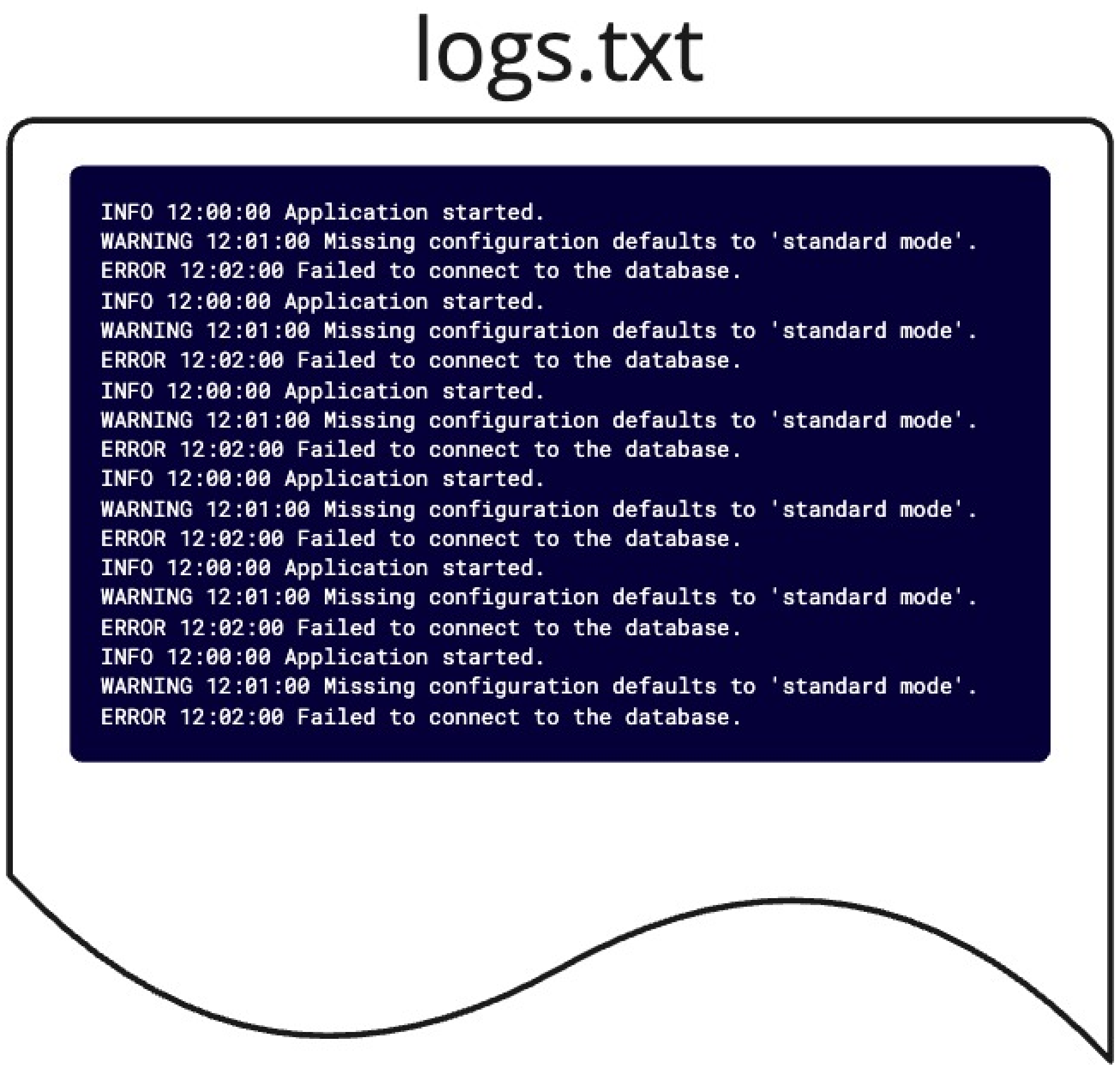
Open and parse the logs.txt file

Extract some useful data from the file

Make sure we have robust error handling



Something Went Wrong?

Panic

- Something went wrong and we **don't know how to deal with it**
- You *can* recover from a panic, but debatable if you would want to

Use a Result or Option

- Something went wrong, but we kind of **expected it** and **can deal** with the error
- Relies on built-in enums like **Option** and **Result**, or custom error-handling enums

```rust
fn main() {
    println!("{}", divide(10.0, 0.0));
}

fn divide(a: f64, b: f64) -> _ {
    if b == 0.0 {
        // Uh oh, division by 0...
        // Return something to indicate
            an error occured
    }

    a / b
}
```

```rust
fn main() {
    println!("{}", divide(10.0, 0.0));
}

fn divide(a: f64, b: f64) -> _ {
    if b == 0.0 {
        // Uh oh, division by 0...
        // Want to use the REsult enum
            in some way
    }

    a / b
}
```

We're Authoring 'divide'...

Is the error so bad that we couldn't possibly keep running our program, or is it a completely unexpected error??

Panic

Is this an operation that can either succeed or fail?

Use the Result enum

Actual implementation of Result has some extra stuff

```
enum Result {
    Ok(value),
    Err(error)
}
```

Result is used when we need to know if something worked or failed

Ok() variant is used when something went well

Err() variant used when something bad happened

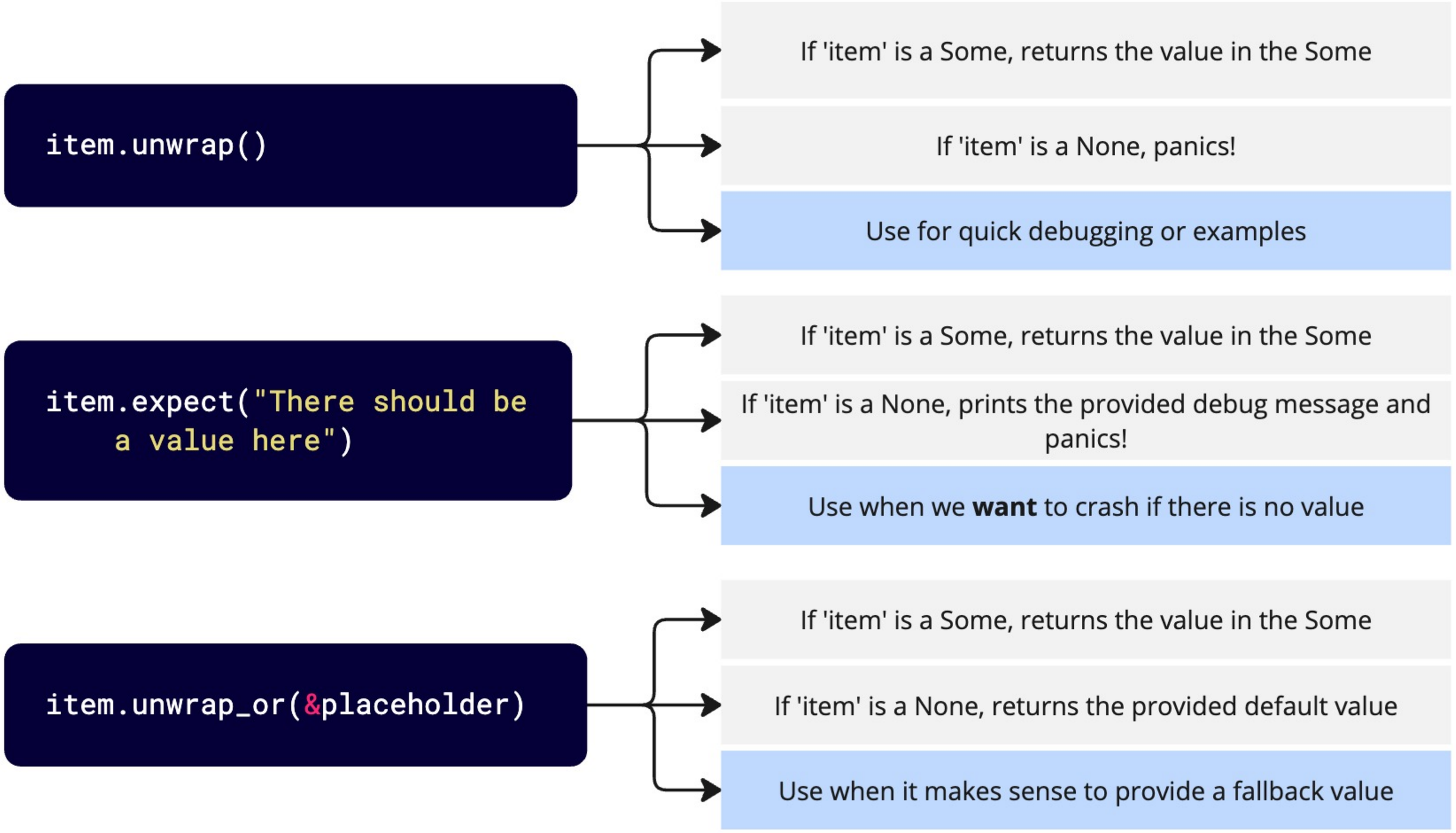
```
enum Option {
    Some(value),
    None
}
```

