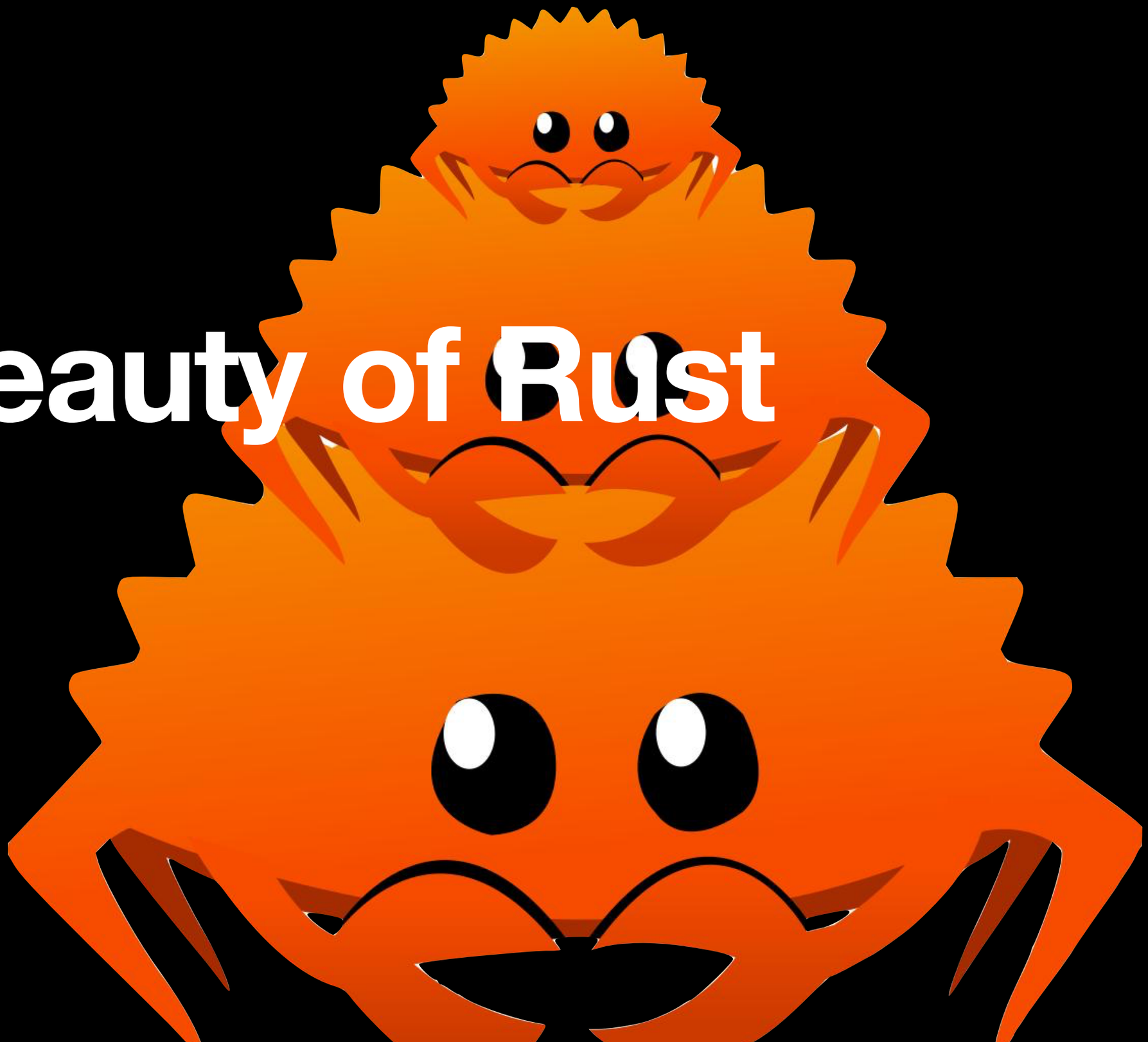
The Beauty of Rust

Collections



Peter Zdankin

What are collections?

- Common types of data structures that you are most likely to use
- Available in almost every language
- Not officially prescribed, but agreed upon that they are useful for all kinds of tasks
- Examples: Tree, HashMap, LinkedList, Array, ArrayList, String,

Why am I talking about collections today?

