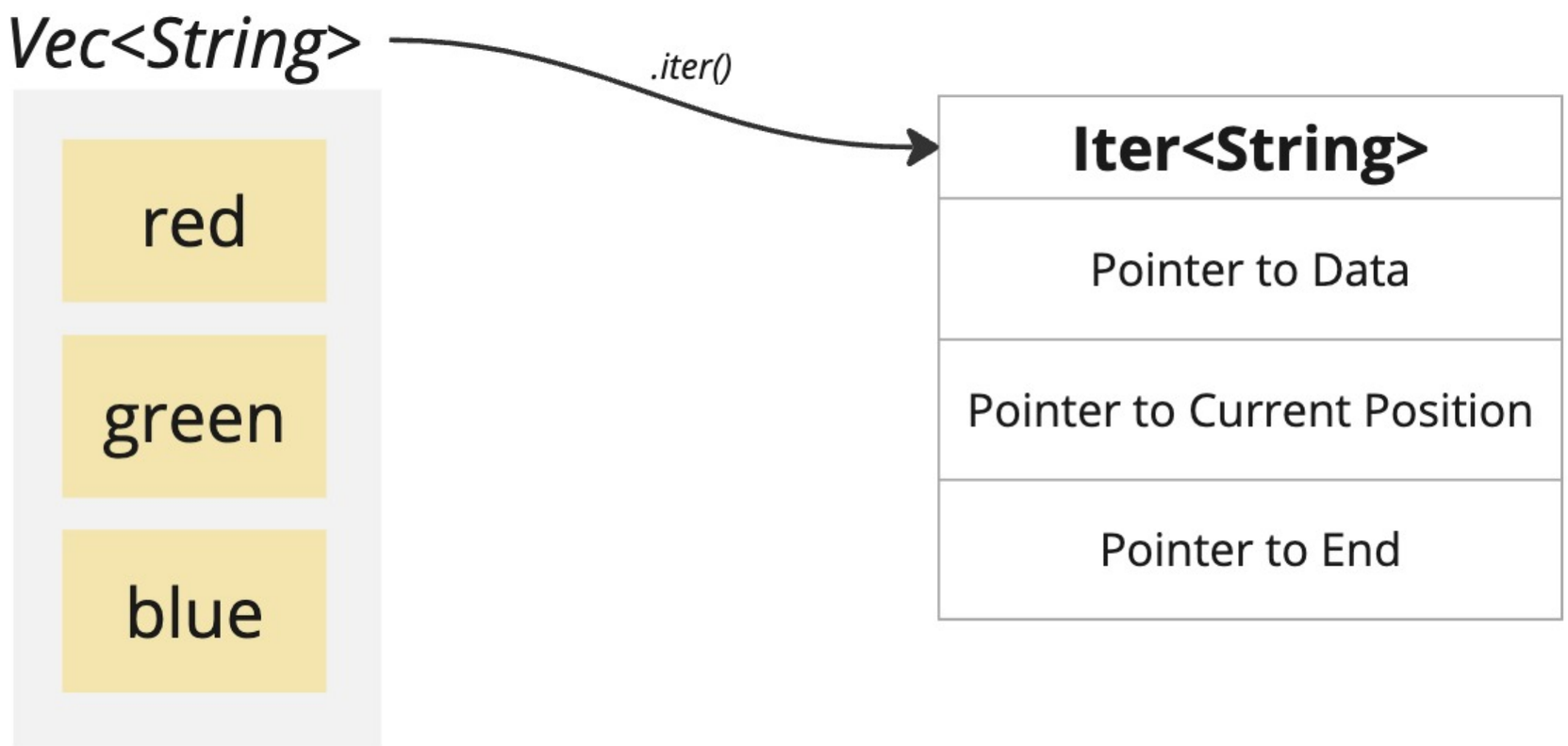
Iterators

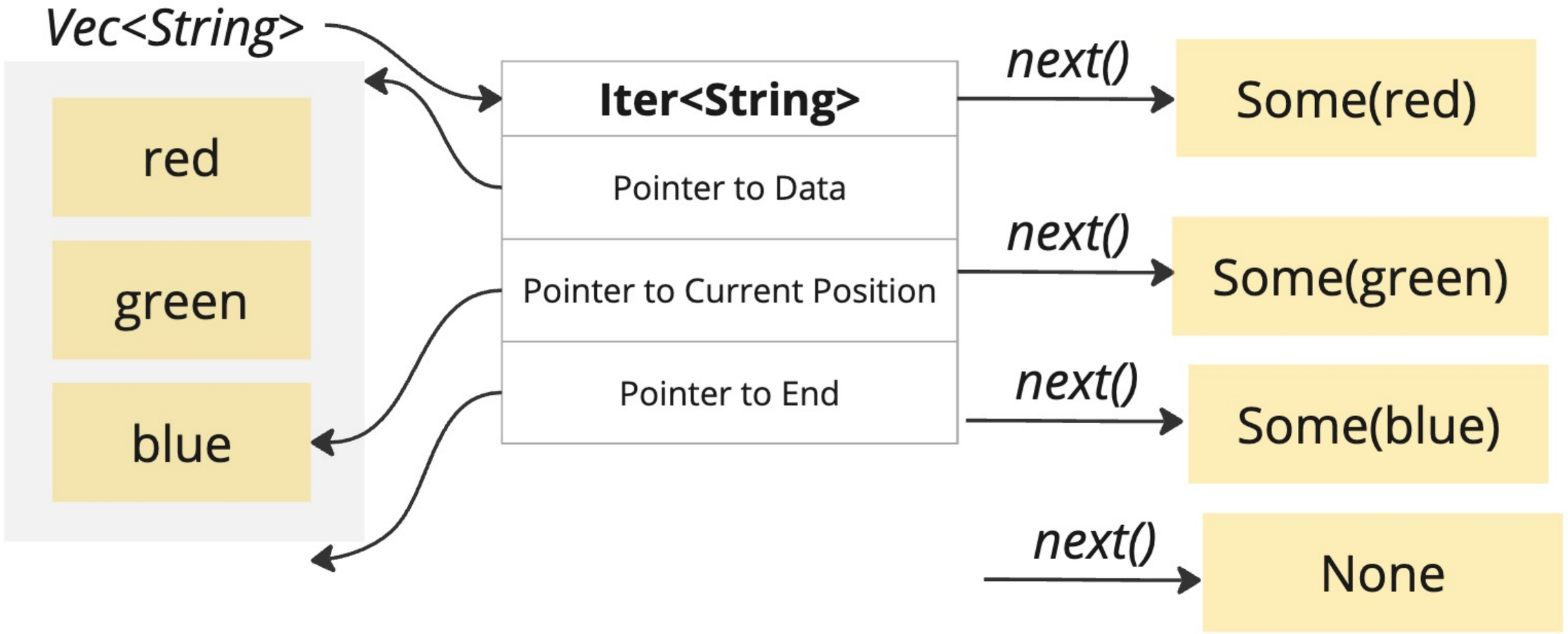
Used to iterate over any kind of data structure

We've already been using them - they are used behind the scenes when you write a for loop

Follow all the same rules of ownership, borrowing, lifetimes

Use the Option enum


Vec<String>
red
green
blue
.iter()
Iter<String>
Pointer to Data
Pointer to Current Position
Pointer to End


Vec<String>
red
green
blue
Iter<String>
Pointer to Data
Pointer to Current Position
Pointer to End
next()
Some(red)
next()
Some(green)
next()
Some(blue)
next()
None

Is 'color' a...

String value?

read only reference to a String?

Mutable Reference to a String?

...And why would we care?

```
let colors = vec![
    String::from("red"),
    String::from("green"),
    String::from("blue"),
];

let mut colors_iter = colors.iter();

if let Some(color) = colors_iter.next() {

}
```

If 'color' is a ref to a String...

We can't change the String using this read-only ref

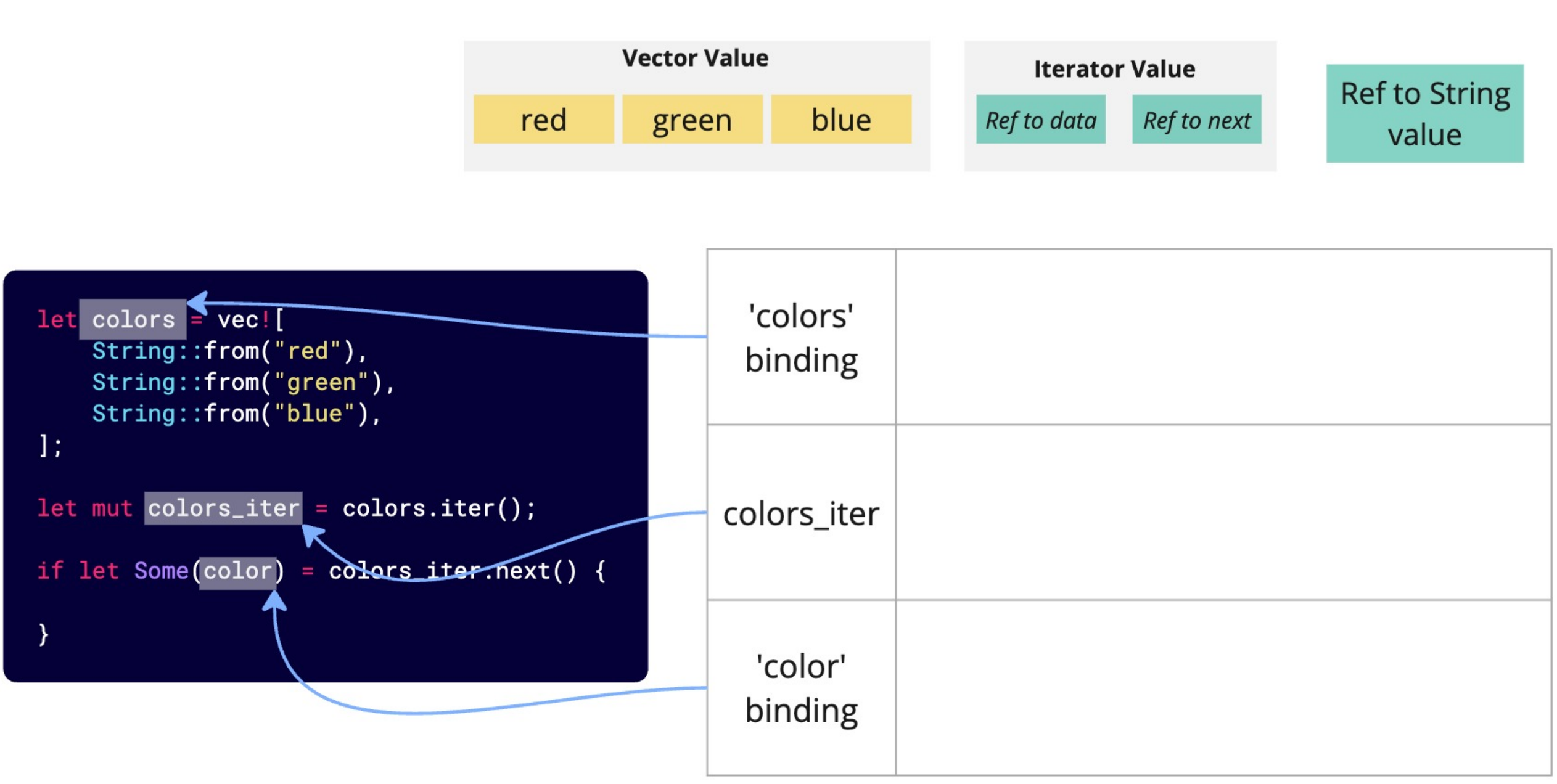
```
let colors = vec![
    String::from("red"),
    String::from("green"),
    String::from("blue"),
];

let mut colors_iter = colors.iter();

if let Some(color) = colors_iter.next() {
    // Error!
    colors.truncate(1);
}
```

If 'color' is a String value...

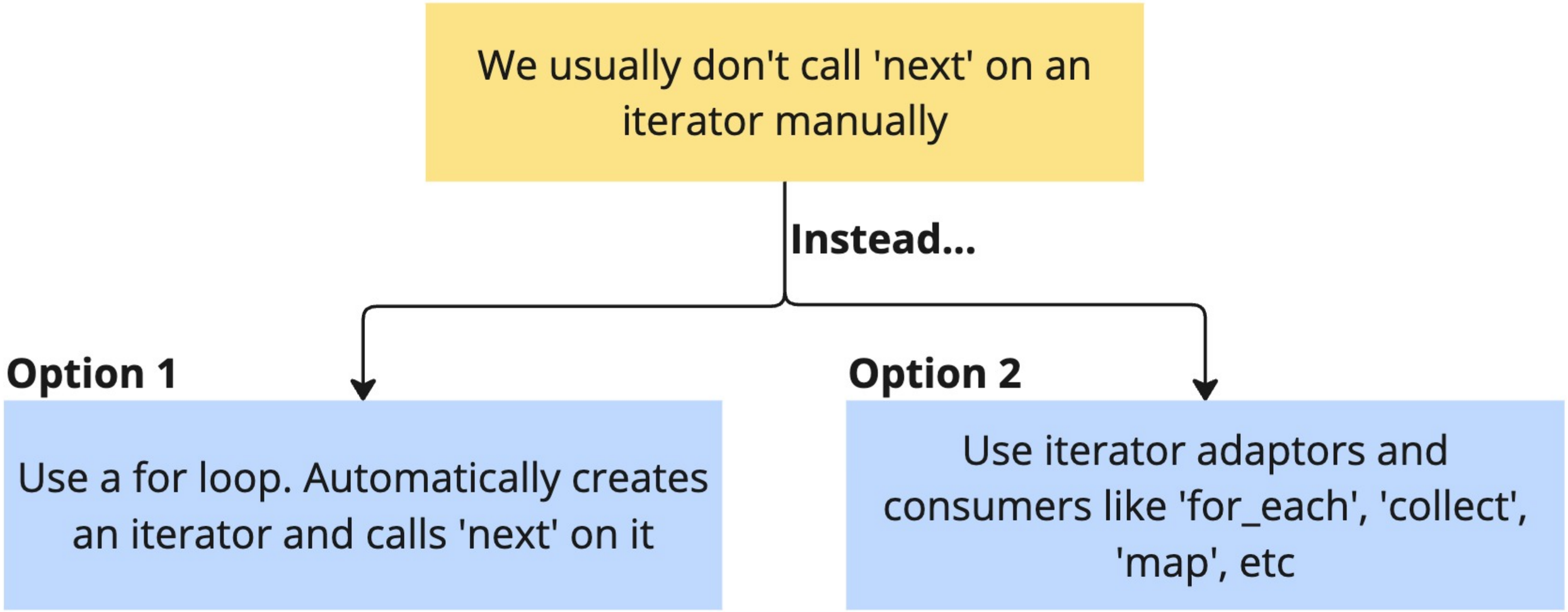
We took ownership, and 'colors' doesn't contain that String anymore

```
let colors = vec![
    String::from("red"),
    String::from("green"),
    String::from("blue"),
];

let mut colors_iter = colors.iter();

if let Some(color) = colors_iter.next() {
    // Error!
    colors.truncate(1);
}
```