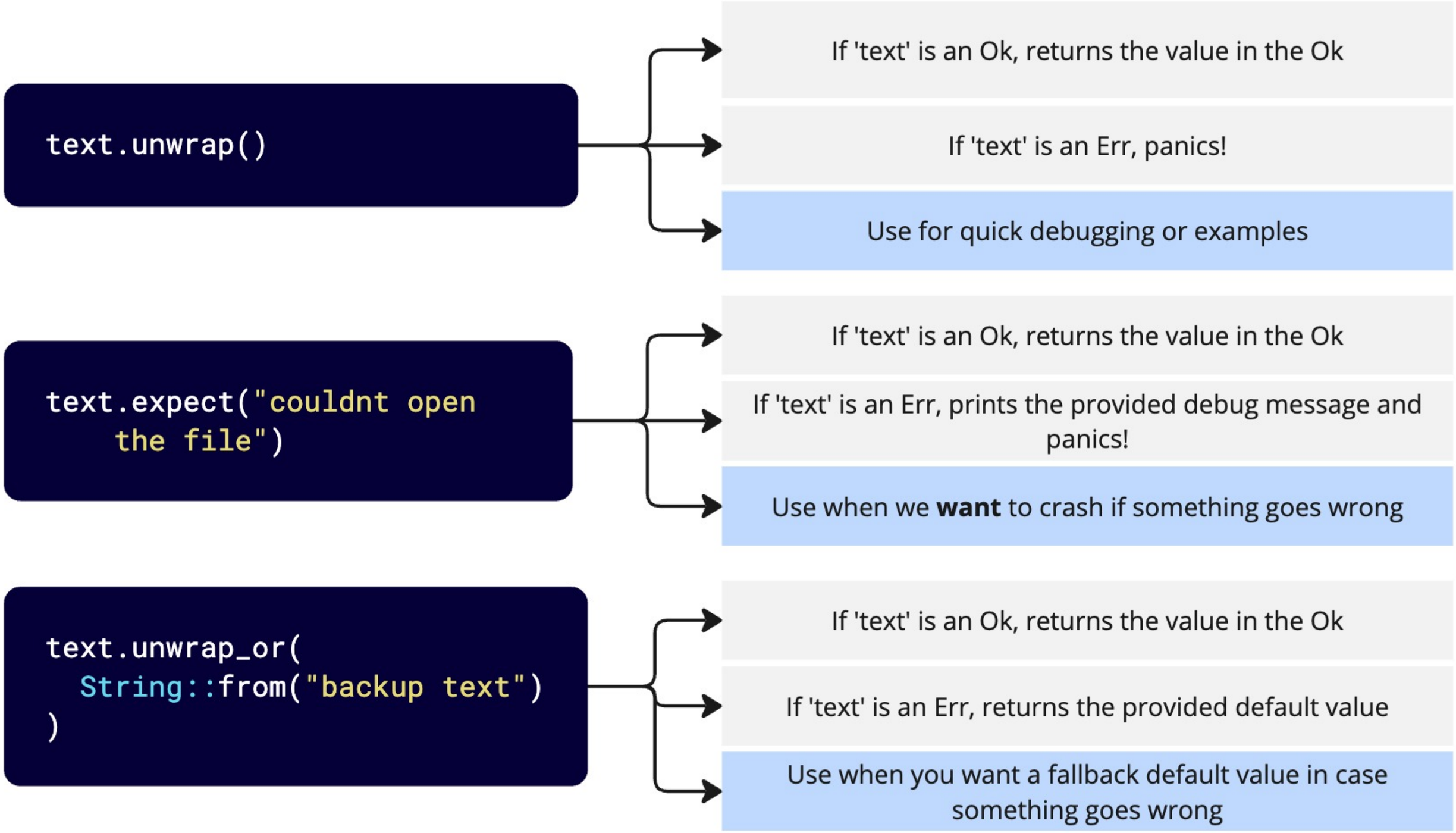
Option is used when we need to know if a value is present or not

Some() variant used when we have a value

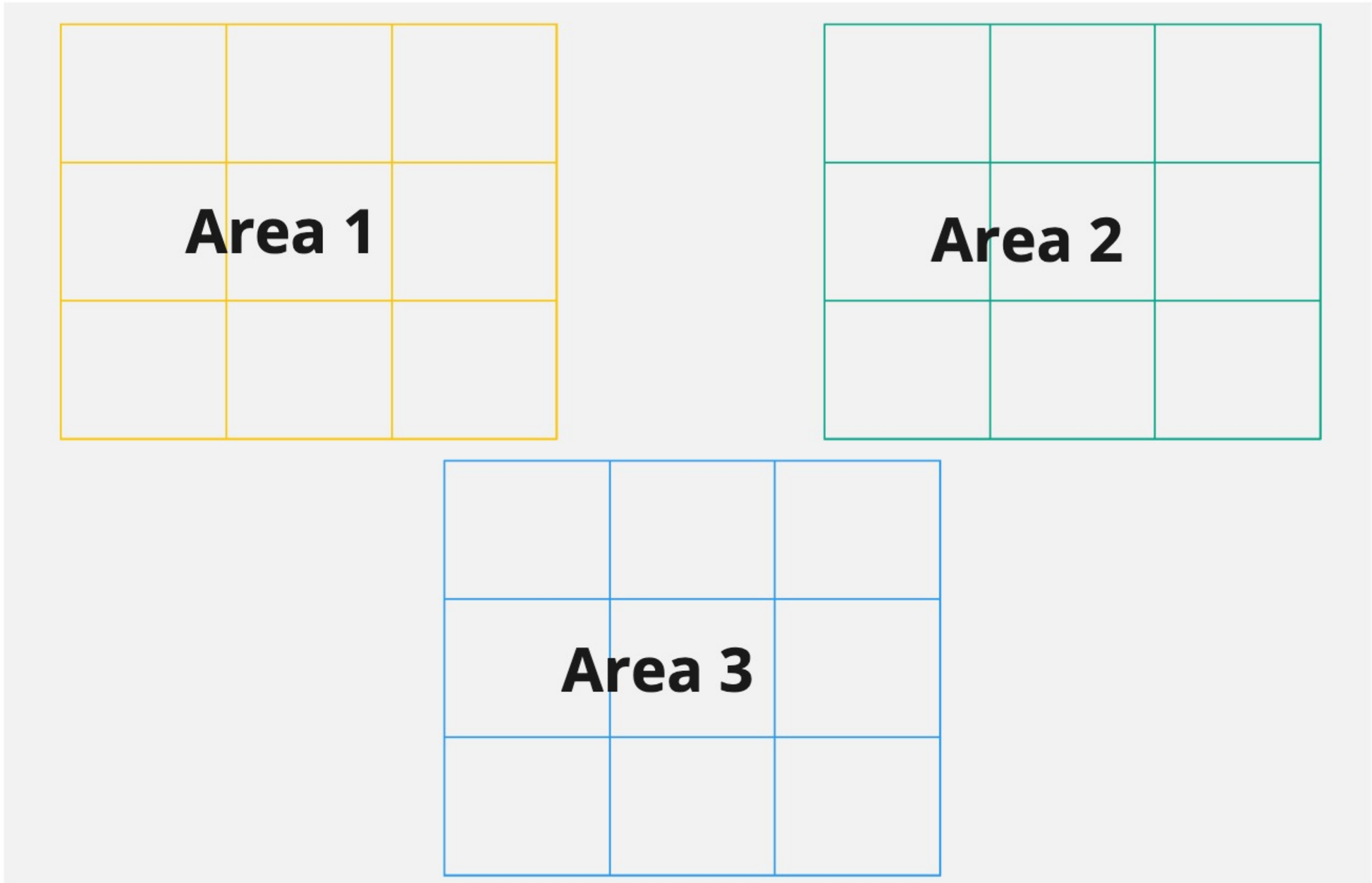
None variant used when there is no value