- They are incredibly useful
- You all should know the most important ones
- It’s pretty fun to use them, instead of primitive data types

Vector

Vec<T>

A List of T values

- A vector contains three variables
  - The address where the values are stored
  - How many elements there are currently stored
  - How many elements there can be stored in total

```rust
let mut v = Vec::new();
for i in 0..10{
    println!("{} {}", v.len(), v.capacity());
    v.push(i);
}

for number in v {
    println!("{number}");
}
```

Adding elements to Vec

- When adding elements, it is first checked if there is enough space
- If yes, no worries
- If no, we need to allocate more space
- Allocating more space invalidates the previously present memory
- What if we write to the Vector while someone else is reading it?

```rust
let mut evil_vec: Vec<String> = Vec::new();
evil_vec.push(String::from("abc"));

let x = &evil_vec[0];
for i in 0..10{
    evil_vec.push(String::from("bce"));
}
println!("{x}");
```

Getting elements from a vec

- There is the safe and the unsafe way
- Sounds bizarre, but yeah

```rust
fn safe_unsafe_get(){
    let mut v = Vec::new();
    for i in 0..10{
        v.push(i);
    }

    //safe
    match v.get(4) {
        Some(value) => println!("{value}"),
        None => println!("ERROR"),
    }
    match v.get(10) {
        Some(value) => println!("{value}"),
        None => println!("ERROR"),
    }

    //unsafe
    println!("{}", v[4]);
    println!("{}", v[10]);
}
```

```
4
ERROR
```

```
4
thread 'main' panicked at 'index out of bounds: the len is 10 but the index is 10', src/main.rs:43:20
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Hashmap

- Key Value Store
- Incredible performance, if your use case fits it
- Being able to use it, separates shitty developers from google powerhouses

Conceptually, Hashmap

TL;DR

- You map from one value to another, e.g. Int -> String
- You get the value associated with the key: map[key]
- You do stuff with the value
- You can iterate over the keys, or the values

Hashmap, concrete:

```rust
fn hashmap_example(){
    let mut map = HashMap::new();
    map.insert("ROT", 120);
    map.insert("BLAU", 140);

    if map.contains_key("ROT"){
        if let Some(points) = map.get_mut(&"ROT"){
            println!("{points}");
            *points += 10;
            println!("{points}");
        }
    }