1	Every value is 'owned' by a single variable, argument, struct, vector, etc at a time	**Ownership**
2	Reassigning the value to a variable, passing it to a function, putting it into a vector, etc, *moves* the value. The old owner can't be used to access the value anymore!	
3	You can create many read-only references to a value that exist at the same time. These refs can all exist at the same time	**Borrowing**
4	You can't move a value while a ref to the value exists	
5	You can make a writeable (mutable) reference to a value *only if* there are no read-only references currently in use. One mutable ref to a value can exist at a time	
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists	
7	Some types of values are *copied* instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	
8	When an owner goes out of scope, the value owned by it is *dropped* (cleaned up in memory)	**Lifetimes**
9	There can't be references to a value when its owner goes out of scope	
10	References to a value can't outlive the value they refer to	
11	**These rules will dramatically change how you write code (compared to other languages)**	
12	**When in doubt, remember that Rust wants to minimize unexpected updates to data**	


Vector Value
red
green
blue
Iterator Value
Ref to data
Ref to next
Ref to String value
let colors = vec![
String::from("red"),
String::from("green"),
String::from("blue"),
];
let mut colors_iter = colors.iter();
if let Some(color) = colors_iter.next() {
}
'colors' binding
colors_iter
'color' binding

iter() →	The iterator will give you a **read-only reference** to each element
into_iter() →	The iterator give you **ownership** of each element
iter_mut() →	The iterator will give you a **mutable reference** to each element


We usually don't call 'next' on an iterator manually
Instead...
Option 1
Use a for loop. Automatically creates an iterator and calls 'next' on it
Option 2
Use iterator adaptors and consumers like 'for_each', 'collect', 'map', etc

```rust
fn print_elements(elements: &Vec<String>) {
    for element in elements {
        println!("{}", element);
    }
}
```

'for' loops will...

Automatically create an iterator for the vector

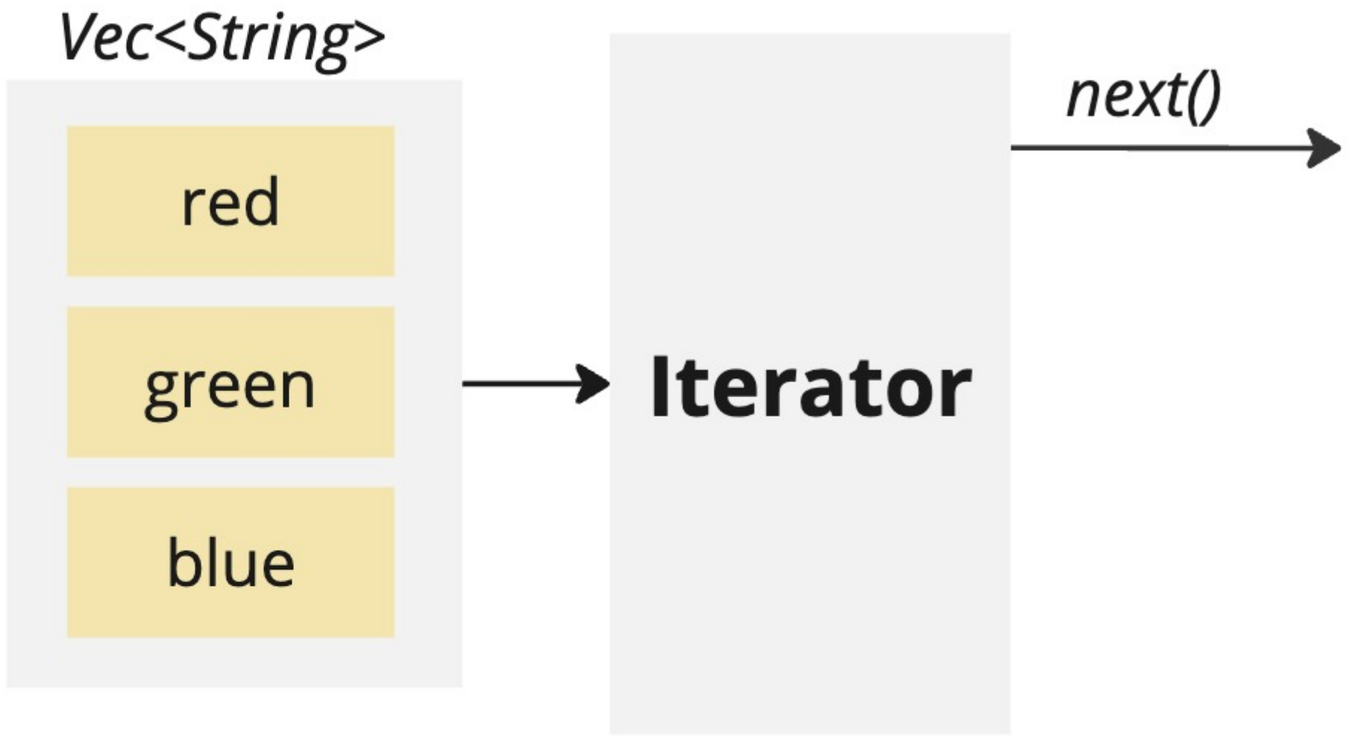
Call 'next' on the iterator and unwrap the Option that comes back

Break once 'next' returns a None

Iterators are '*lazy*'. Nothing happens until...

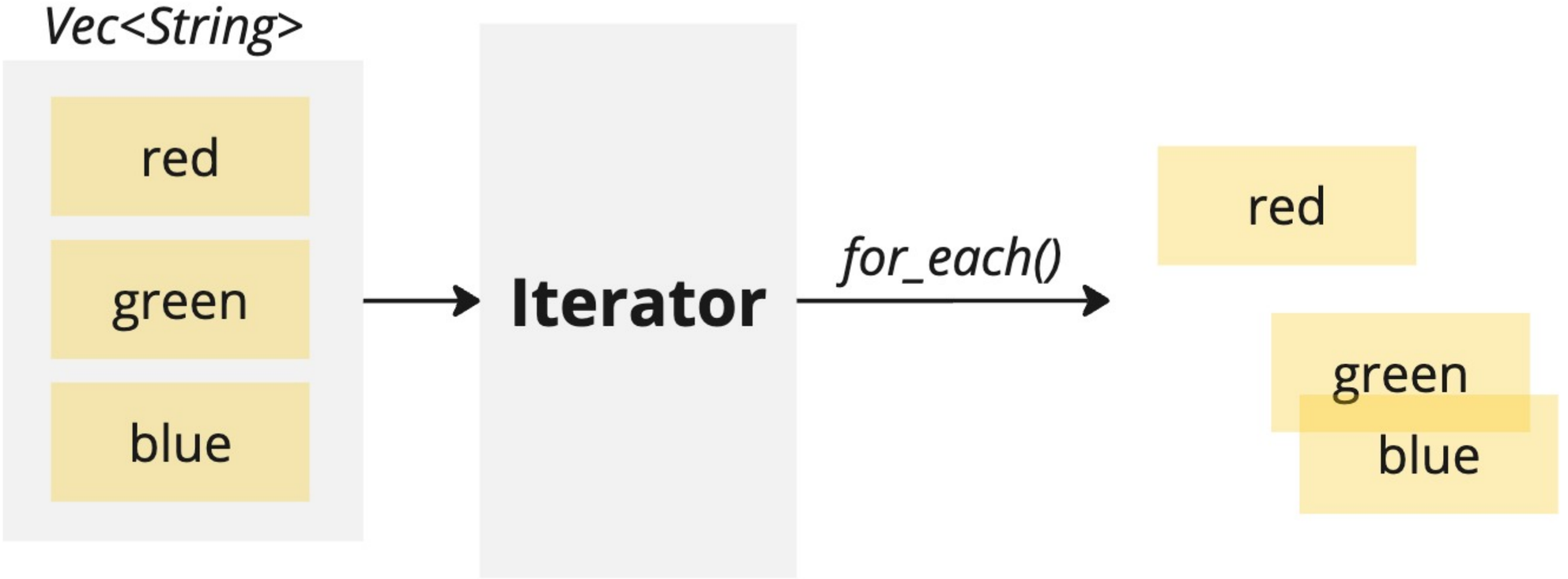
A) You call 'next'

B) You use a function that calls 'next' automatically



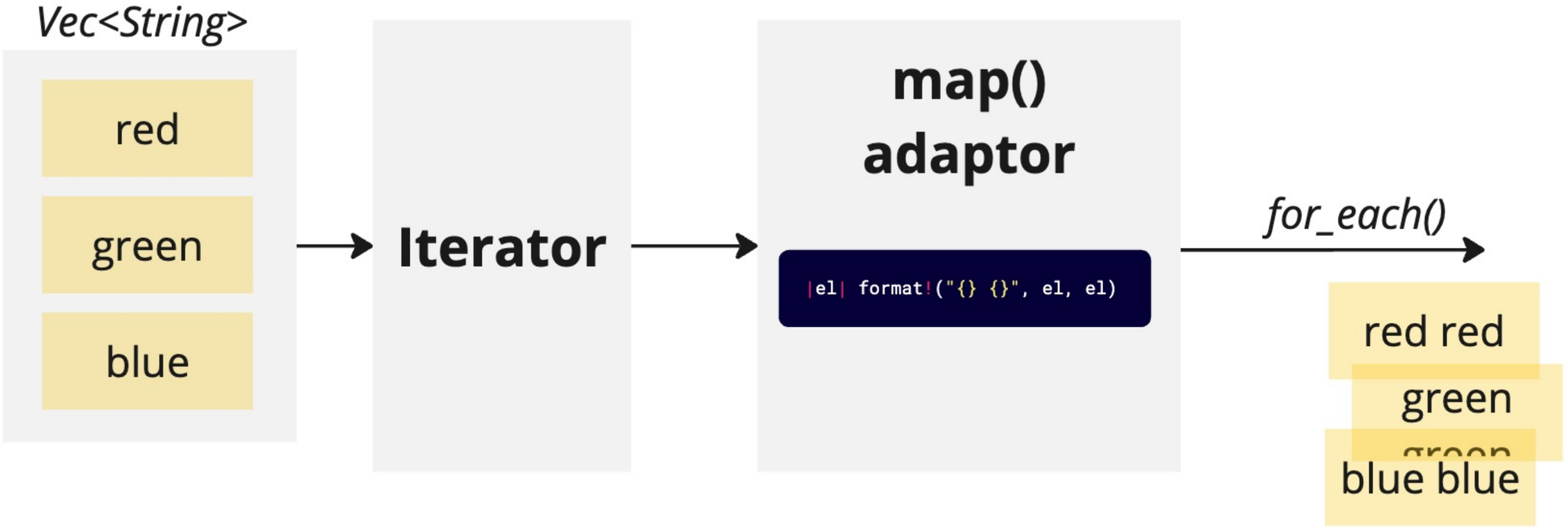
for_each() is an iterator **consumer**

It will repeatedly call 'next()' on the iterator until it gets 'None'



map() is an iterator **adaptor**

Adaptors create a step in a processing pipeline, but don't actually cause any iteration

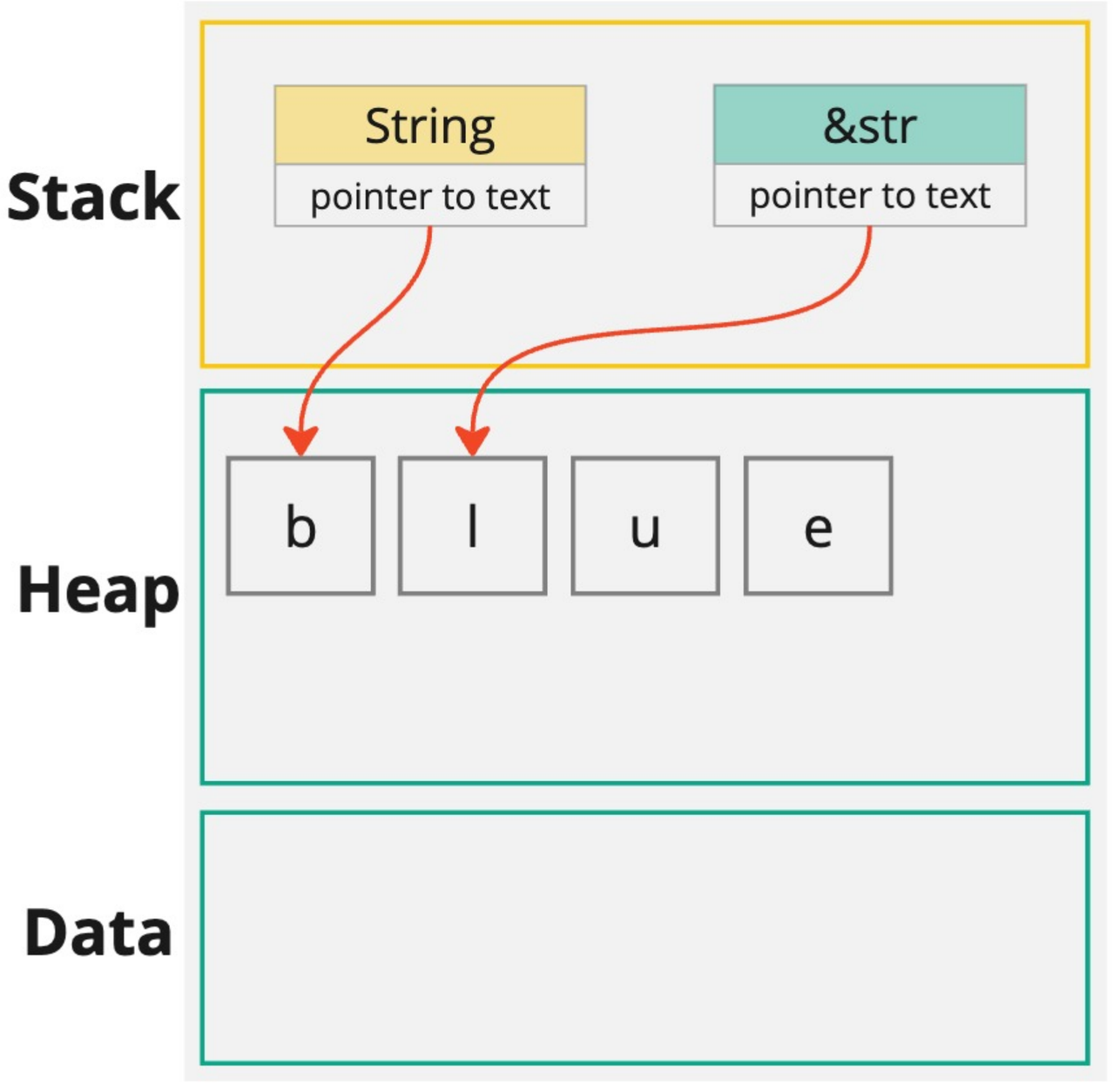


```rust
fn print_elements(elements: &Vec<String>) {

}
```