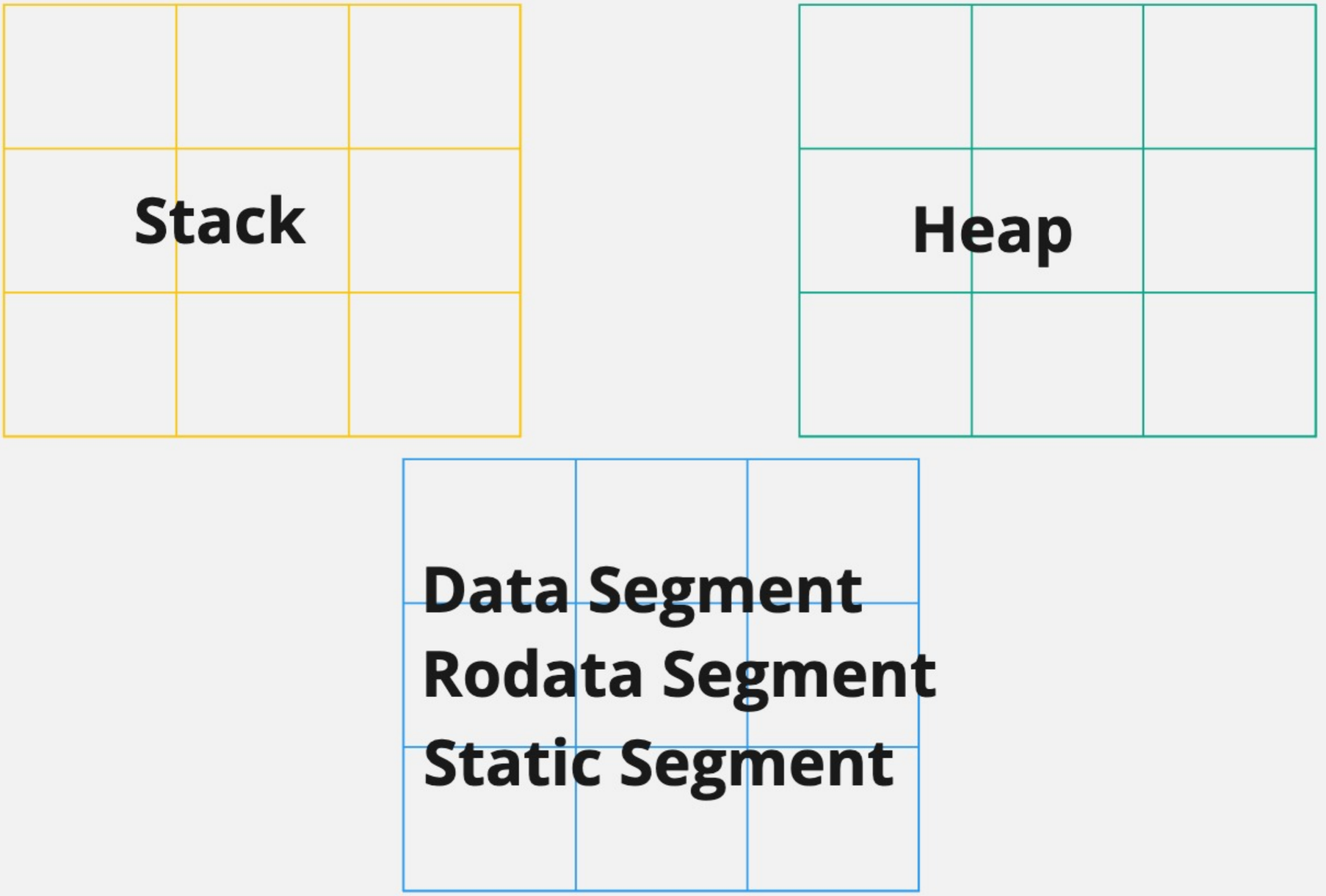
```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

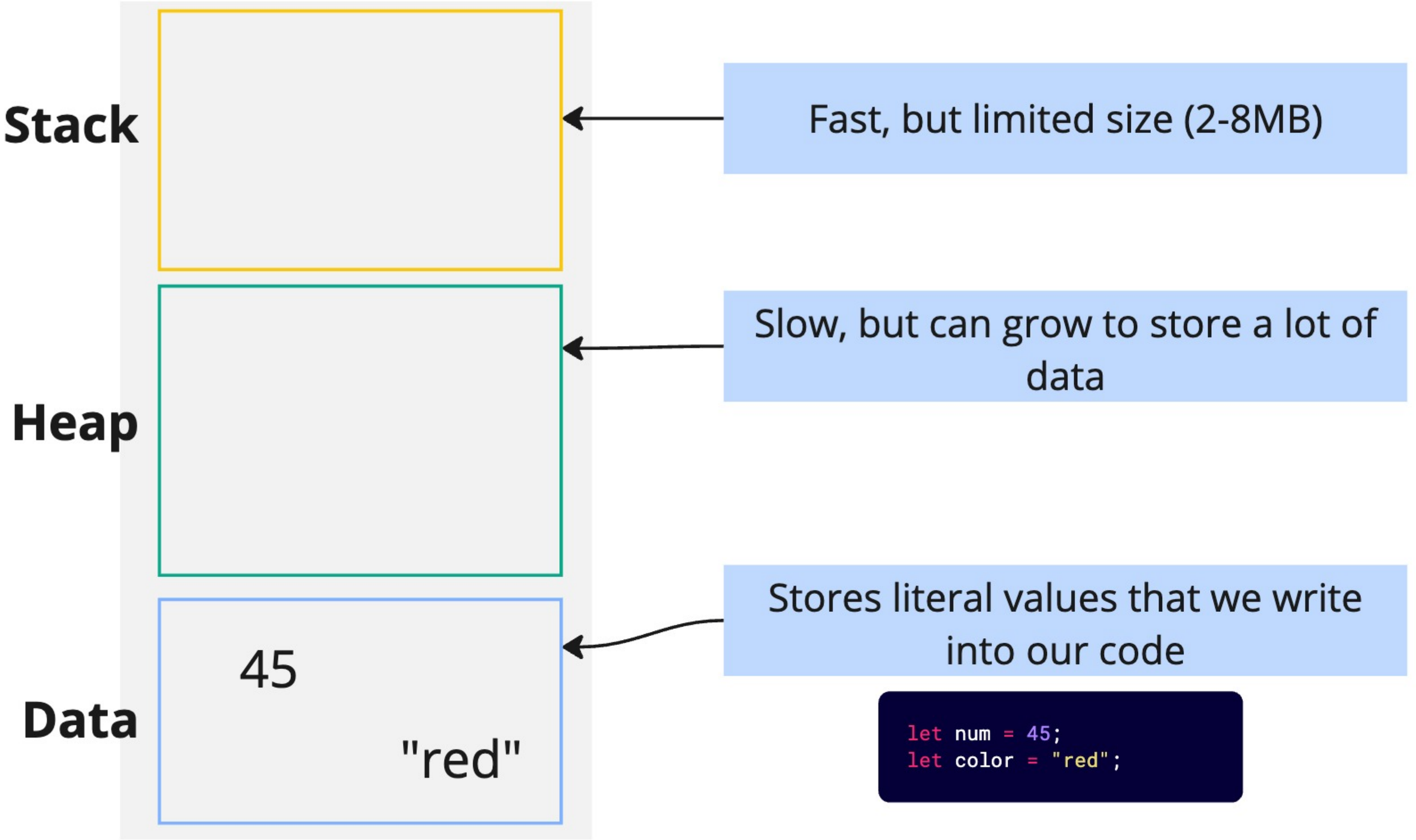

item.unwrap()
If 'item' is a Some, returns the value in the Some
If 'item' is a None, panics!
Use for quick debugging or examples
item.expect("There should be a value here")
If 'item' is a Some, returns the value in the Some
If 'item' is a None, prints the provided debug message and panics!
Use when we **want** to crash if there is no value
item.unwrap_or(&placeholder)
If 'item' is a Some, returns the value in the Some
If 'item' is a None, returns the provided default value
Use when it makes sense to provide a fallback value

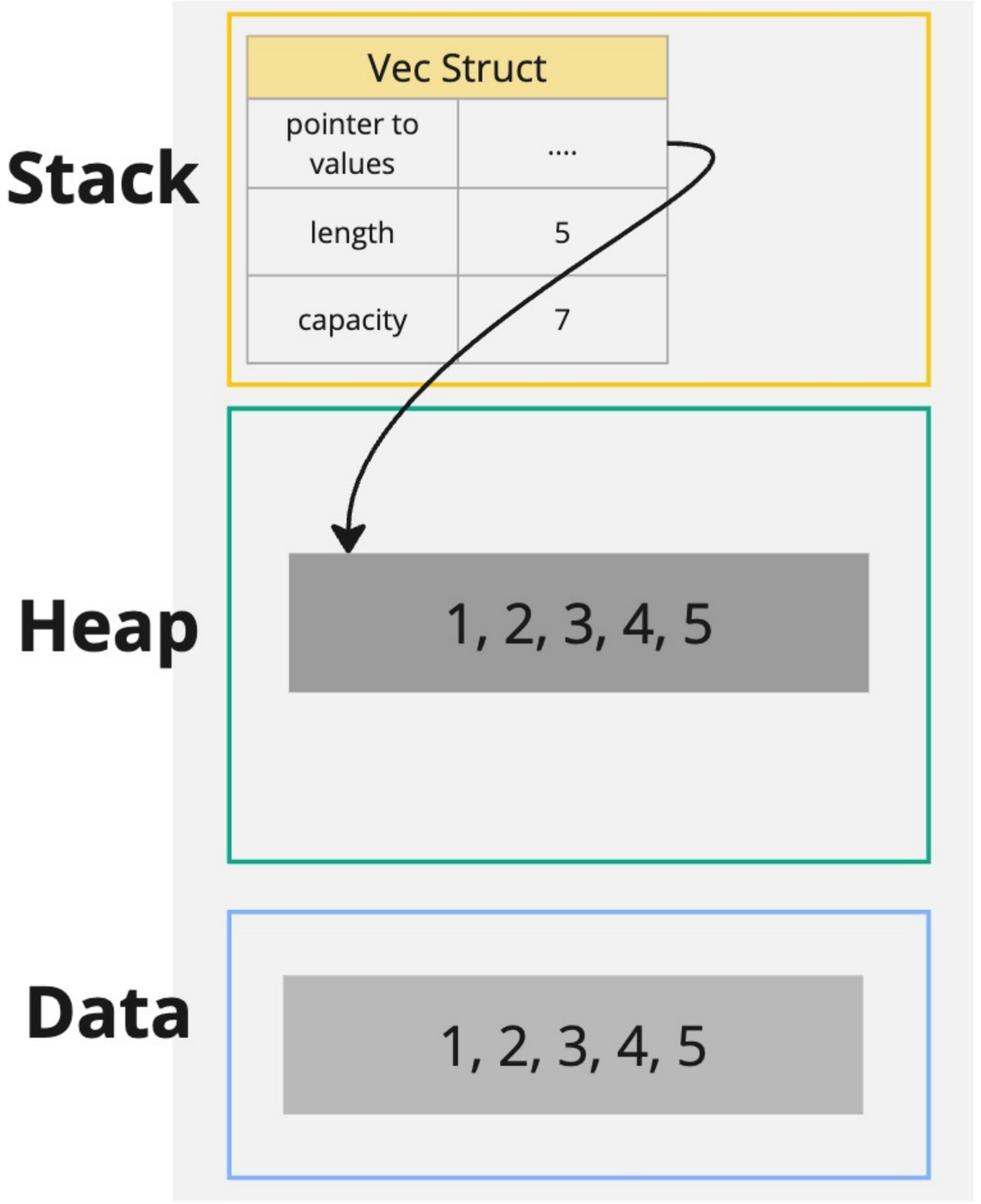

text.unwrap()
If 'text' is an Ok, returns the value in the Ok
If 'text' is an Err, panics!
Use for quick debugging or examples
text.expect("couldnt open the file")
If 'text' is an Ok, returns the value in the Ok
If 'text' is an Err, prints the provided debug message and panics!
Use when we **want** to crash if something goes wrong
text.unwrap_or(String::from("backup text"))
If 'text' is an Ok, returns the value in the Ok
If 'text' is an Err, returns the provided default value
Use when you want a fallback default value in case something goes wrong

Computer Memory




Stack
Heap
Data Segment
Rodata Segment
Static Segment


Stack
Heap
Data
45
"red"
Fast, but limited size (2-8MB)
Slow, but can grow to store a lot of data
Stores literal values that we write into our code
let num = 45;
let color = "red";



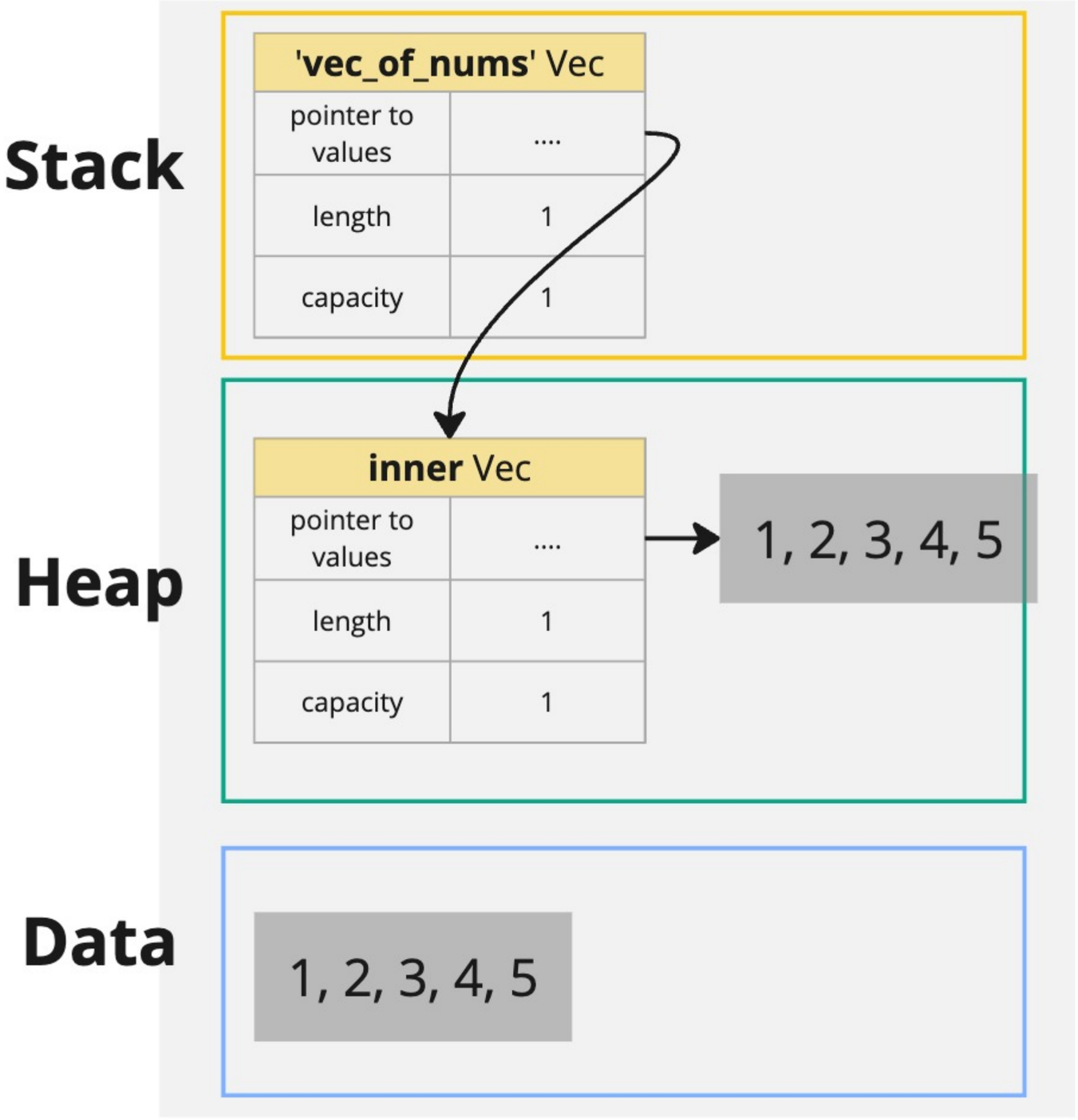

