hi".into());
```

```
// otherwise: to_string()  
let s = "hi".to_string()
```

Schnell!

Langsam!

```
let s = "hello".to_string();
```

```
// often just &  
let a: &str = &s;  
takes_string_slice(&s);
```

```
// if impossible, use `as_str()`  
let c = s.as_str();
```

String

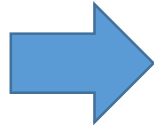
Owned String (verwaltet Stringbuffer)

Typische Fehler

~~Unicode ist komplex!~~
Sprache ist komplex!

- Indizierung [\[1\]](#) [\[2\]](#)

```
let s = "すa";  
let a = s[1]; // error!
```



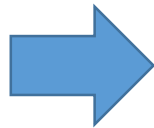
- UTF-8 Indizierung nicht trivial möglich!
- Unklar, was „gezählt“ werden soll: Bytes, Codepoints oder Grapheme Clusters

```
let a = s.chars().nth(1);
```

Auch nicht perfekt!

- Länge

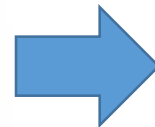
```
println!("{}", s.len());  
// output: 4
```



```
let a = s.chars().count();
```

- Kapitalisierung [\[3\]](#)

```
'x'.to_uppercase(); // not a char!  
'ß'.to_uppercase(); // ???
```



```
'x'.to_ascii_uppercase(); // : char  
"ß".to_uppercase(); // : String
```

Verallgemeinerung: Slices

- Slice `[T]` vergleichbar mit `str`
 - Im Speicher sind `[u8]` und `str` exakt gleich!
 - `[T]` ist ein *unsized type* (wie `str`)
 - `&[T]` enthält Pointer *und* Länge (wie `&str`)
- Owned Version von `[T]`?
 - Meistgenutzt: `Vec<T>` (später mehr dazu)
 - Viele Typen/Quellen von denen `[T]` ausgeborgt werden kann!
- Funktionsargumente: immer `&str/[T]` statt `&String/&Vec<T>` nutzen! (`&mut String/&mut Vec<T>` ist aber ok!)