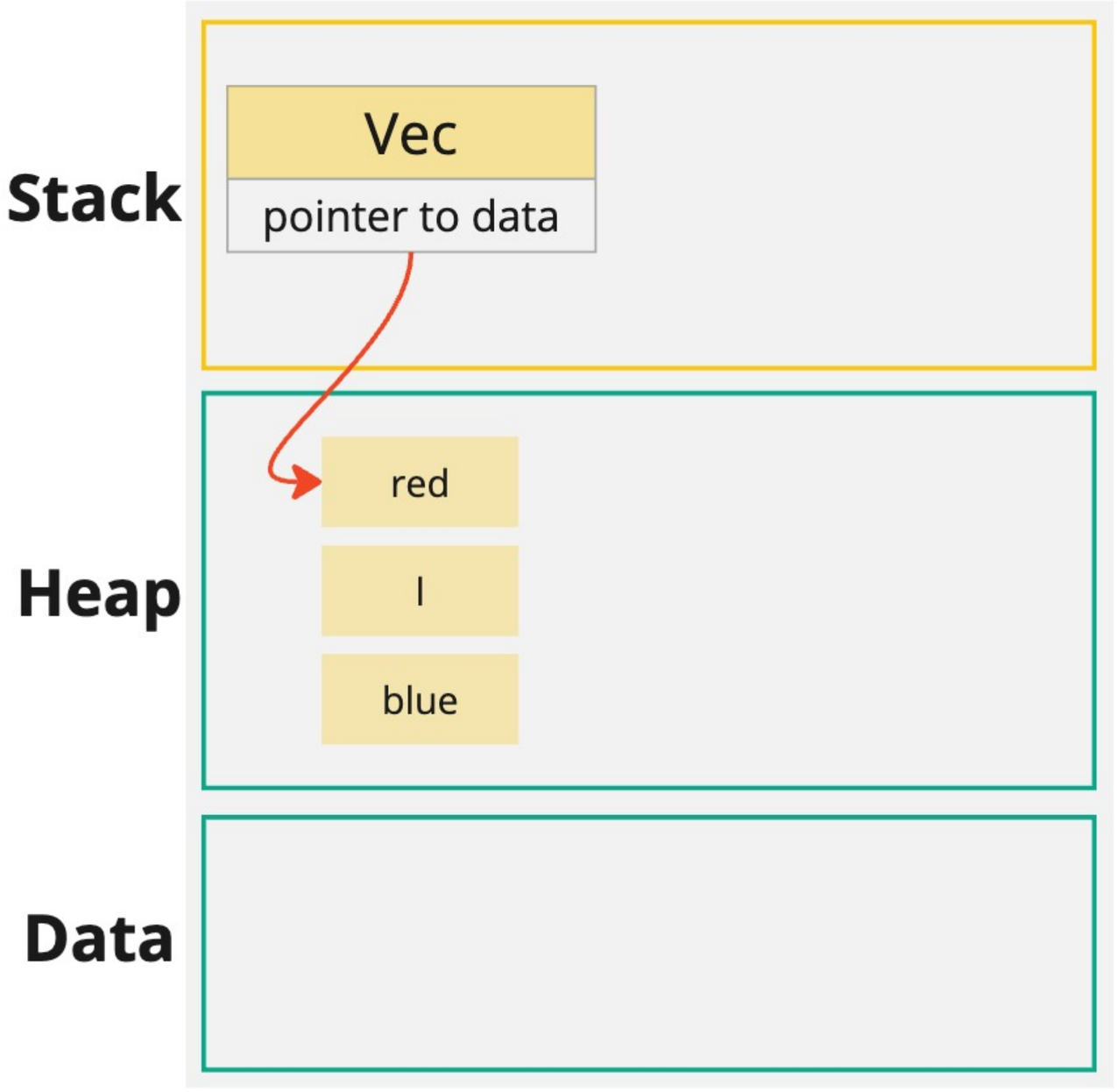
You might be getting a warning from this

Something about using `&[_]`

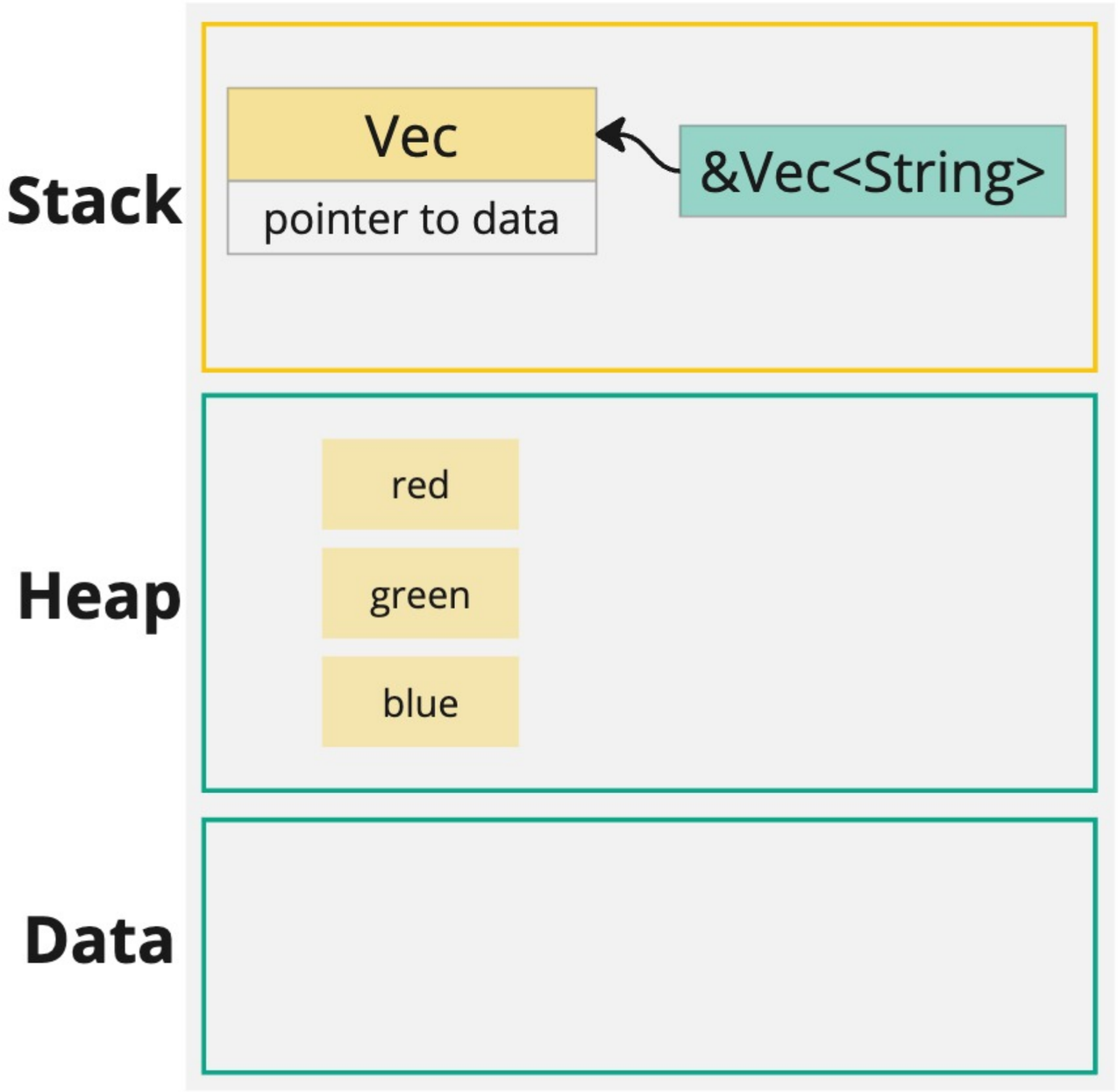
Reminder on &str




```rust
fn get_color_slice() -> &str {
    let color = String::from("blue");
    let color_ref = &color[0..4];

    color_ref
}
```

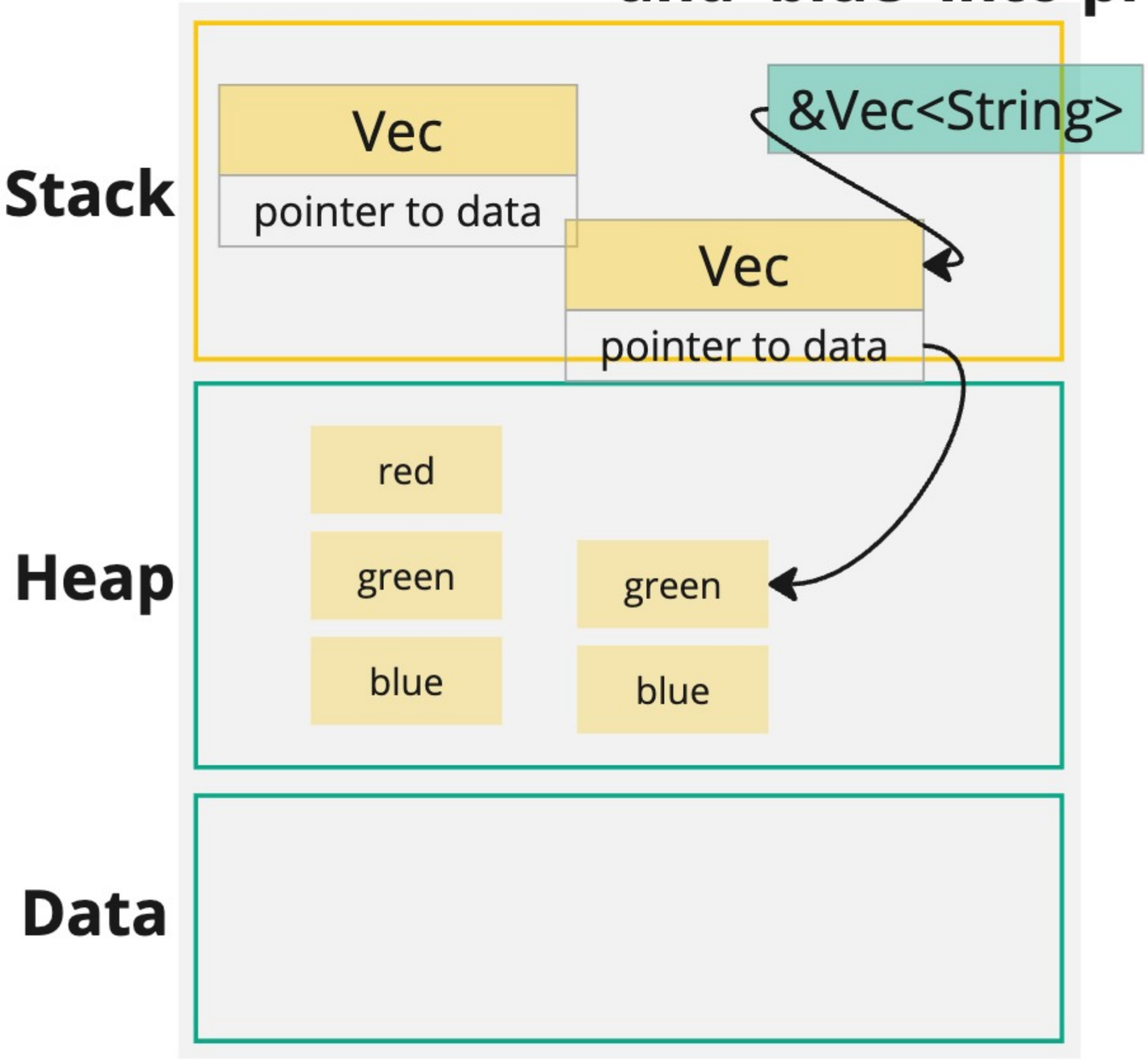



```rust
fn print_elements(elements: &Vec<String>) {
    /* code... */
}

fn main() {
    let colors = vec![
        String::from("red"),
        String::from("green"),
        String::from("blue"),
    ];


    print_elements(&colors);
}
```




```rust
fn print_elements(elements: &Vec<String>) {
    /* code... */
}

fn main() {
    let colors = vec![
        String::from("red"),
        String::from("green"),
        String::from("blue"),
    ];


    print_elements(&colors);
}
```