```rust
let nums = vec![1,2,3,4,5];
```

Super Common Pattern

Stack stores metadata about a datastructure

Heap stores the actual data

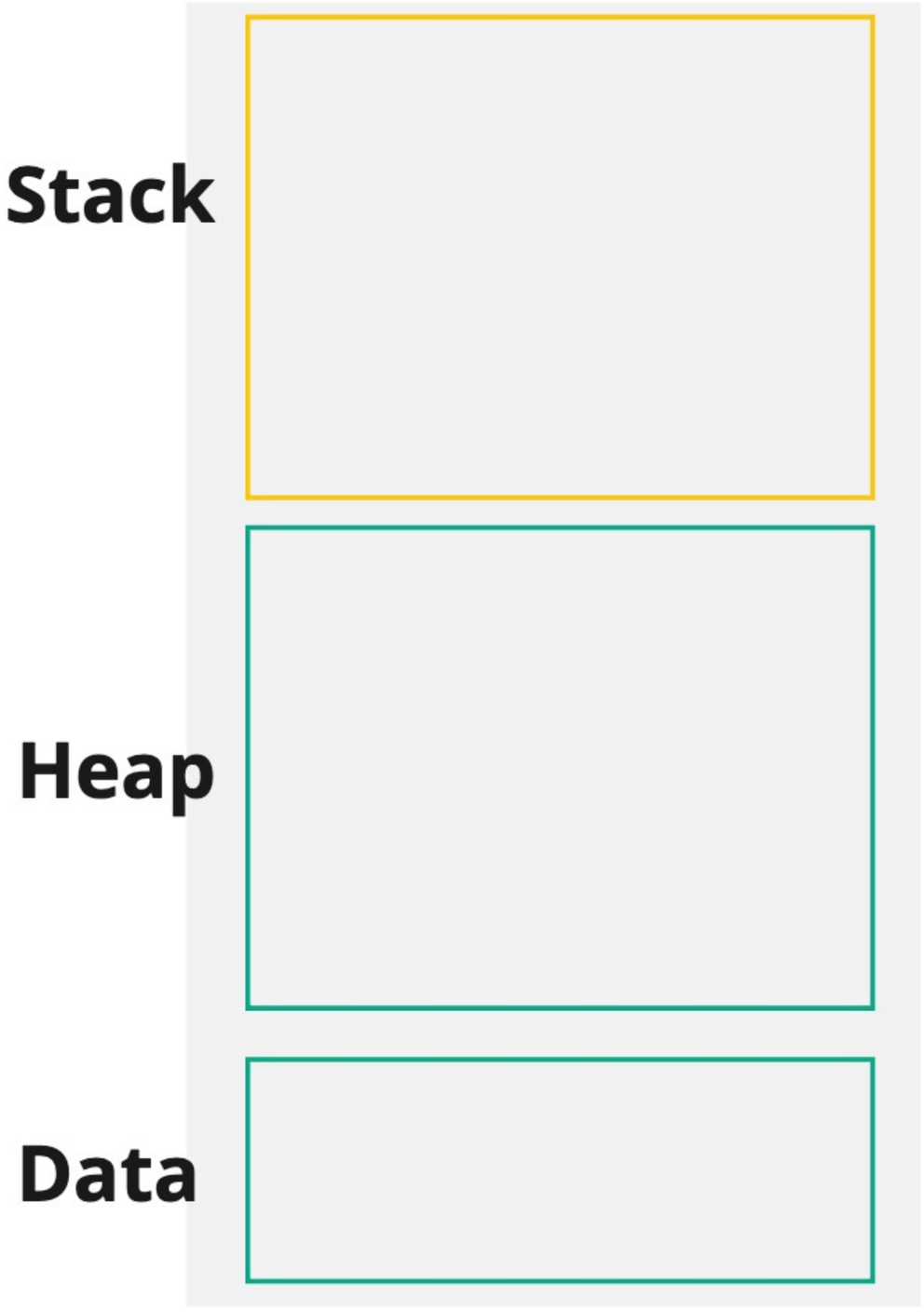
Avoids running out of memory in the stack if the datastructure grows to hold a lot of data



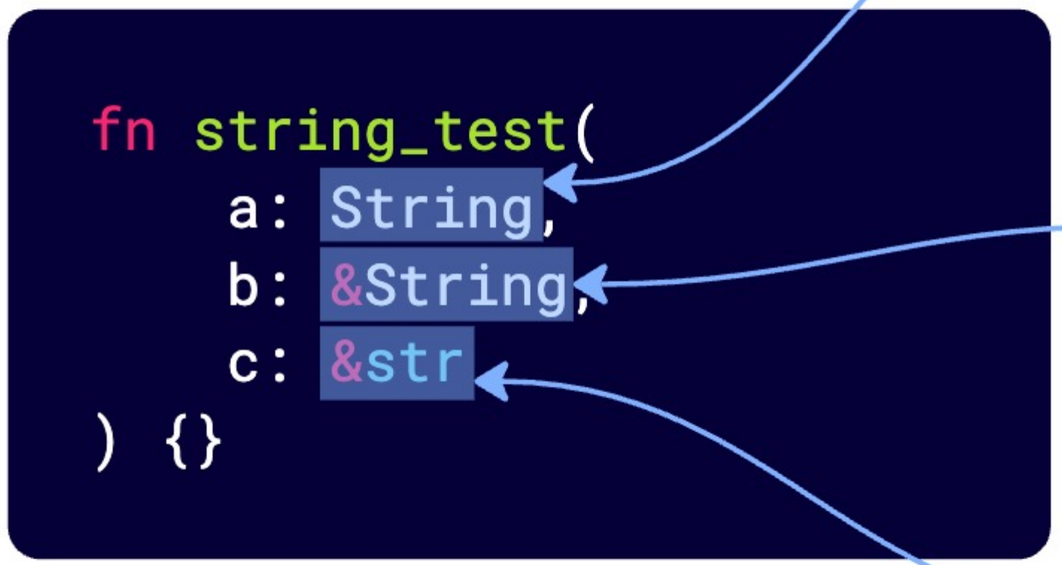

```
let vec_of_nums = vec![
    vec![1, 2, 3, 4, 5]
];
```

Corner Case

If a data structure owns another data structure, the child's metadata will be placed on the heap




