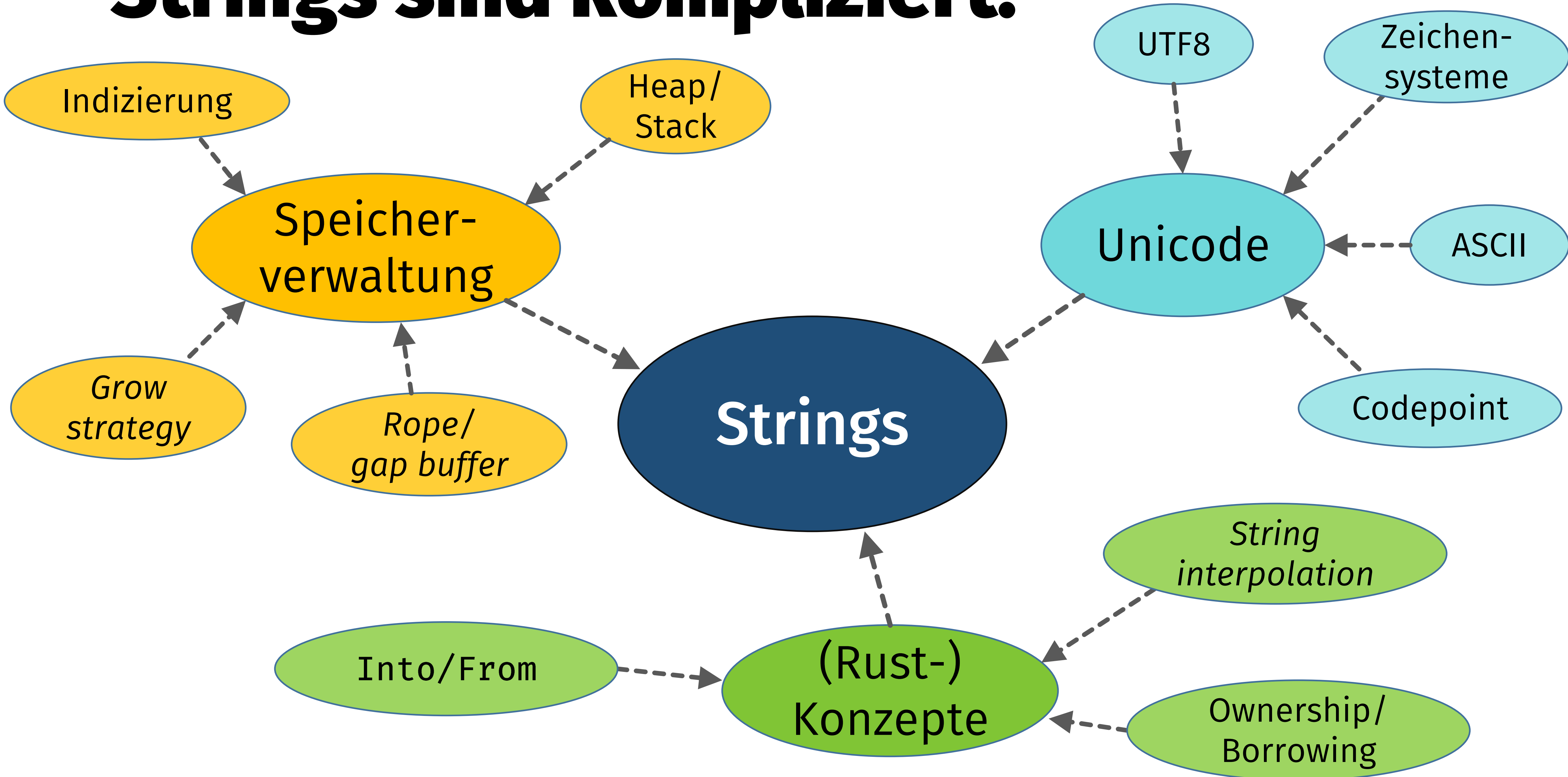
4.

Strings

Strings sind kompliziert!

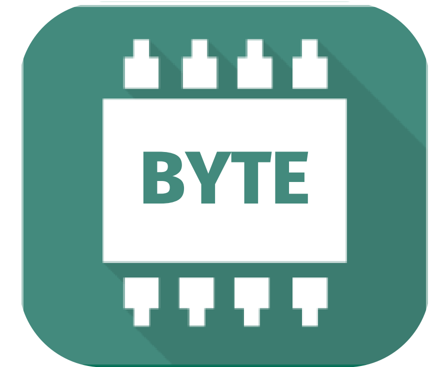


Die (stark vereinfachte) Geschichte der Zeichenkodierung

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

Hallo zusammen, ich bin *ASCII* und brauche 7 Bits!