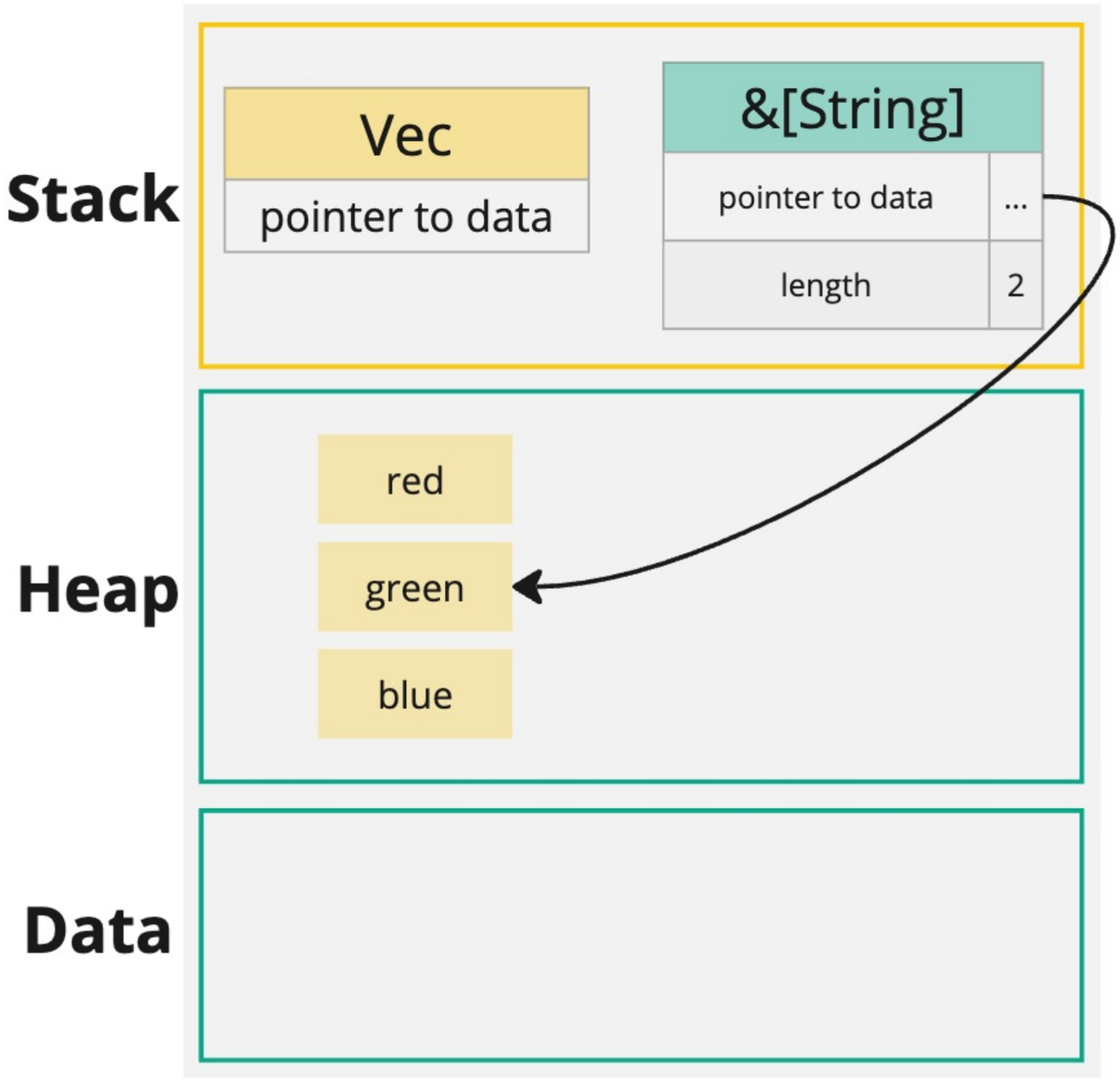
What if we only wanted to pass 'green' and 'blue' into print_elements()?




```rust
fn print_elements(elements: &Vec<String>) {
    /* code... */
}

fn main() {
    let colors = vec![
        String::from("red"),
        String::from("green"),
        String::from("blue"),
    ];


    print_elements(&colors);
}
```

Solution: Vector Slice

Lets us point at a portion of data owned by something else




```
fn print_elements(elements: &[String]) {
    /* code... */
}

fn main() {
    let colors = vec![
        String::from("red"),
        String::from("green"),
        String::from("blue"),
    ];


    print_elements(&colors[1..3]);
}
```

The warning is suggesting we do **this**

instead of **this**

```
fn print_elements(elements: &Vec<String>) {
    /* code... */
}
```

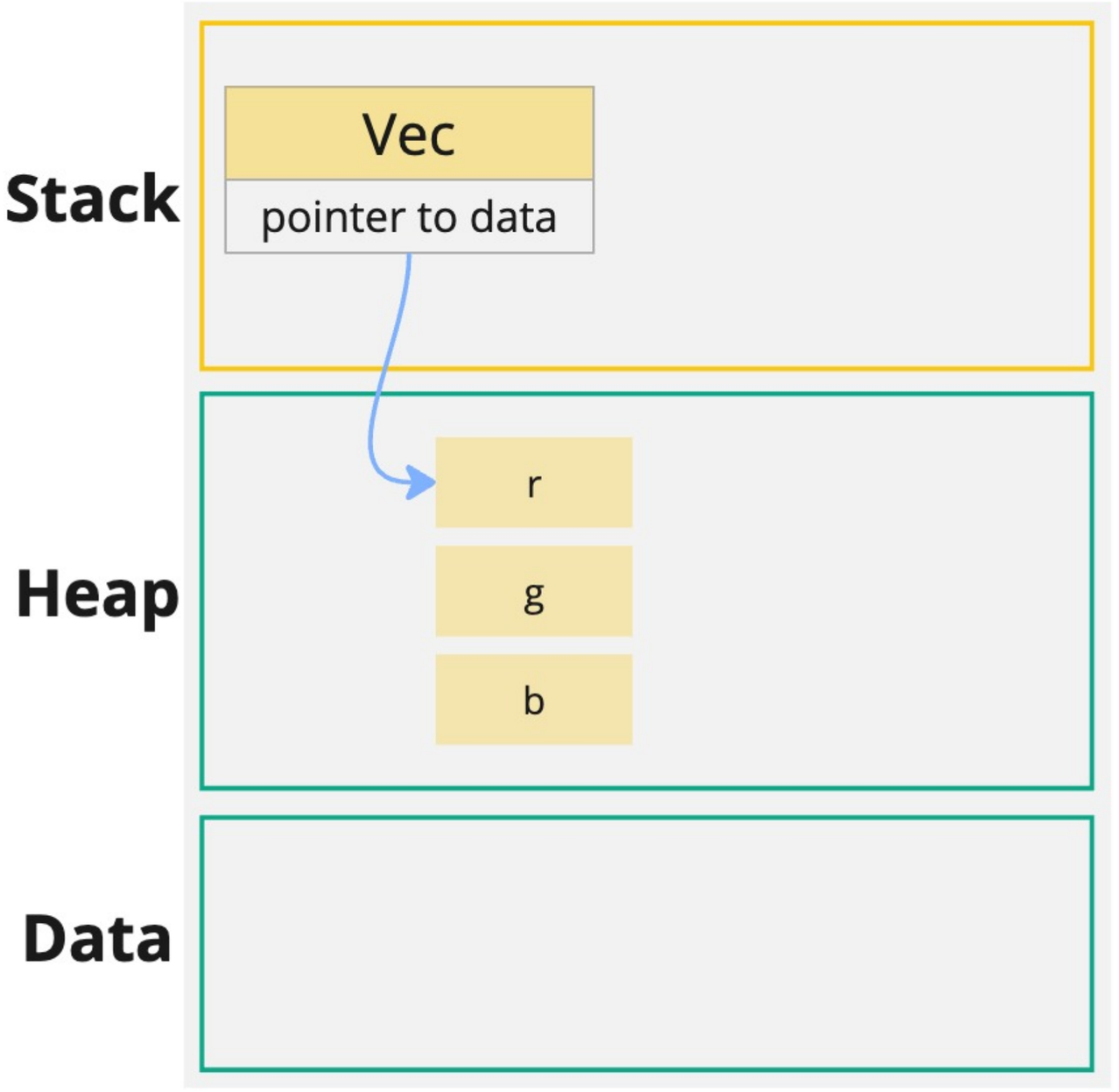
```
fn print_elements(elements: &[String]) {
    /* code... */
}
```

Why?

```rust
fn print_elements(elements: &[String]) {
    /* code... */
}
```

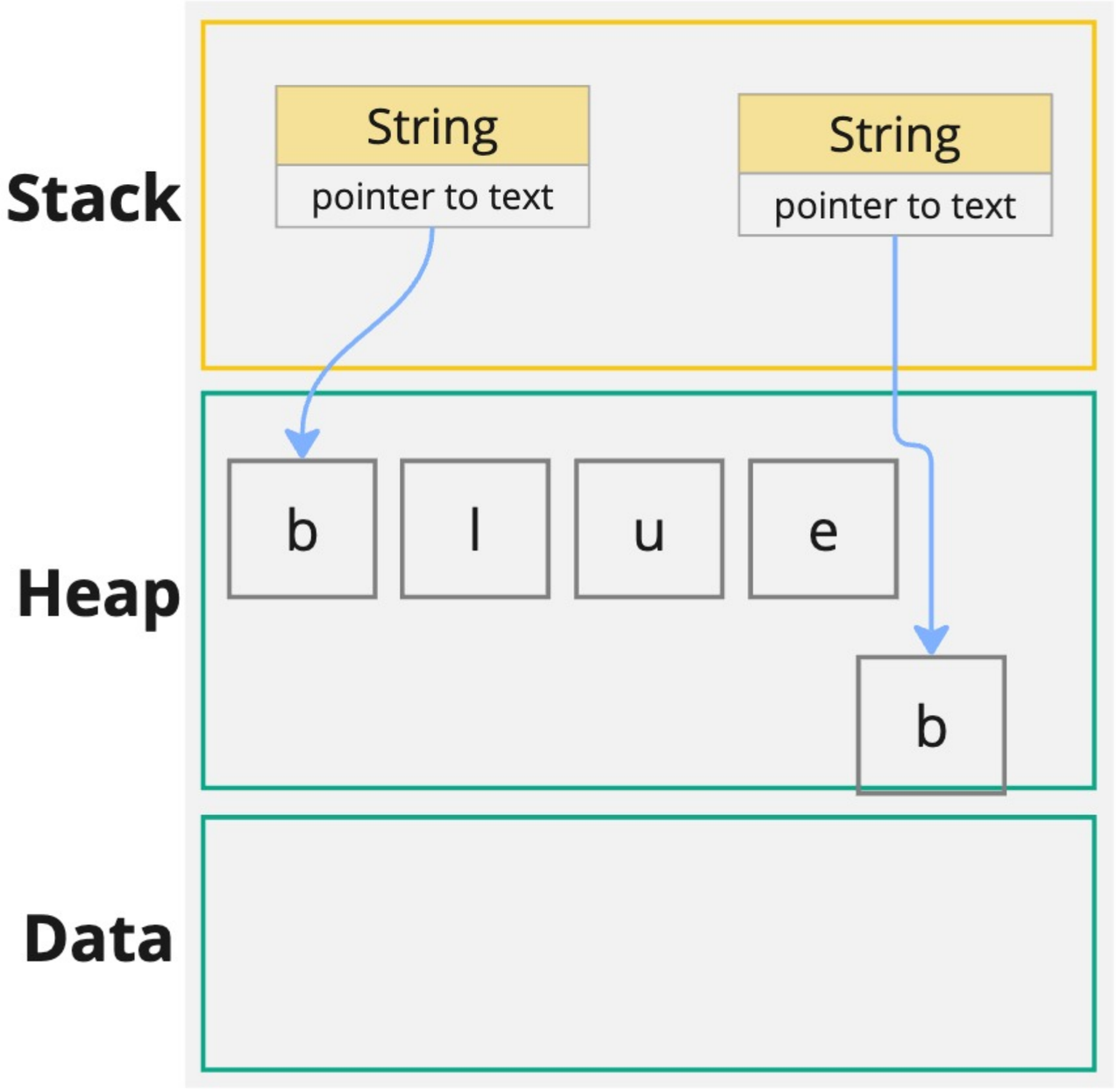
Can be called with ***either***
&Vec<String>
or
&[String]