```
fn string_test(
    a: String,
    b: &String,
    c: &str
) {}
```



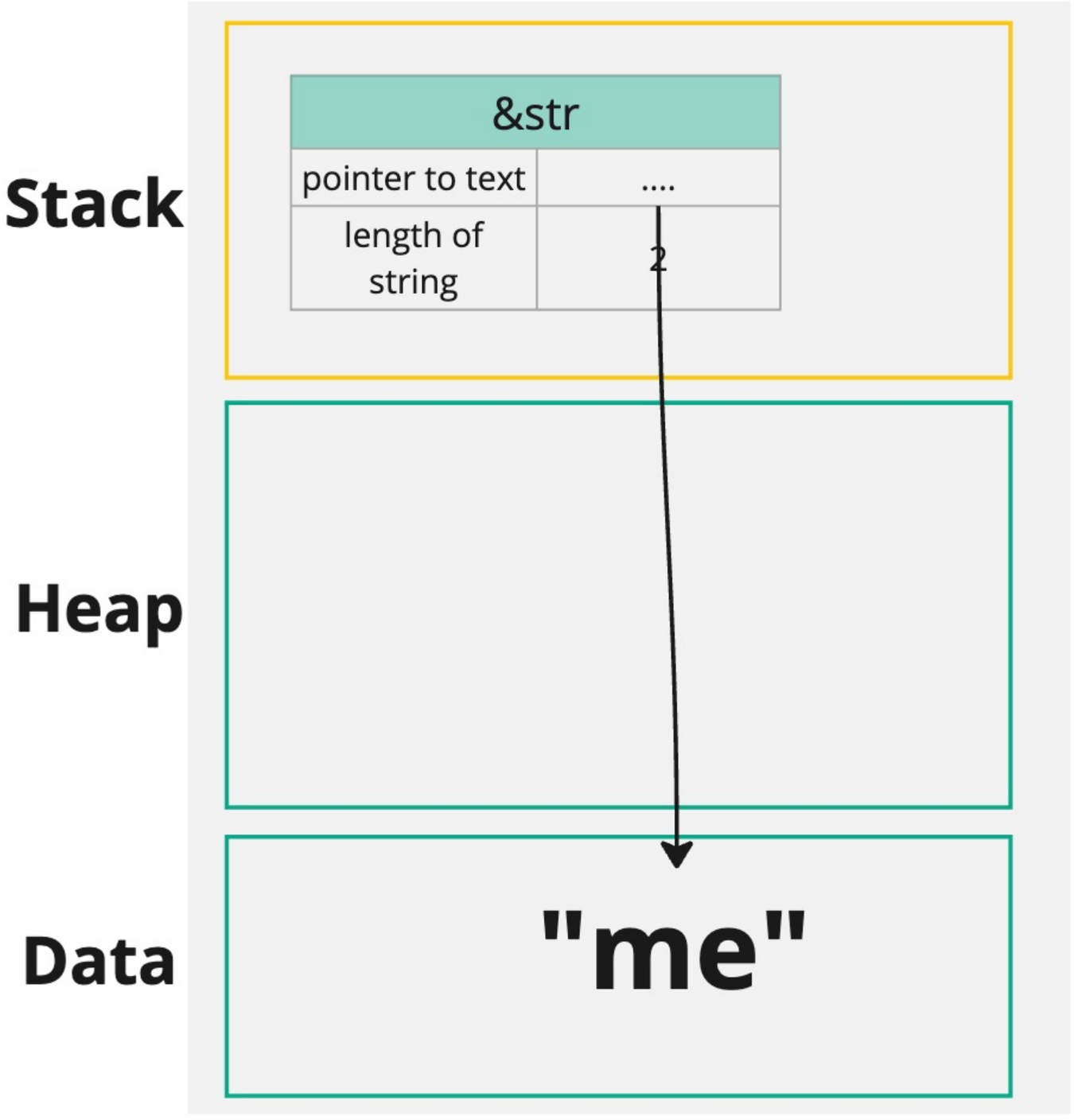
A **String** instance.
Has a struct in the stack and text data in the heap

A **read-only ref to a String** instance.
Can't be used to change the text data

A **read-only string slice**.
Has a struct in the stack and text data in the heap.

&str

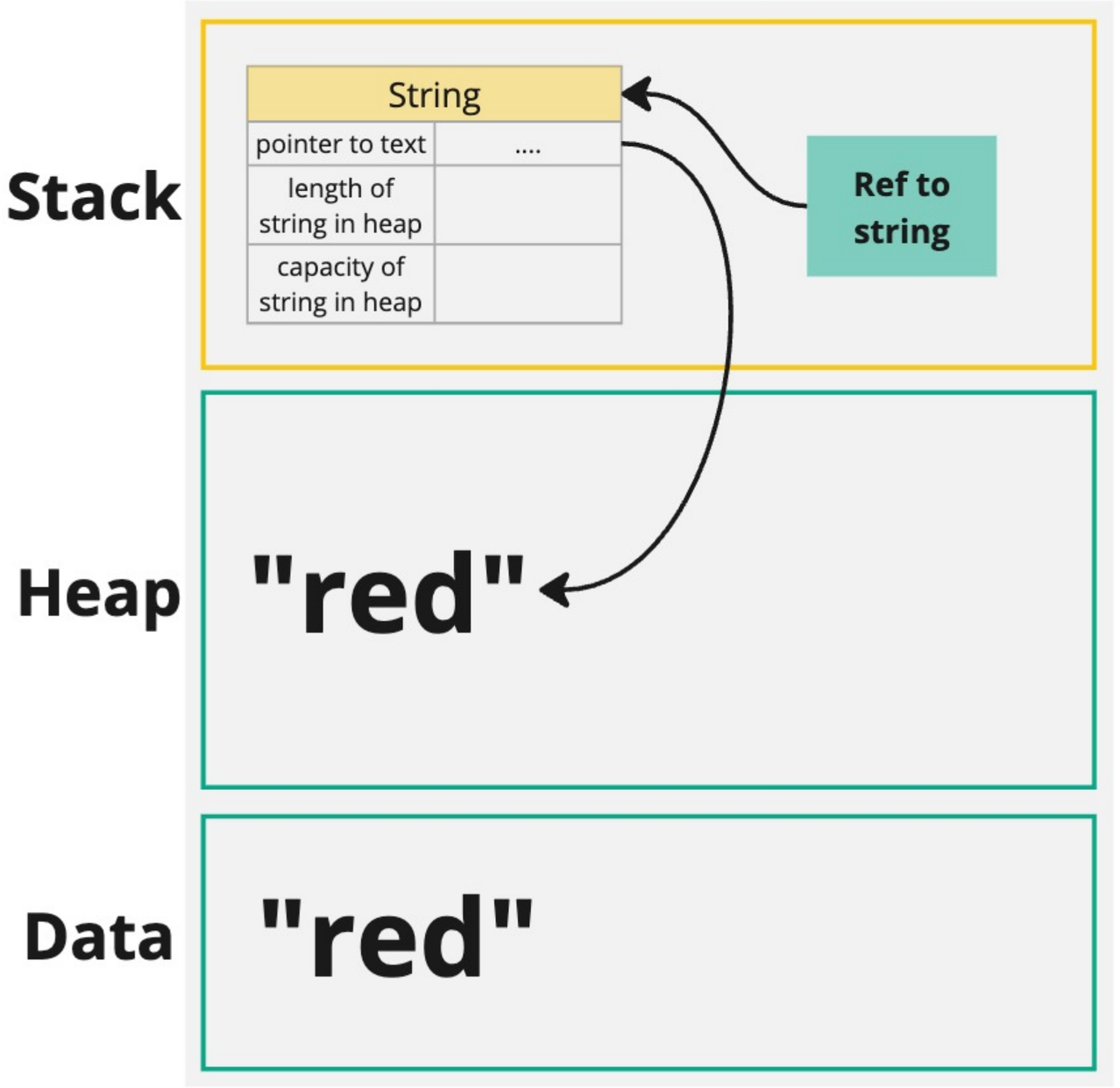
```
let name = "me";
```



Why is there &String and &str?

Both provide a read-only reference to text data

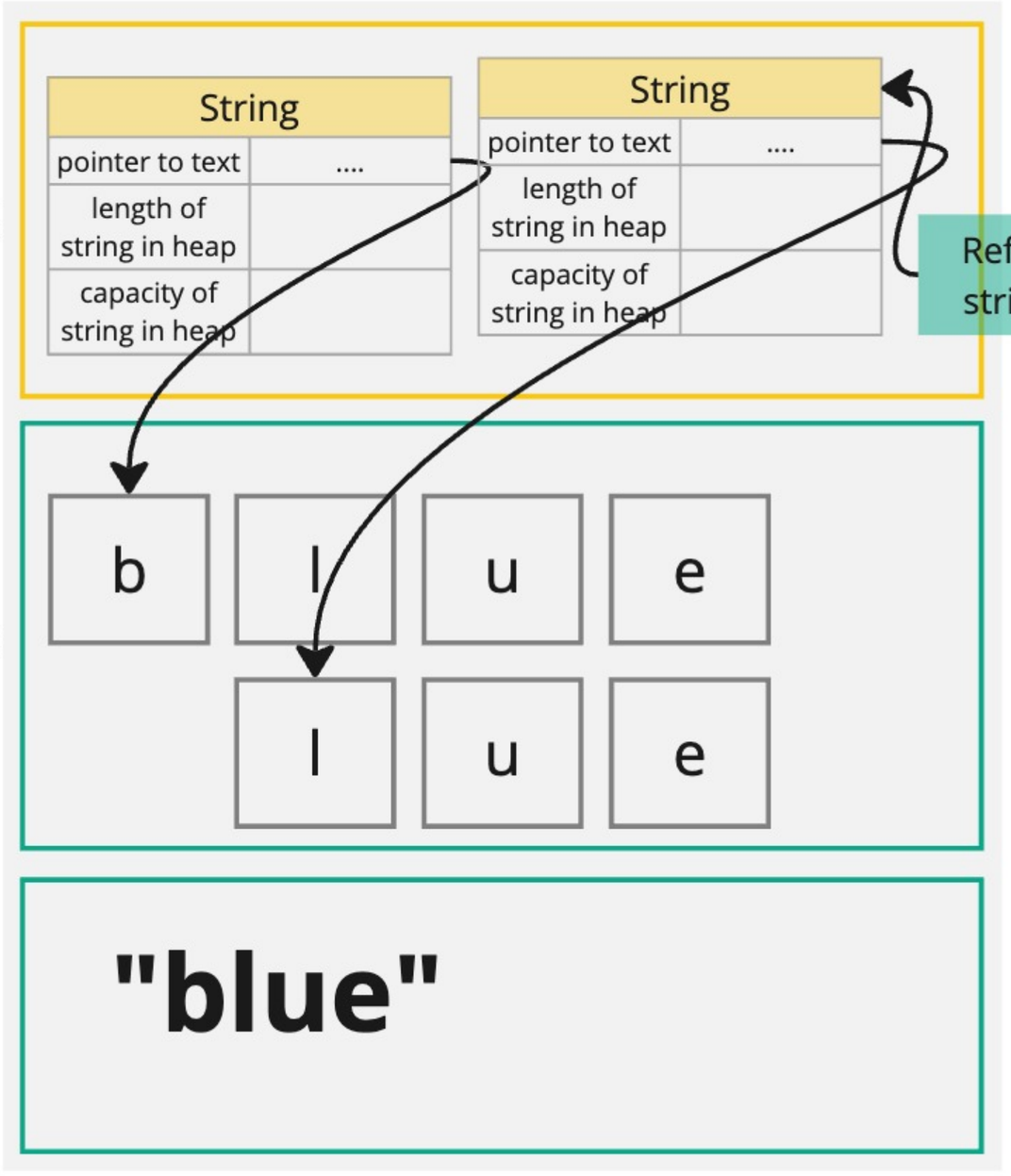
Why is there &String and &str? Reason #1




```
let color = String::from("red");
let color_ref = &color;
```

&str lets you refer to text in the data segment without a heap allocation

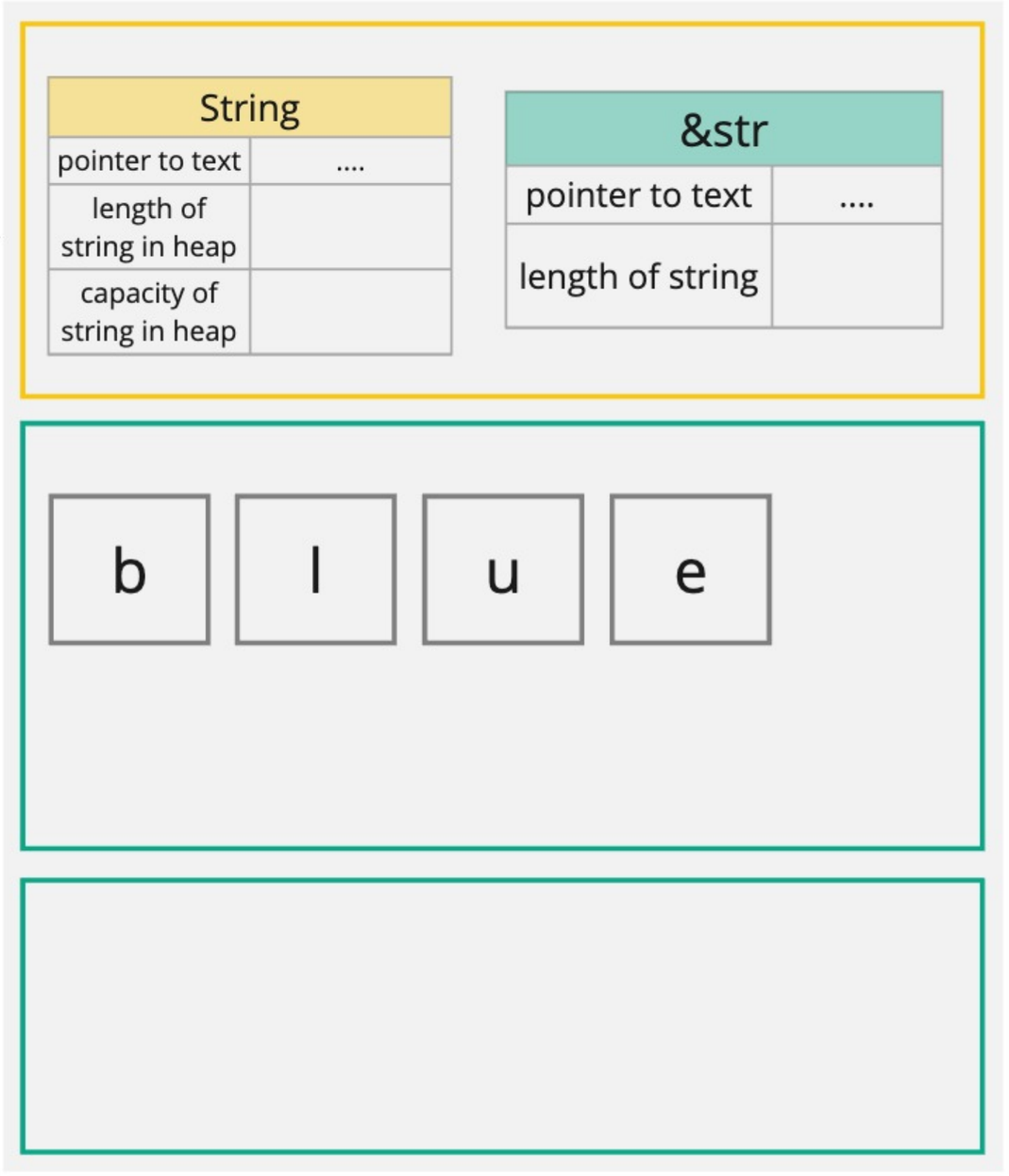
Why is there &String and &str? Reason #2