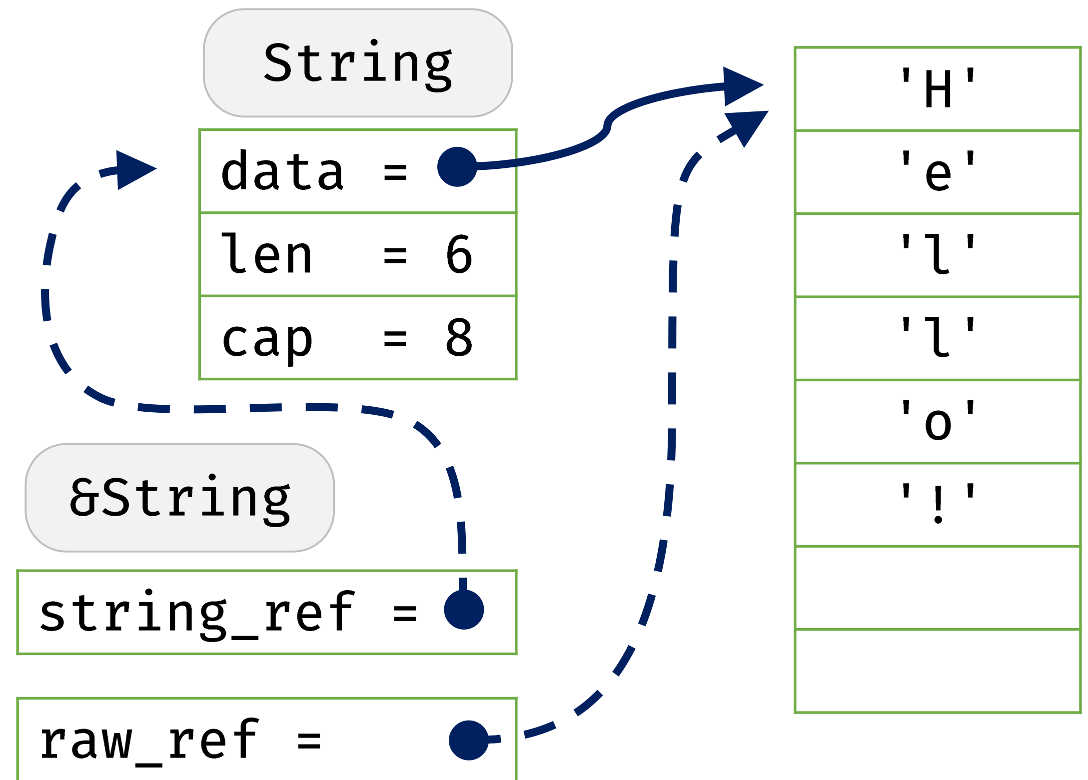
Dann h�tte ich aber noch ein Bit über...



Unicode

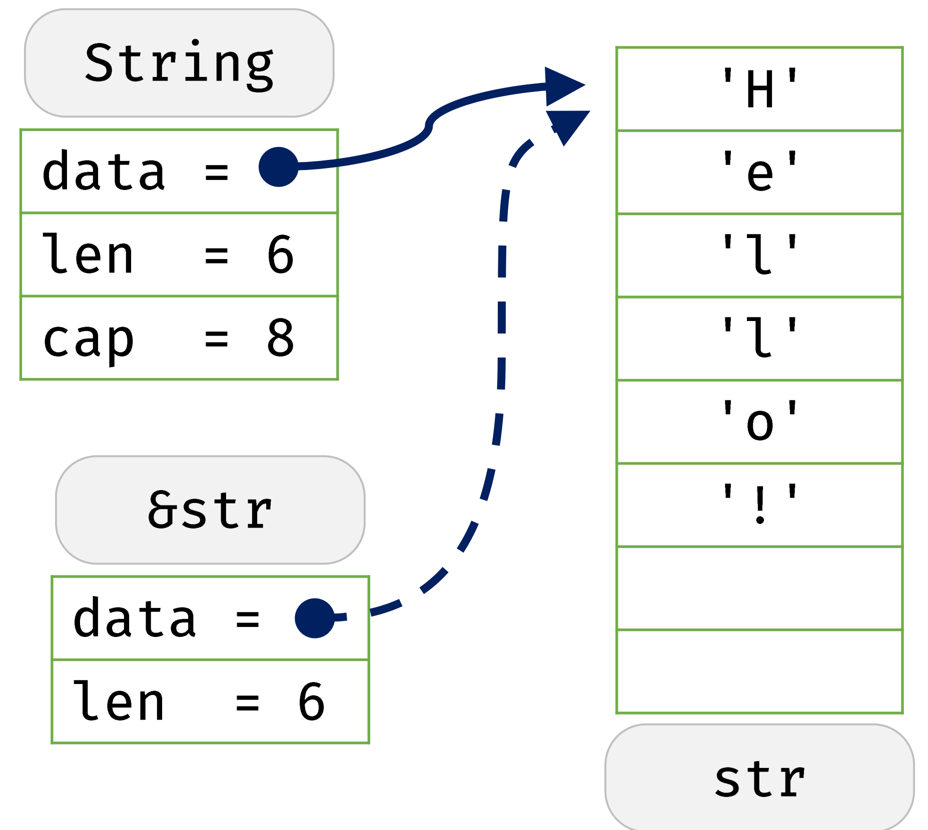
- 32-Bit *Code Points*
  - Zeichen für quasi alle Kulturen אע すほ
  - Nicht-Text Zeichen 🌈 ♥ ☃
  - u.v.m.
- Verwaltet weiterhin Schreibrichtung, Rendering, …
- UTF-8 Kodierung mit variabler Byte-Zahl
  - Erstes Bit ist *marker bit* („geht’s nach diesem byte noch weiter?“)
  - Kompatibel mit ASCII! (häufige 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 & (+ optional range)
let a: &str = &s;
let b: &str = &s[1..3];

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

`String`	Owned String (verwaltet Stringbuffer)

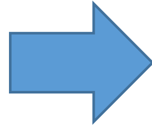
Typische Fehler

~~Unicode ist komplex!~~

Sprache ist komplex!

- Indizierung [1] [2]

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



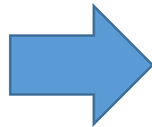
- UTF-8 Indizierung nicht trivial möglich!
- Unklar, was „gezählt“ werden soll: Bytes, *Codepoints* oder *Grapheme Clusters*

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

Auch nicht perfekt!

- Länge

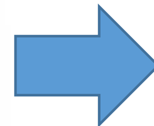
```
println!("{}", s.len());
// output: 4
```



```
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!)