Single function that can work with either a full vector or just a portion of a vector



shorten_strings() should modify the strings in the vector

We don't want to create a new vector

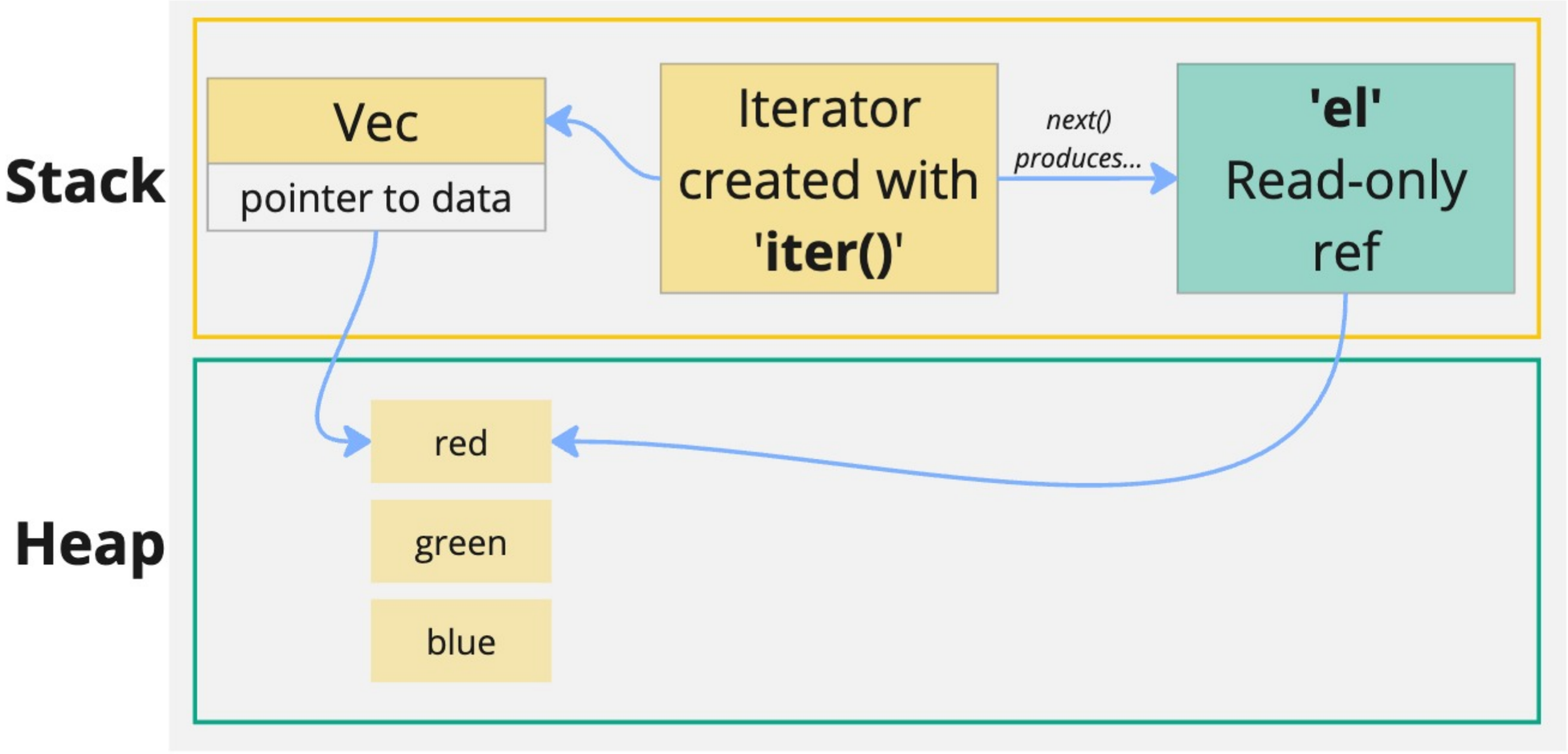


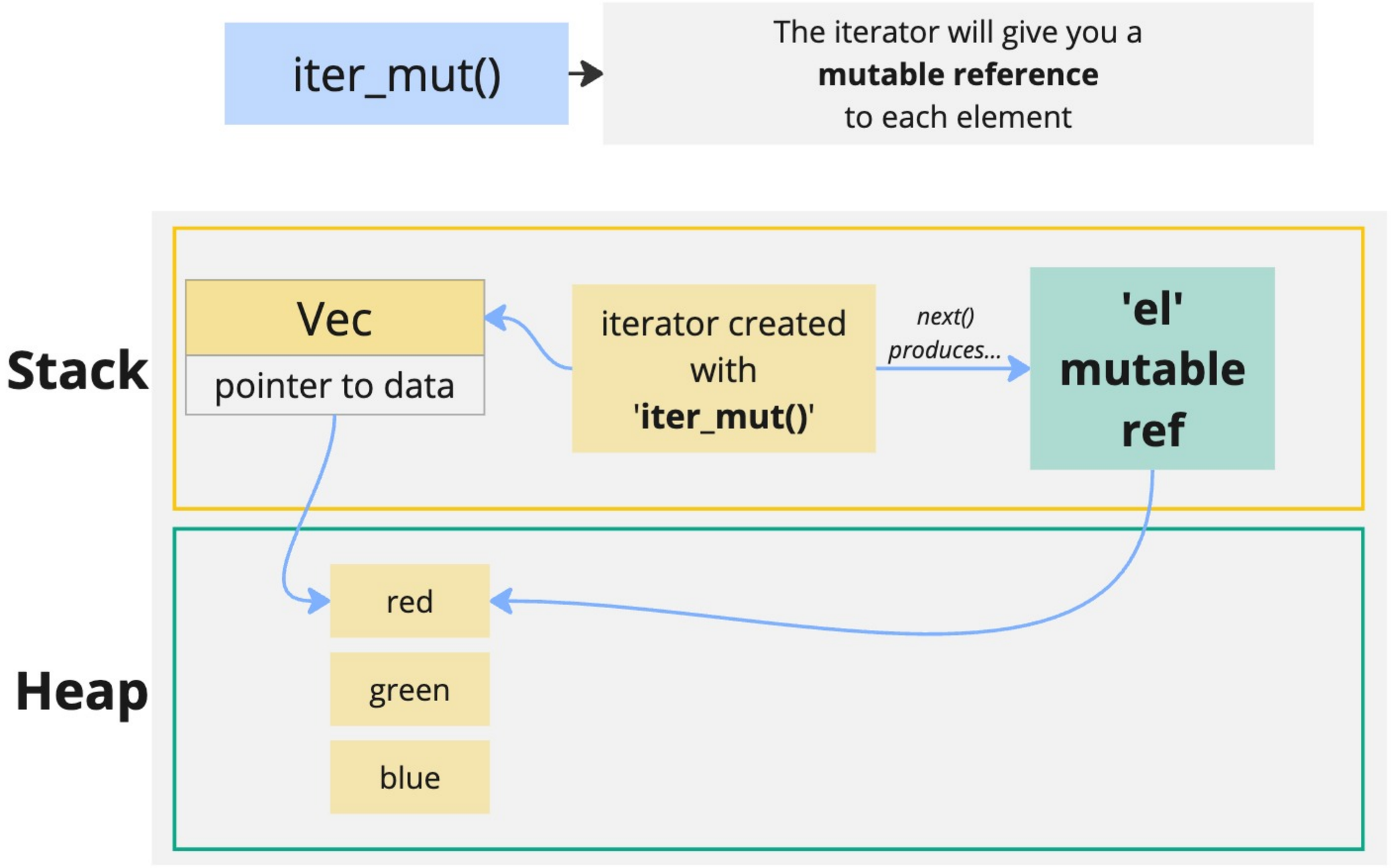
truncate()

Modifies a string in place

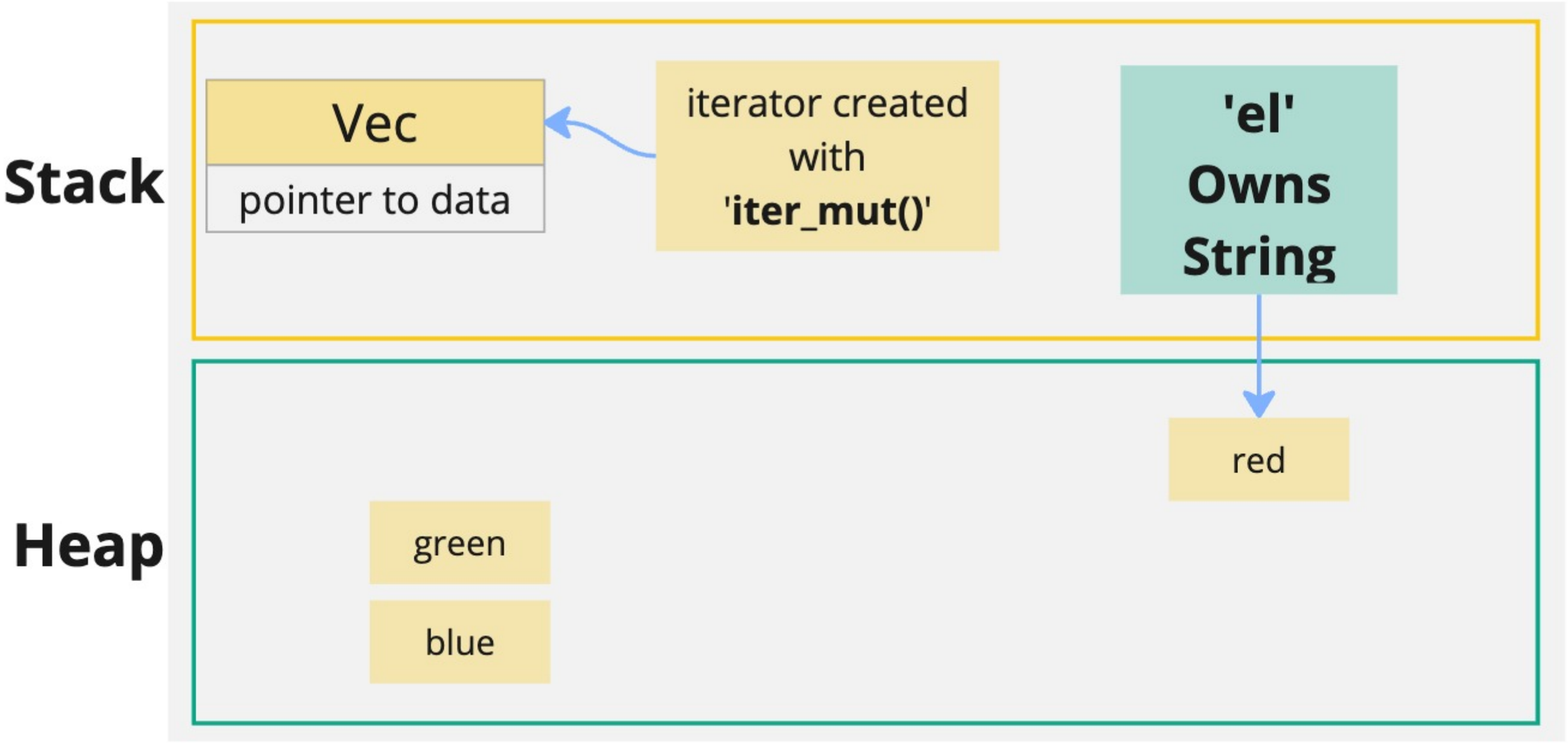
```
fn main() {
    let mut color = String::from("blue");

    color.truncate(1);
}
```

iter() → The iterator will give you a **read-only reference** to each element

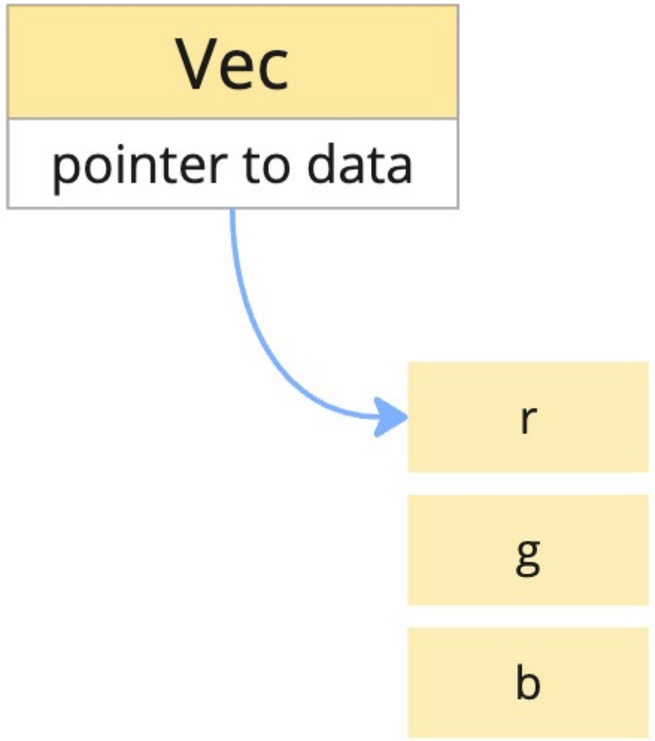



iter_mut()
The iterator will give you a
mutable reference
to each element
Stack
Vec
pointer to data
iterator created
with
'iter_mut()'
next()
produces...
'el'
mutable
ref
Heap
red
green
blue

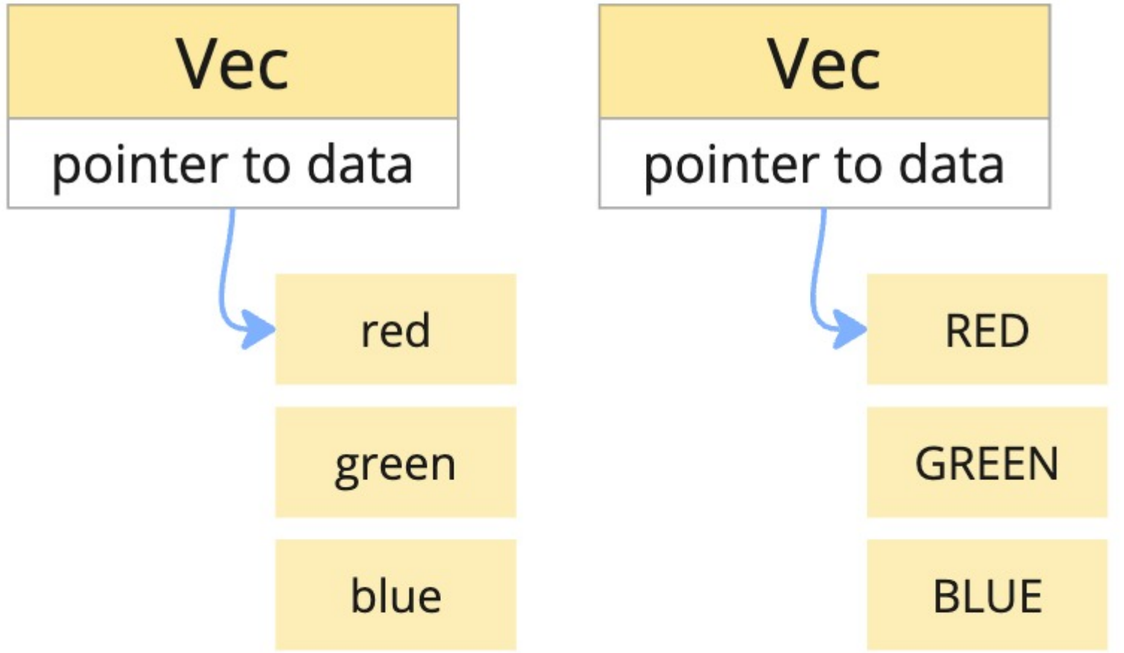
into_iter() → The iterator give you **ownership** of each element, unless called on a ref to a vector



shorten_strings()