    for (key, value) in map{
        println!("{key} {value}");
    }


}
```

Example

- I was a tutor in a programming class once
- I wanted to efficiently find plagiarism of students
- There is a bunch of code of a bunch of groups over a bunch of semesters

- How would you find duplicated code?

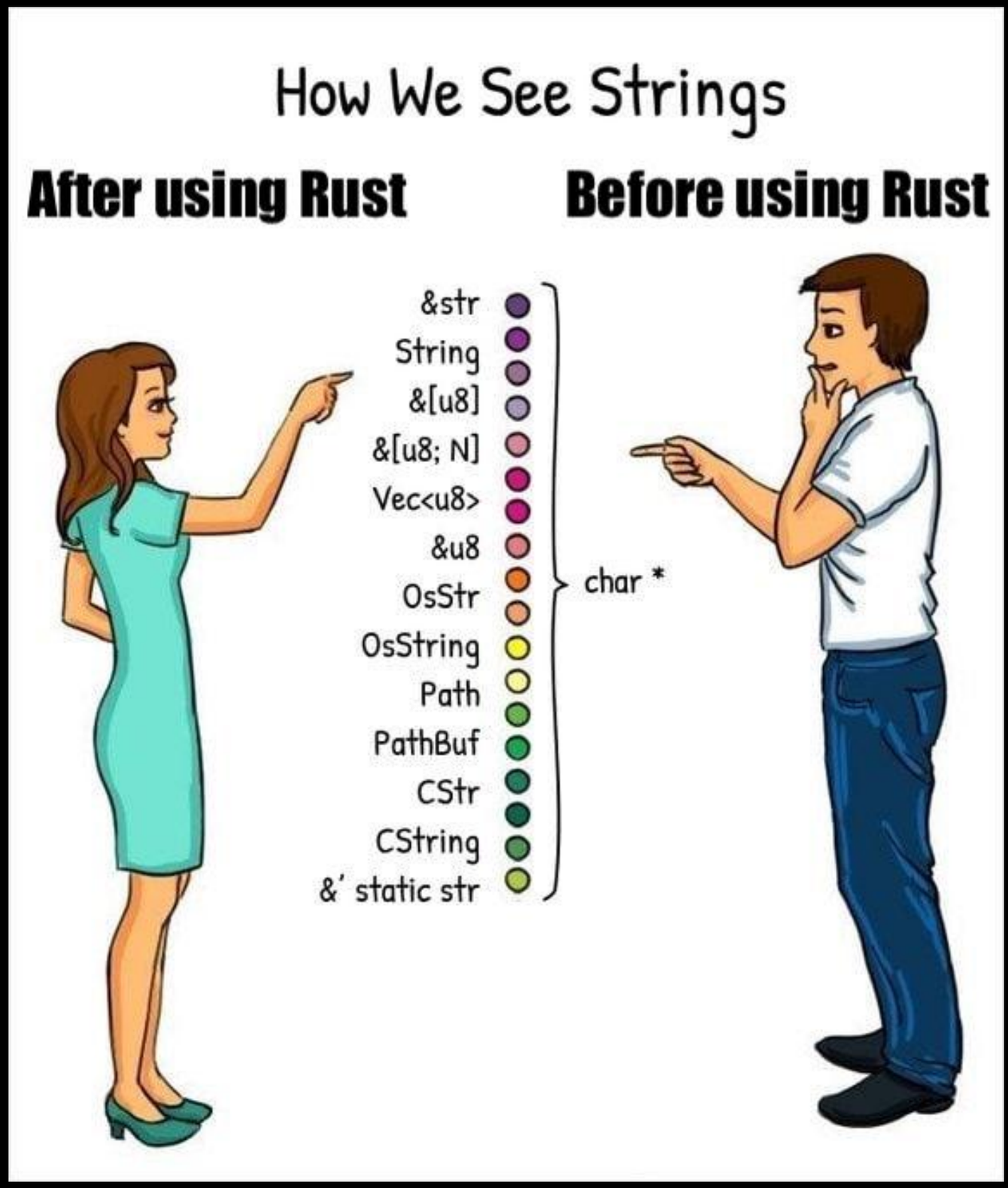
Solution

- Source code is pretty unique
- if let Some(points) = map.get_mut(&“ROT")
- Remove whitespaces
- ifletSome(points)=map.get_mut(&“ROT“)

- Use the line of code as the key, and the group that wrote it as a value, if the key is longer than a few characters
- map[ifletSome(points)=map.get_mut(&“ROT“)] = „Group 2, 2019“
- Iterate over all values that are bigger than 1

String

Strings are complicated



Used in countless different settings

- Chat messages
- Operating system paths
- Interaction with C ABIs
- Static storage
- Dynamic buffers
- Across languages

„Never ever assume you understand strings“

Me

ASCII

- 7 Bit code points (0..128)
- Basically, American keyboard + some extra characters
- Were used everywhere before Unicode became a thing

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	□

ASCII is great if you only use char*

- The C version of a string uses char*
- Fits perfectly, 7 bit

- Where does the string end? Fixed length? End of Word symbol?
- What if you store an ö?
- What if you store emojis?
- What if some API requires the usage of char* but you use Ö?

Unicode

- A universal lexicon of all symbols that humanity has ever used with enough space for countless new emojis

- An unicode symbol can be written in UTF-8, UTF-16, or UTF-32
- Literally anything uses UTF-8

UTF-8

- Uses 1, 2, 3 or 4 bytes to represent an unicode code point
- 1 Byte: 0xxxxxxx : 7 x’s
- 2 Byte: 110xxxxx 10xxxxxx : 11x’s
- 3 Byte: 1110xxxx 10xxxxxx 10xxxxxx : 16 x’s
- 4 Byte: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx : 21 x’s

- You take the binary representation of your code point and fit it from right to left in the x’s
- Leftover x’s are padded with 0

How many code points?

- Imagine you have a 200 byte UTF-8 encoded text
- How many code points are there?
  - 50 - 200
- You must traverse front to back to count how many code points there are
- Therefore, working with text is difficult!
- If you just access text[20] you could corrupt text

Strings in Rust

- In Rust you will encounter two main types of strings
- Immutable string slices

```
let data = "initial contents";
```

- Heap-managed strings

```
let s = String::from("initial contents");
```

Modifying strings

Only the ones you can modify

```
let mut str1 = String::from("Hello, ");
str1.push_str("World");
println!("{str1}");
str1.push('!');
println!("{str1}");
```

Getting fancy with strings

I want to do some nice formatting

```
let i = 5;
let nice = format!("You have {i} points");
println!("{nice}");
```

Getting something from a String?

Besides creepy stares

```rust
let x = nice[3];
```

```
error[E0277]: the type `String` cannot be indexed by `{integer}`
  --> src/main.rs:79:13
   |
79 |     let x = nice[3];
   |             ^^^^^^^ `String` cannot be indexed by `{integer}`
   |
   = help: the trait `Index<{integer}>` is not implemented for `String`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

You can use slices

But beware

```rust
let i = 5;
let nice = format!("Yöu have {i} points");
println!("{nice}");

let x = &nice[0..2];
println!("{x}");
```

```
You have 5 points
thread 'main' panicked at 'byte index 2 is not a char boundary; it is inside 'ö' (bytes 1..3) of `Yöu have 5 points`', src/main.rs:7
9:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Safer option

```
let x = nice.get(0..3);
if let Some(slice) = nice.get(0..2){
    println!("{slice}");
}
if let Some(slice) = nice.get(0..3){
    println!("{slice}");
}
```

Exercises

- Create a Vec<i32>, that stores the numbers from 1 to 1000
- Remove all odd numbers from the Vec
- Create a Vec that contains 500 distinct names
- Create a HashMap that maps the names to the first vec
- Double each value in this HashMap
- Print out all keys and values in this HashMap

- Create a String that contains all keys of the HashMap