```
let color = String::from("blue");
let portion = String::from(
    color.skip(1)
)
let portion_ref = &portion;

// I sitll want to get a read only
    ref to the characters "lue"
```

&str lets you 'slice' (take a portion) of text that is already on the heap

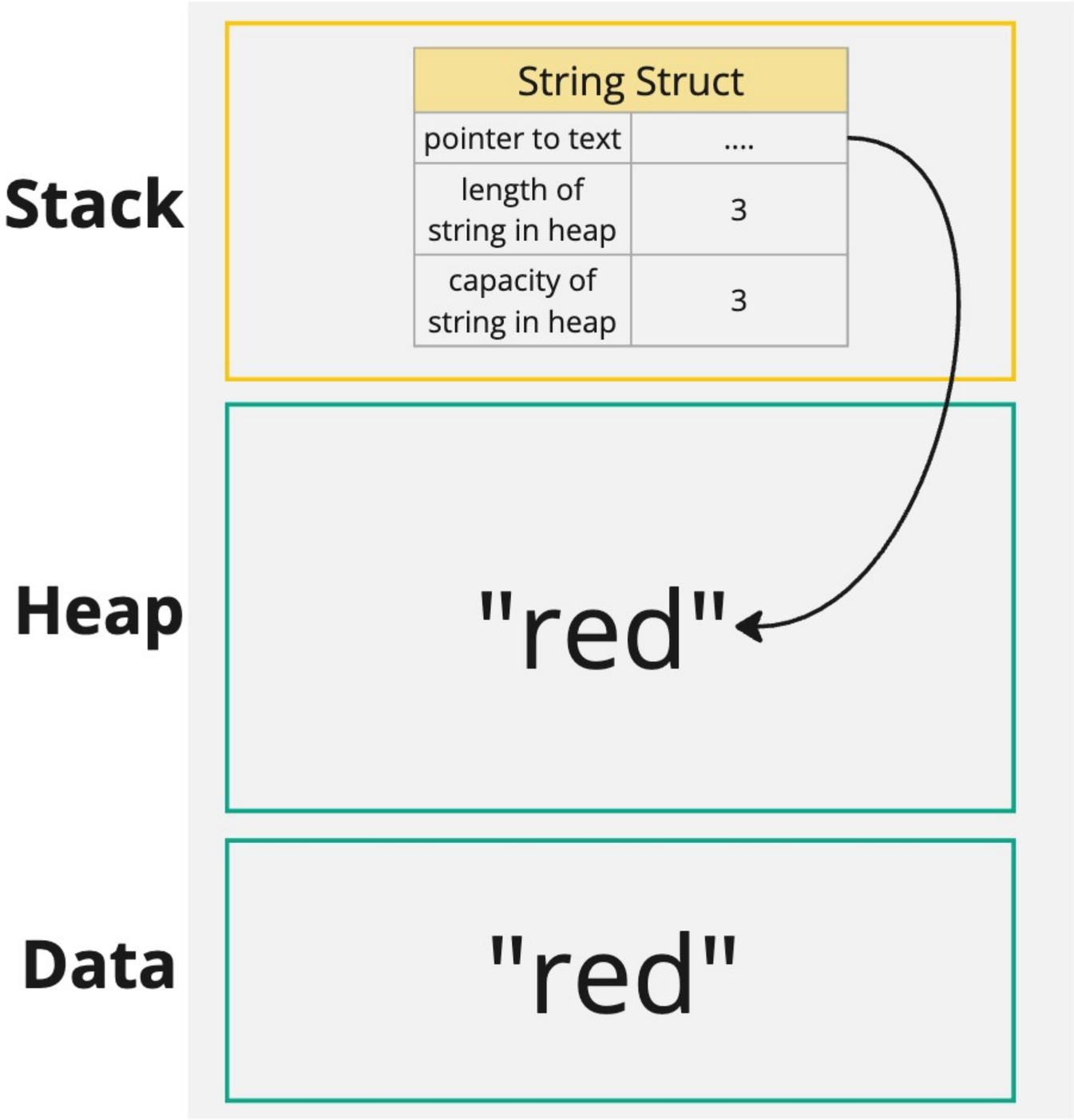
All the normal ownership rules apply!




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

    color_ref
}
```

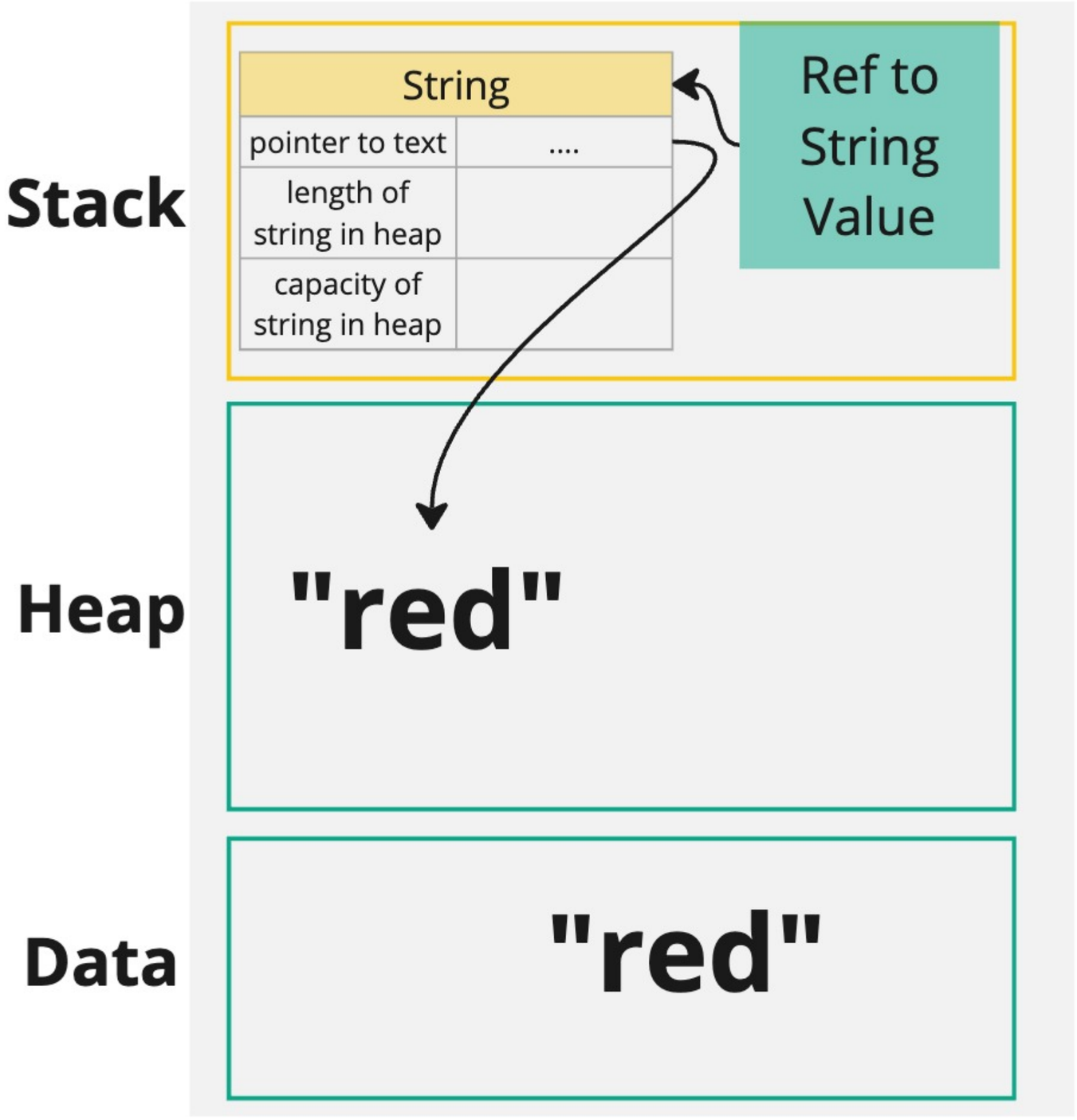
String

```
let color = String::from("red")
```



Use anytime we want ownership of text

Use anytime we want text that can grow or shrink



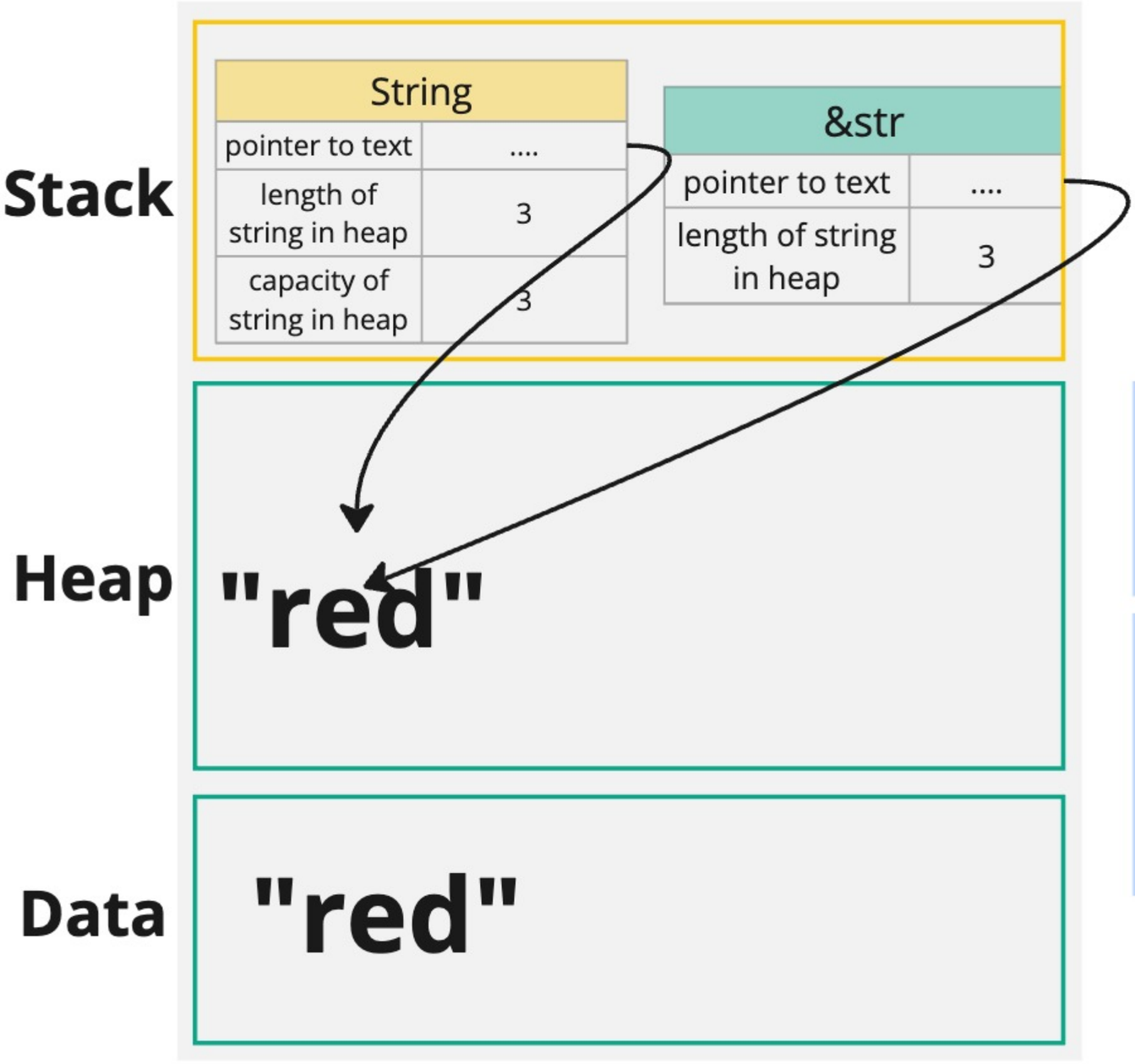
&String

```
let color = String::from("red")
let color_ref = &color
```

Rarely used!