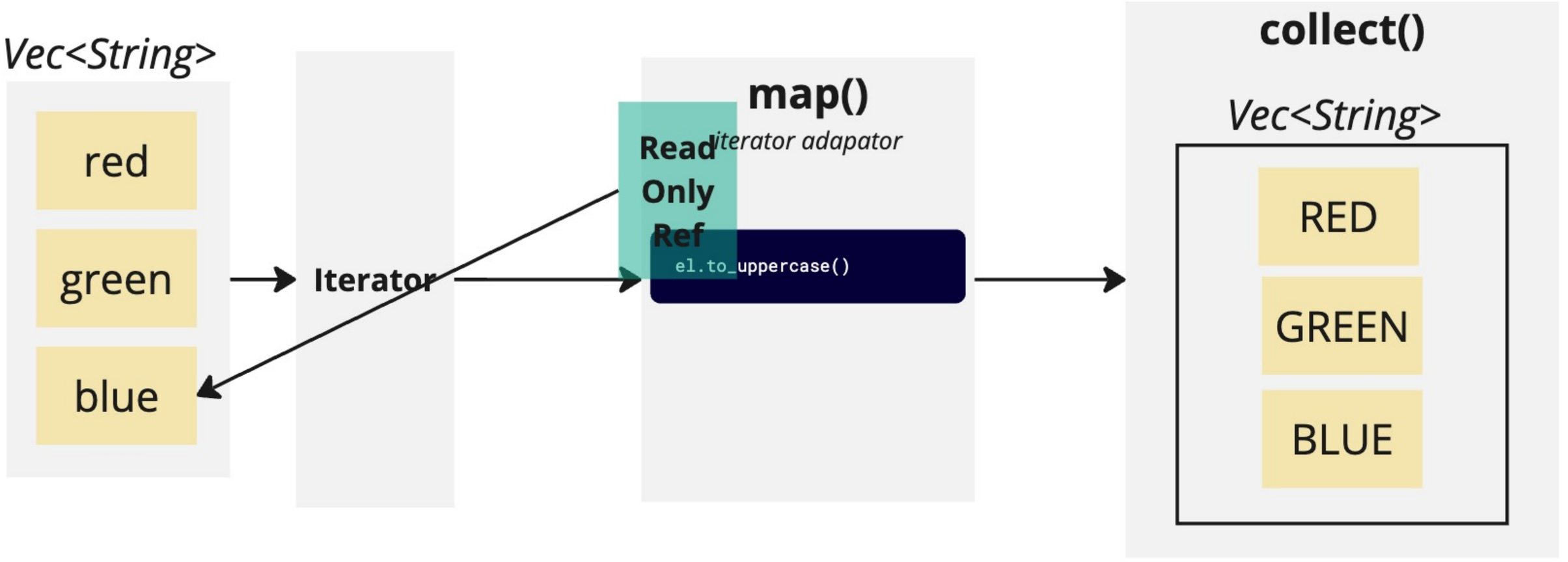


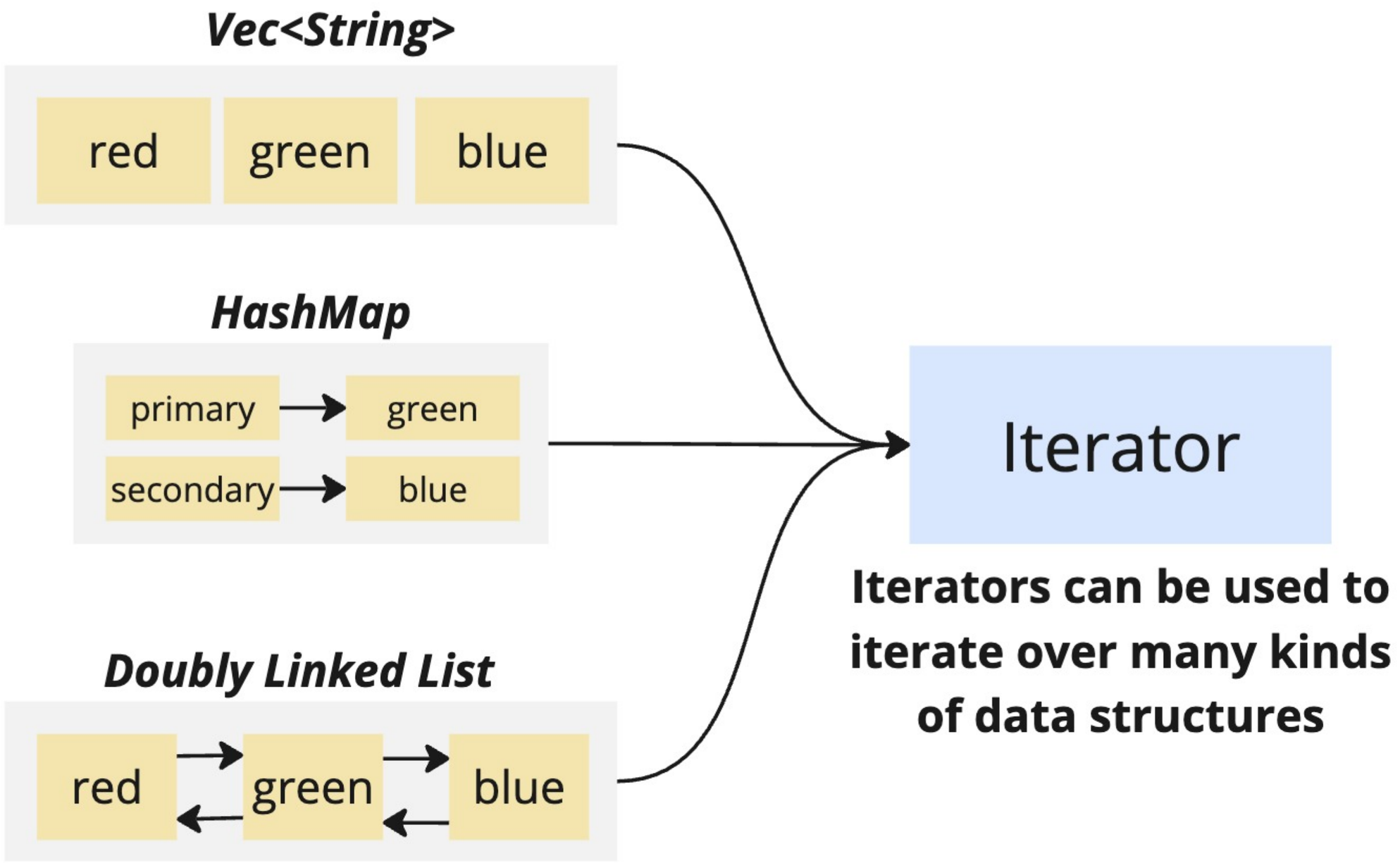
to_uppercase()

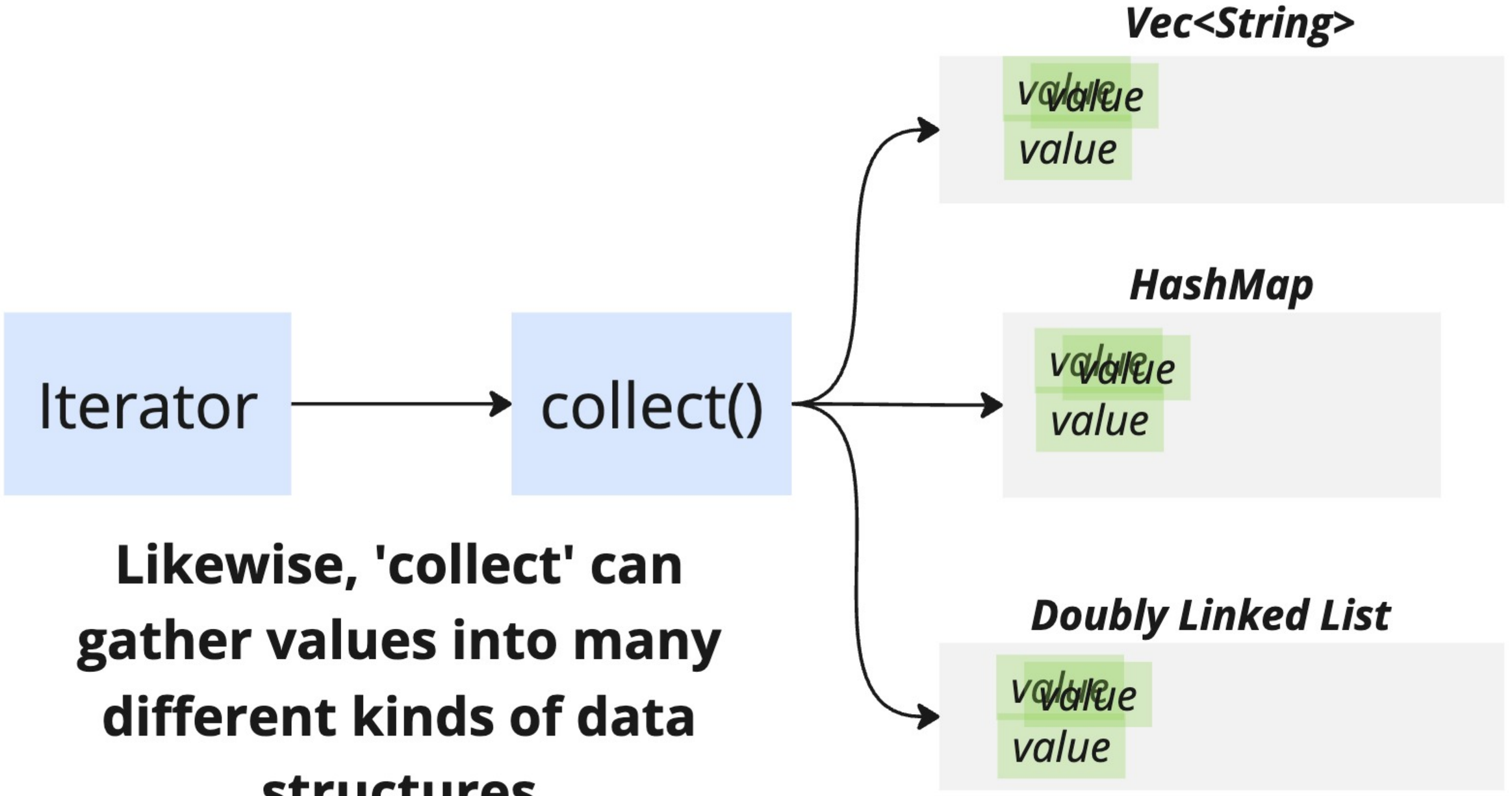


collect() is an iterator **consumer**

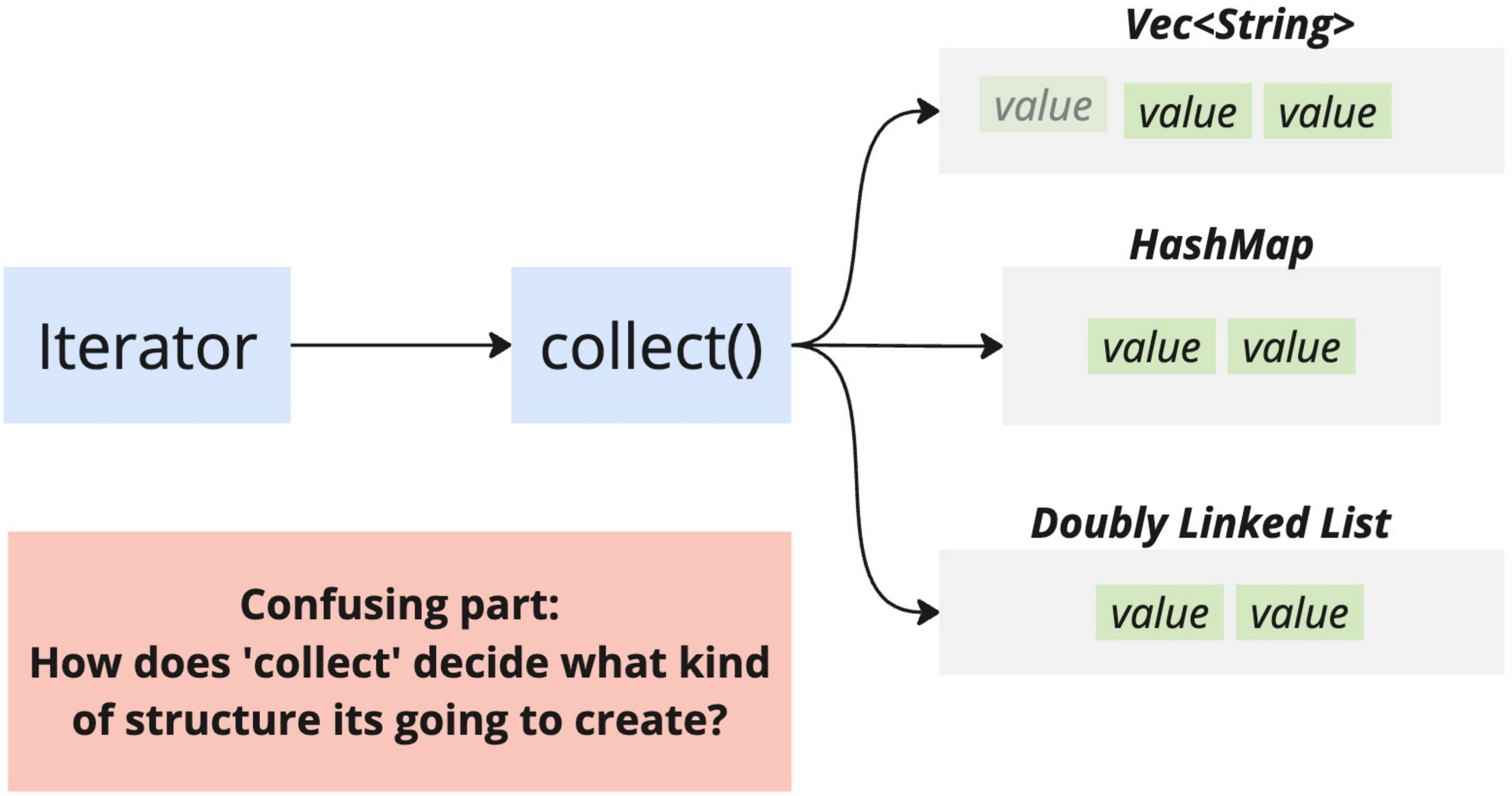
It will automatically
call 'next()' for you

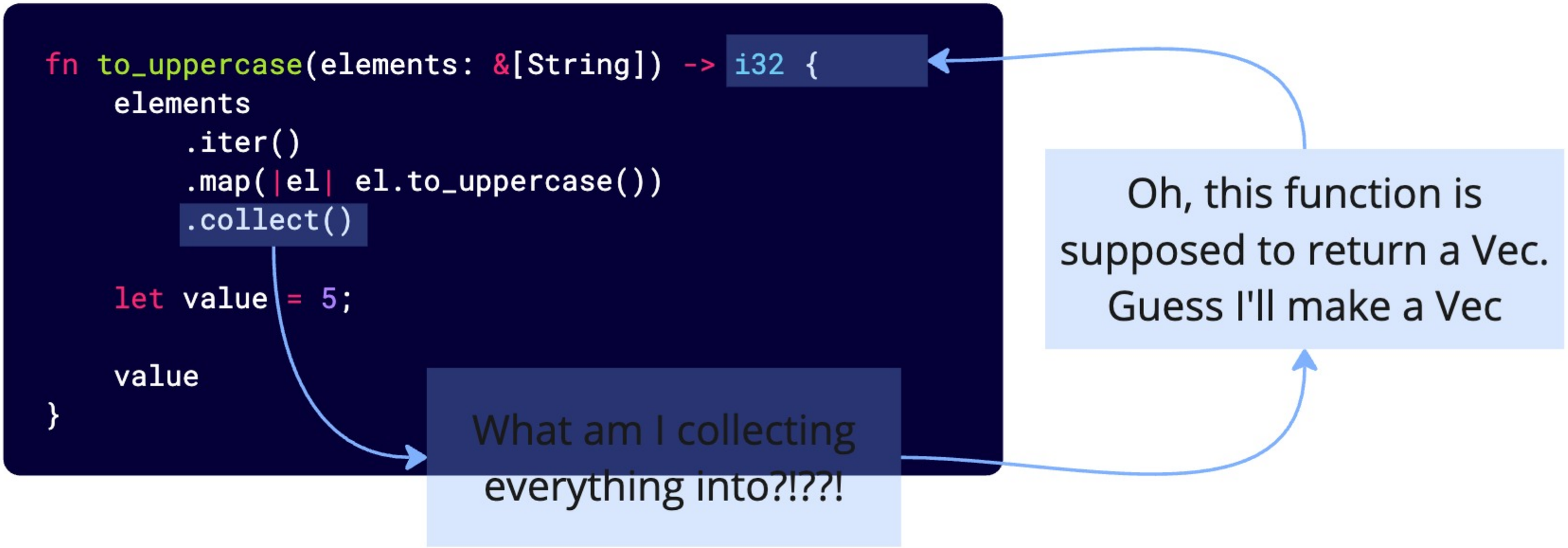



Vec<String>
red
green
blue
HashMap
primary
green
secondary
blue
Doubly Linked List
red
green
blue
Iterator
Iterators can be used to iterate over many kinds of data structures

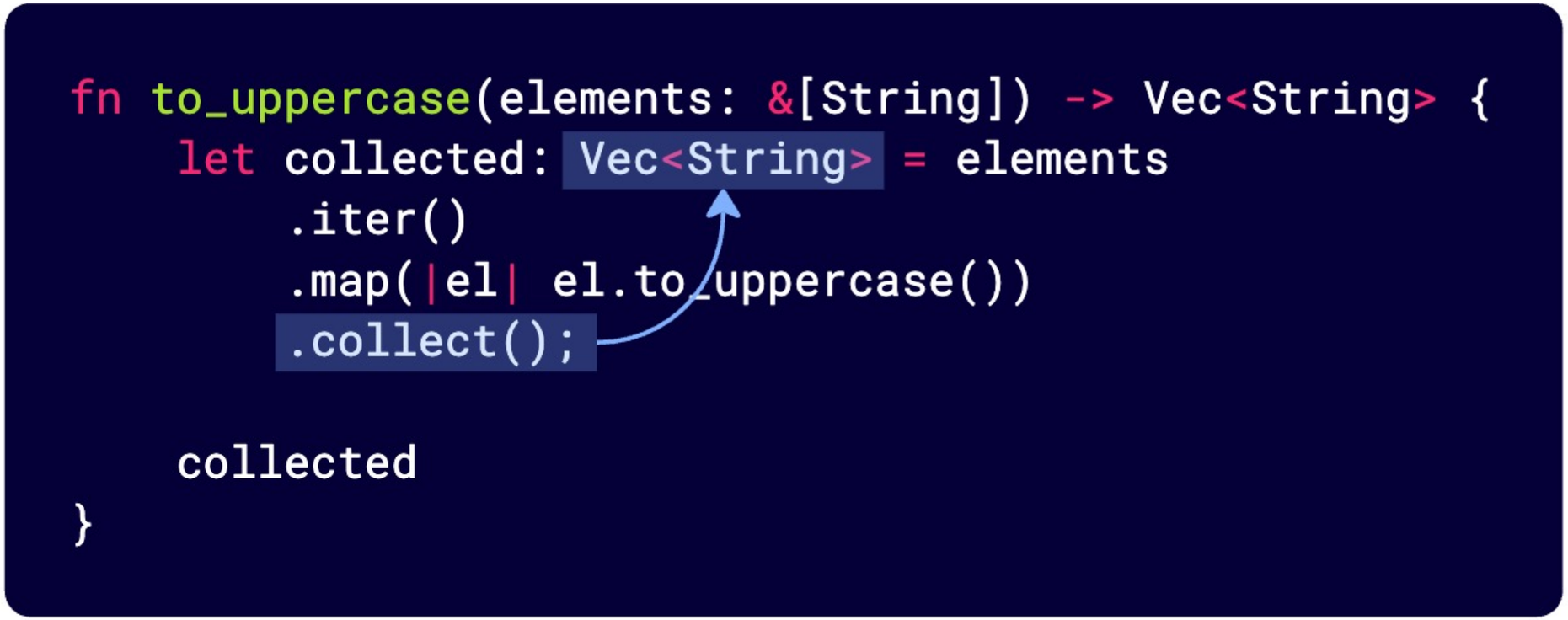


Likewise, 'collect' can gather values into many different kinds of data structures

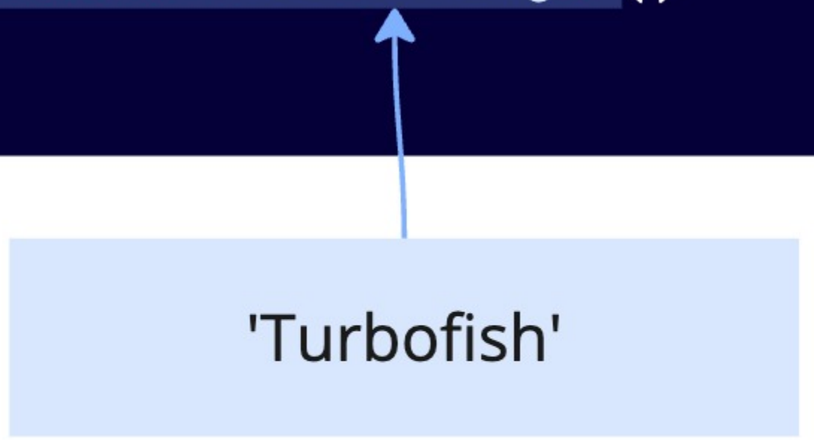

Vec<String>
value
value
value
HashMap
value
value
Doubly Linked List
value
value
Iterator
collect()
Confusing part:
How does 'collect' decide what kind of structure its going to create?


fn to_uppercase(elements: &[String]) -> i32 {
elements
.iter()
.map(|el| el.to_uppercase())
.collect()
let value = 5;
value
}
Oh, this function is supposed to return a Vec. Guess I'll make a Vec
What am I collecting everything into?!??!

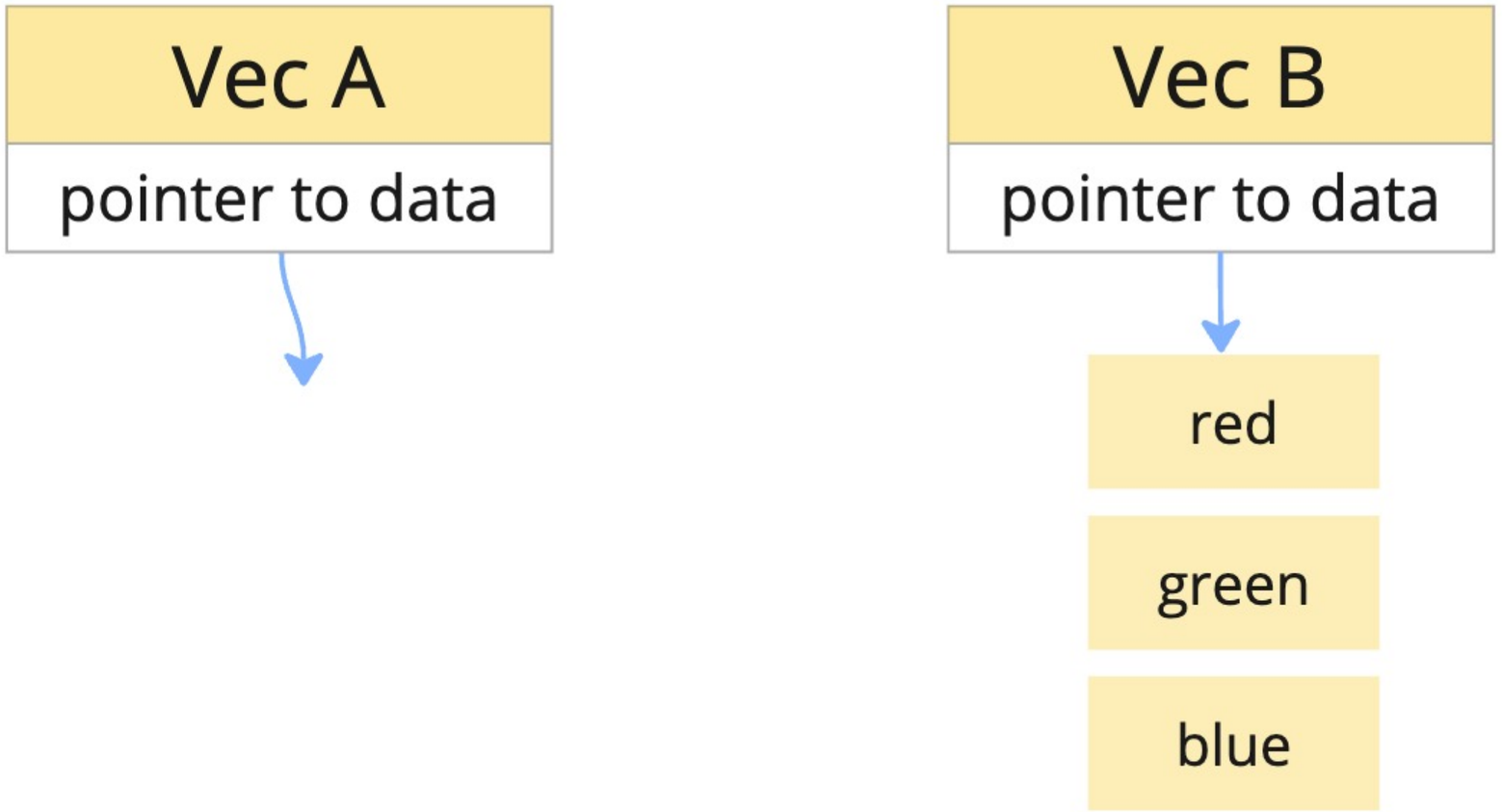
Collect will also look at variable type annotations



```rust
fn to_uppercase(elements: &[String]) -> Vec<String> {
    elements
        .iter()
        .map(|el| el.to_uppercase())
        .collect::<Vec<String>>()
}
```



move_elements(vec_a, vec_b)

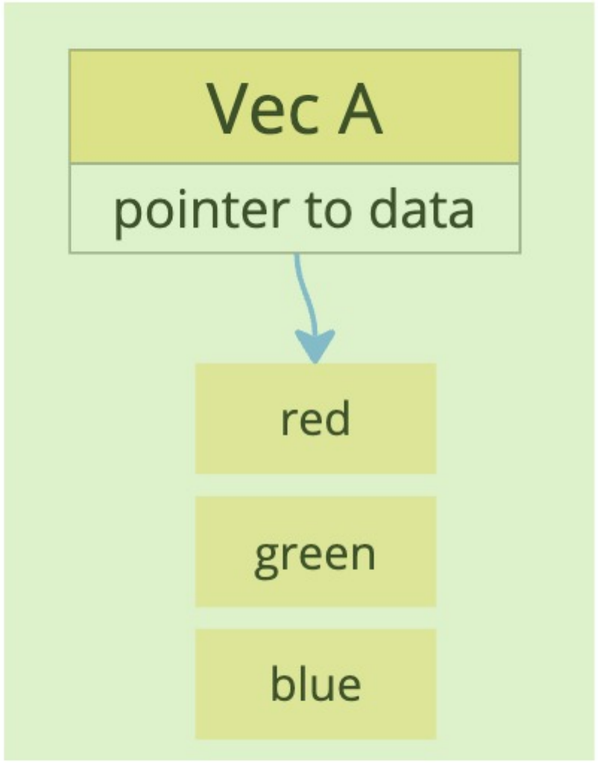


iter() →	The iterator will give you a **read-only reference** to each element
iter_mut() →	The iterator will give you a **mutable reference** to each element
into_iter() →	The iterator give you **ownership** of each element, *unless called on ref to a vector*