Rust will automatically turn &String into &str for you

&str




```
let color = String::from("red");
let c = color.as_str();
```

Use anytime you don't want to take ownership of text

Use anytime you want to refer to a ***portion*** of a string owned by something else

Summary

Name	When to use	Uses memory in...	Notes
String	When you want to take ownership of text data. When you have a string that might grow or shrink.	Stack and Heap	
&String	Usually never	Stack	Rust automatically turns &String into a &str for you.
&str	When you want to read all or a portion of some text owned by something else.	Stack	Refers directly to heap-allocated or data-allocated text

```
function extract_errors(log) -> list of strings:
    split log by newline characters into lines

    initialize an empty list called result

    for each line in lines:
        if line starts with "ERROR":
            add the line to result list

    return result list
```

Pseudocode for what we want to do

This works

```
fn main() {
    match fs::read_to_string("logs.txt") {
        Ok(text_that_was_read) => {
            let error_logs = extract_errors(
                text_that_was_read.as_str()
            );
            println!("{:#?}", error_logs);
        }
        Err(why_this_failed) => {
            println!("{}", why_this_failed)
        }
    }
}
```

This results in an error

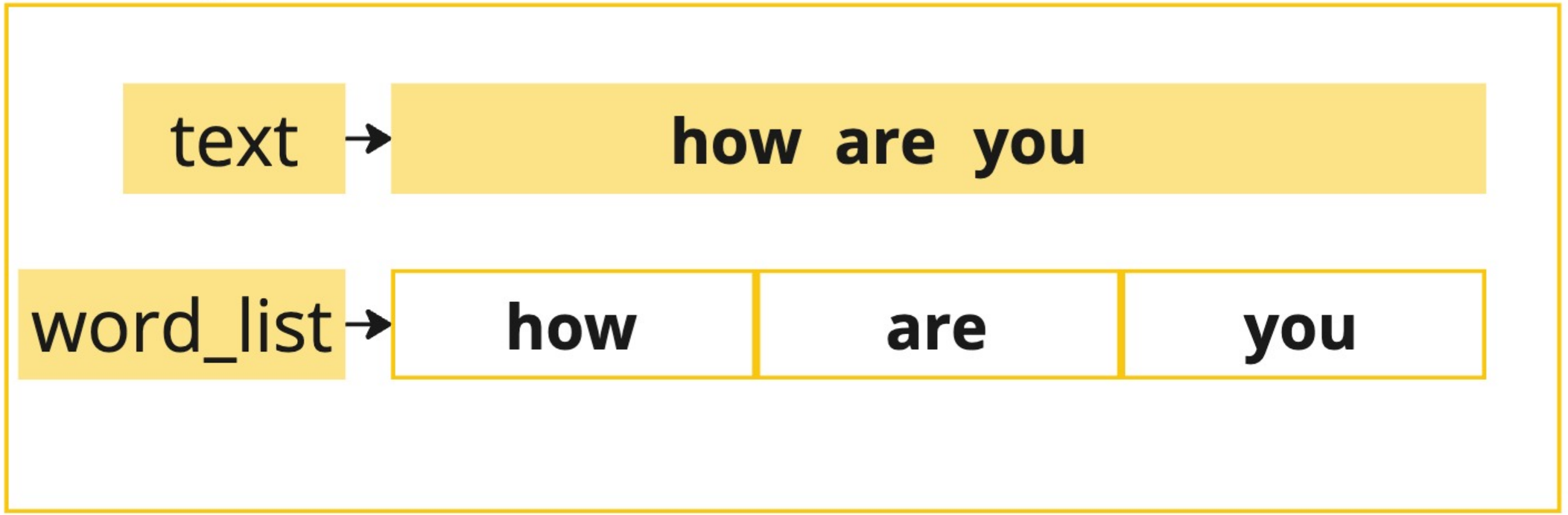
```
fn main() {
    let mut error_logs = vec![];

    match fs::read_to_string("logs.txt") {
        Ok(text_that_was_read) => {
            error_logs = extract_errors(
                text_that_was_read.as_str()
            );
        }
        Err(why_this_failed) => {
            println!("{}", why_this_failed)
        }
    }

    println!("{:#?}", error_logs);
}
```

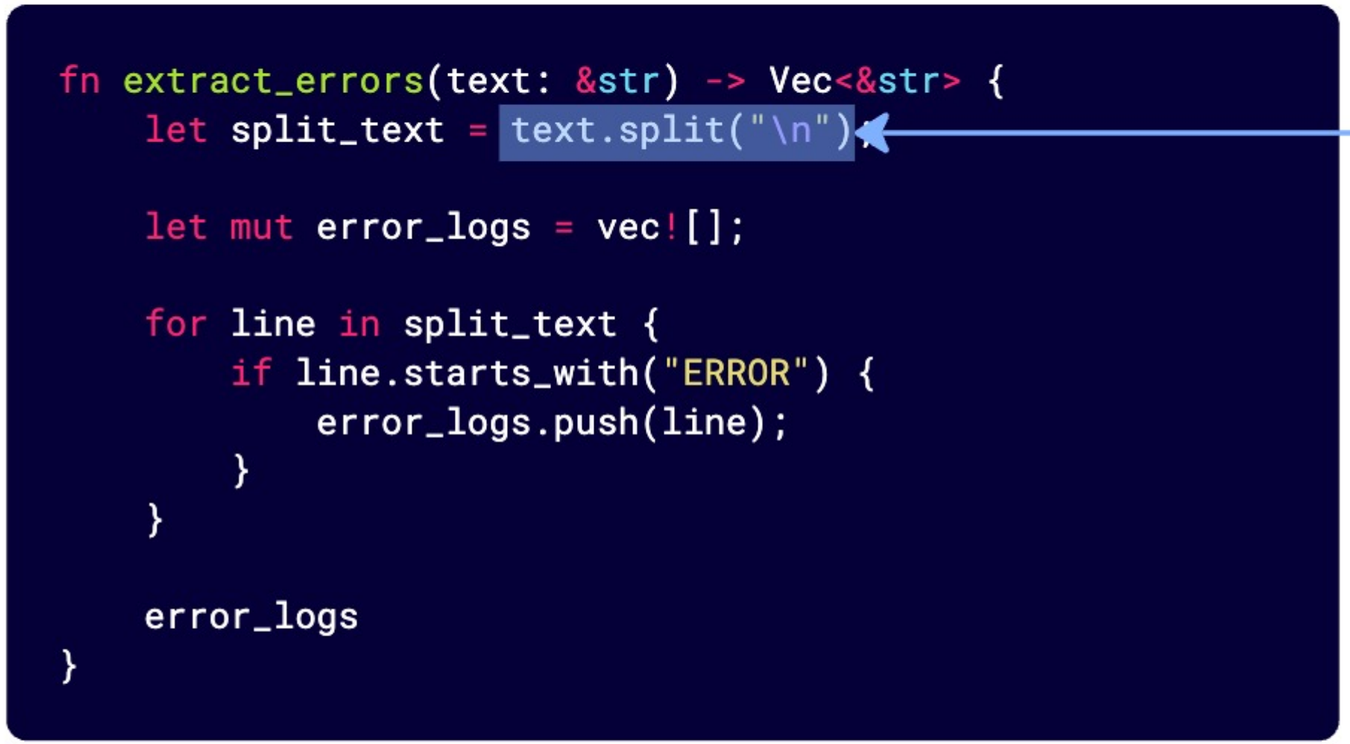
Do we really need to figure out why?

Python

```
text = "how are you"
word_list = text.split(" ")
```

Memory





Split returns Vec<&str>

Will this have any affect on our code?

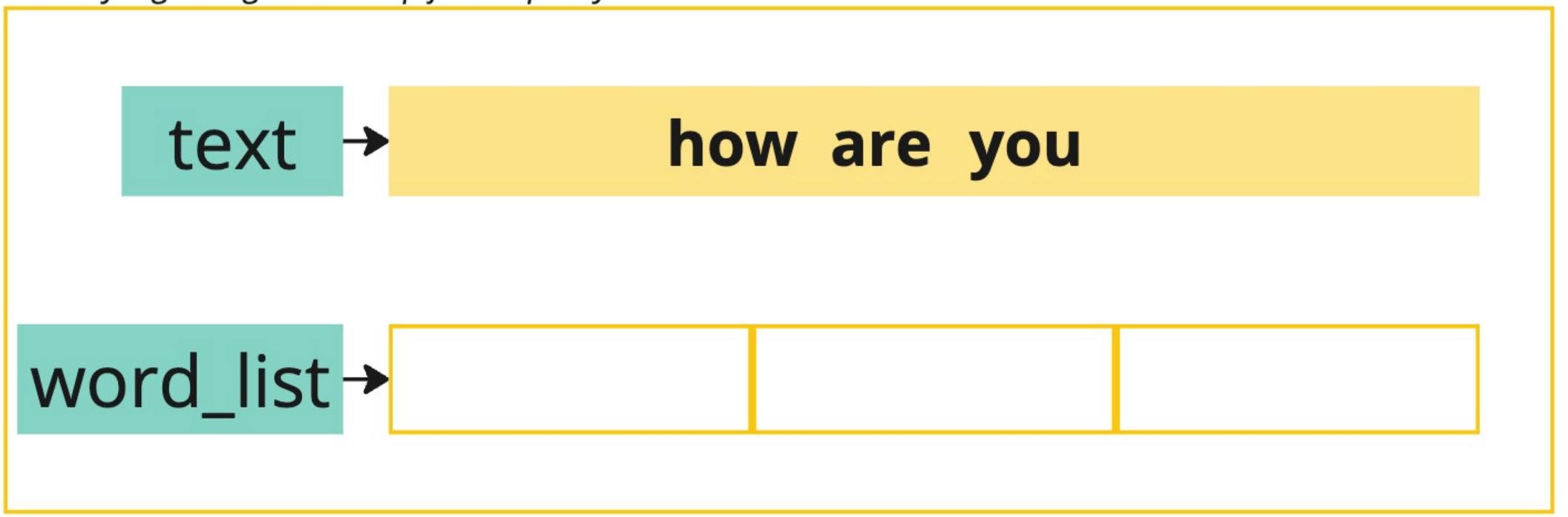
↓

YES!