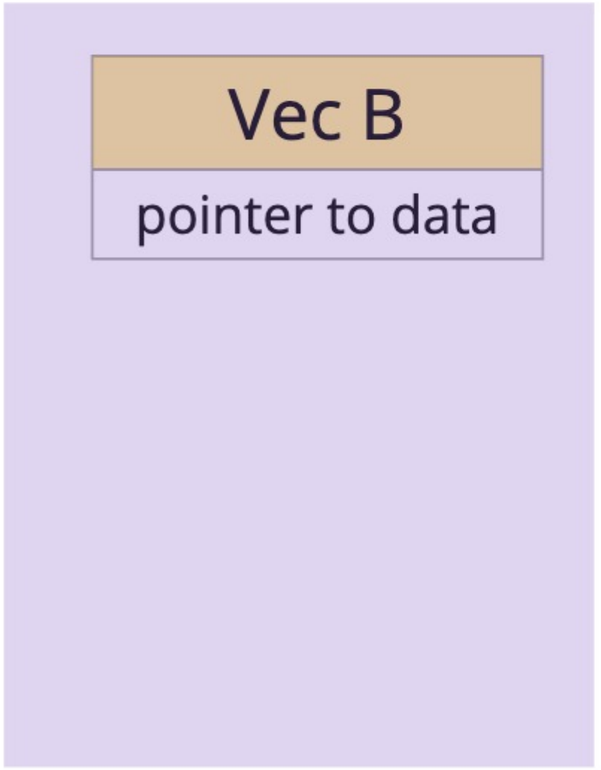
into_iter() will give you something different depending on how its called

`&colors.into_iter()`	Iterator created out of a **reference**	Iterator will produce **refs to each value**
`&mut colors.into_iter()`	Iterator created out of a **mutable reference**	Iterator will produce **mutable refs to each value**
`colors.into_iter()`	Iterator created out of a **value**	Iterator will produce each **value. Also moves ownership of these values.**

move_elements(vec_a, vec_b)



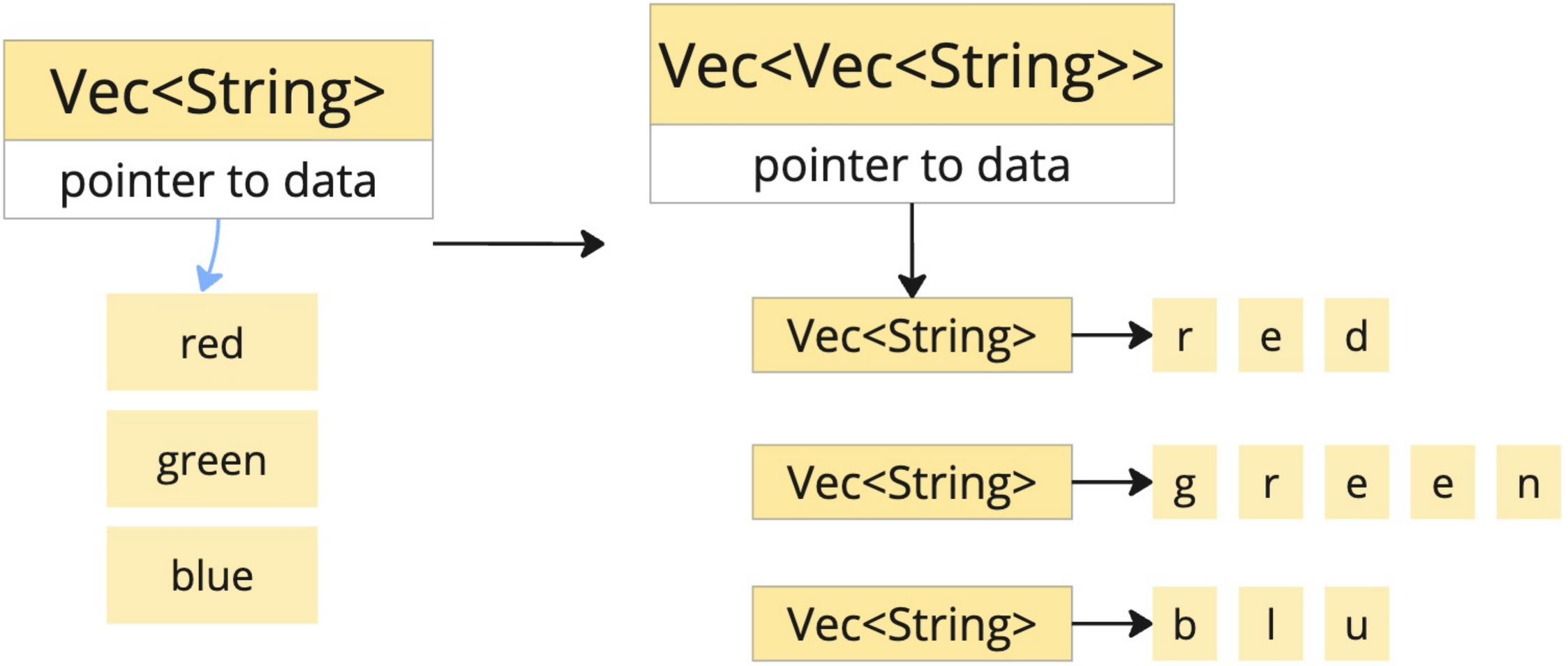
We are trying to **store** these values

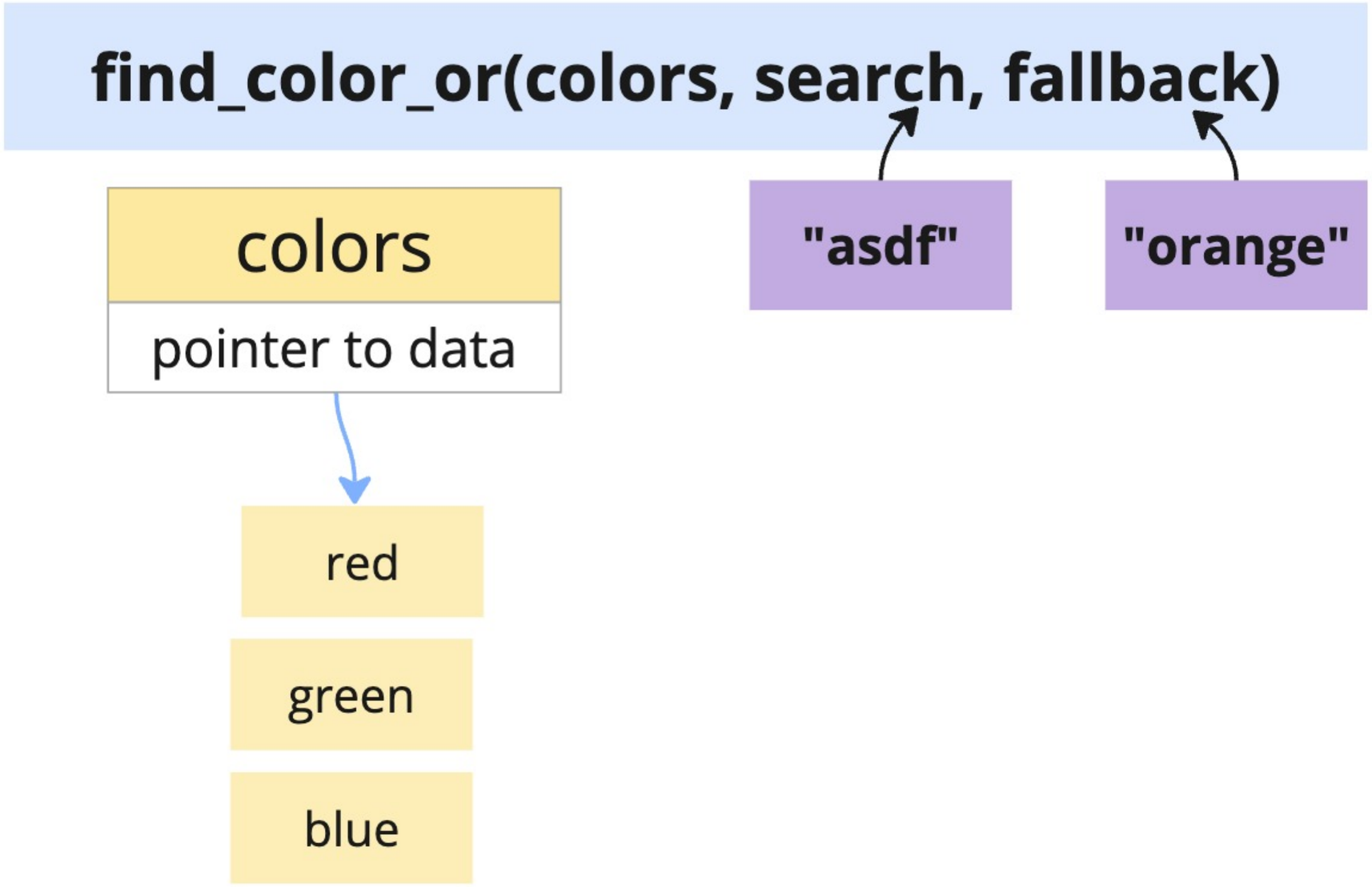


We are trying to **modify** this value

```rust
let colors = vec![
    String::from("red"),
    String::from("green"),
    String::from("blue"),
];
```

Name	Description
shorten_strings()	Shortens each string in the vector to 1 character
move_elements()	Moves elements from one vector to another
print_elements()	Prints each element in the vector one by one
to_uppercase()	Return a new vector with each element capitalized
explode()	Turns a Vec<String> into a Vec<Vec<String>>
find_color_or()	Finds a matching element or returns a fallback


Vec<String>
pointer to data
red
green
blue
Vec<Vec<String>>
pointer to data
Vec<String>
r
e
d
Vec<String>
g
r
e
e
n
Vec<String>
b
l
u


find_color_or(colors, search, fallback)
colors
pointer to data
"asdf"
"orange"
red
green
blue

```rust
fn find_color_or(elements: &[String], search: String, fallback: String) -> String
```

```rust
fn find_color_or(elements: &[String], search: &str, fallback: &str) -> &str
```

```rust
fn find_color_or(elements: &[String], search: ___, fallback: ___) -> ___
```

```
fn find_color_or(
    elements: &[String],
    search: ___,
    fallback: ___
)
    -> ___
```

Which of the following do we need to do?

C	Use it in a calculation

Function Argument Types

Need to store the argument somewhere?	**Favor taking ownership (receive a value)**
Need to do a calculation with the value?	**Favor receiving a read-only ref**
Need to change the value in some way?	**Favor receiving a mutable ref**

Which of the following do we need to do?

C Use it in a calculation

```
fn find_color_or(
    elements: &[String],
    search: &str,
    fallback: &str
) -> ___ {}
```

Return a &str?

```rust
fn find_color_or(
    elements: &[String],
    search: &str,
    fallback: &str
) -> &str {}

fn check_colors() {
    let colors = vec![
        /* list of colors */
    ];

    let result = find_color_or(
        &colors,
        "re",
        "orange"
    );

    result
    // 'colors' goes out of scope
}
```

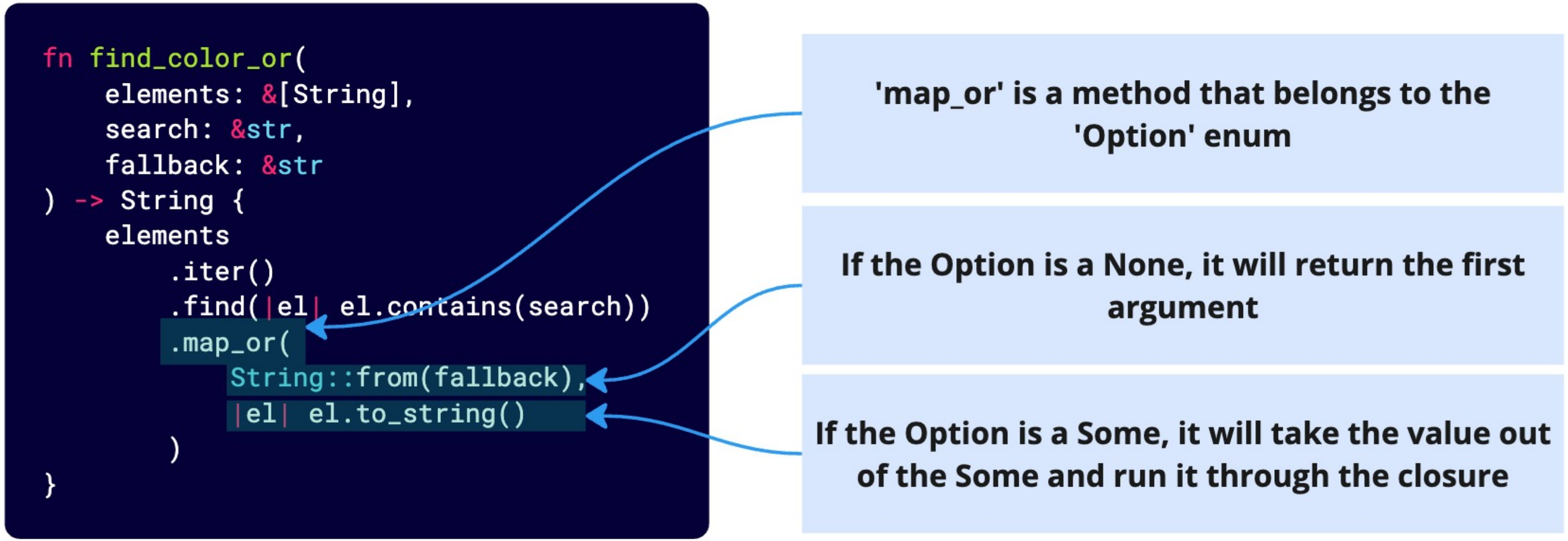
'colors' binding	
'result' binding	**Ref to value**

```
fn find_color_or(elements: &[String], search: &str, fallback: &str) -> String {
    /*

    for each element in elements
        if the element contains search
            return the element as a String

    return the fallback as a String

    */
}
```

```rust
fn find_color_or(
    elements: &[String],
    search: &str,
    fallback: &str
) -> String {
    elements
        .iter()
        .find(|el| el.contains(search))
        .map_or(
            String::from(fallback),
            |el| el.to_string()
        )
}
```

Calls 'next' on the iterator until it gets an element that returns a truthy value from the closure function

Returns an 'Option'
Some(color) if it found something
None if it didn't find anything


fn find_color_or(
elements: &[String],
search: &str,
fallback: &str
) -> String {
elements
.iter()
.find(|el| el.contains(search))
.map_or(
String::from(fallback),
|el| el.to_string()
)
}
'map_or' is a method that belongs to the 'Option' enum
If the Option is a None, it will return the first argument
If the Option is a Some, it will take the value out of the Some and run it through the closure