Rust

```
let text = "how are you"
let word_list = text.split(" ")
```

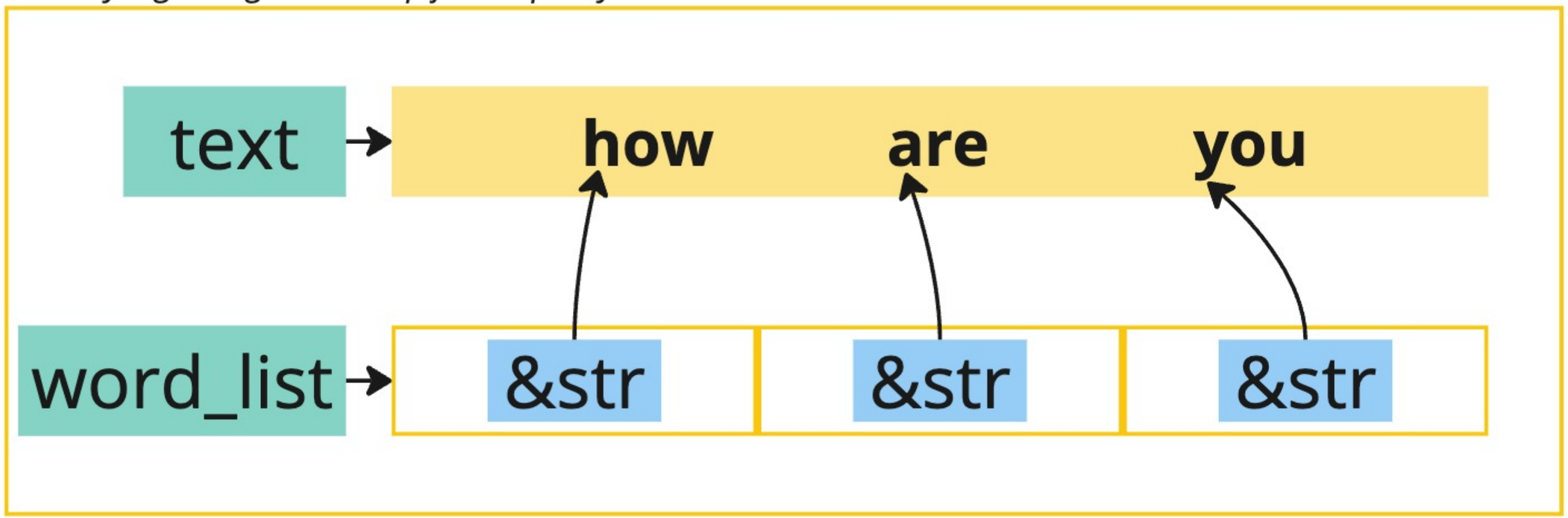
Memory - Ignoring stack/heap for simplicity

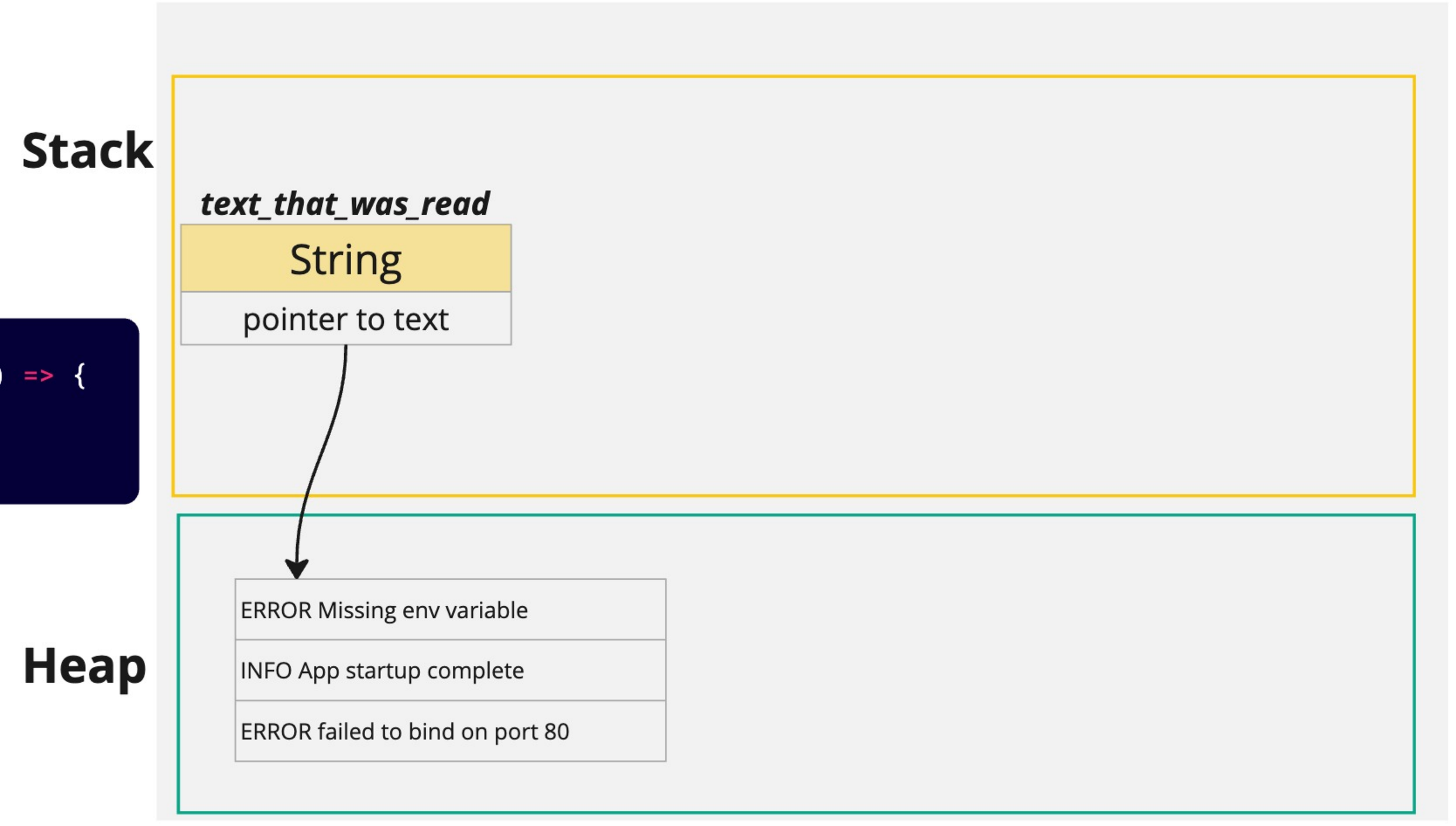


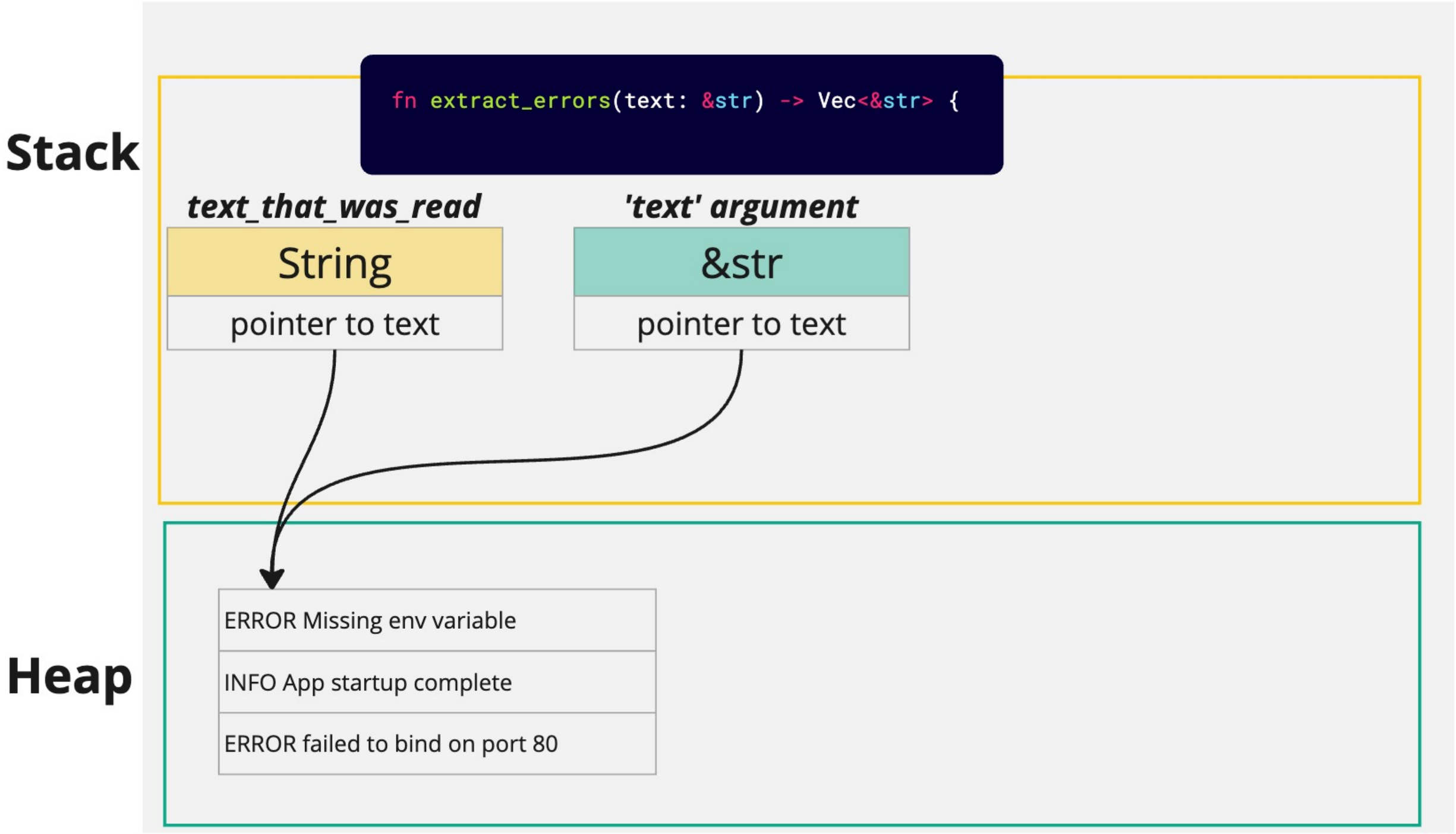
Rust

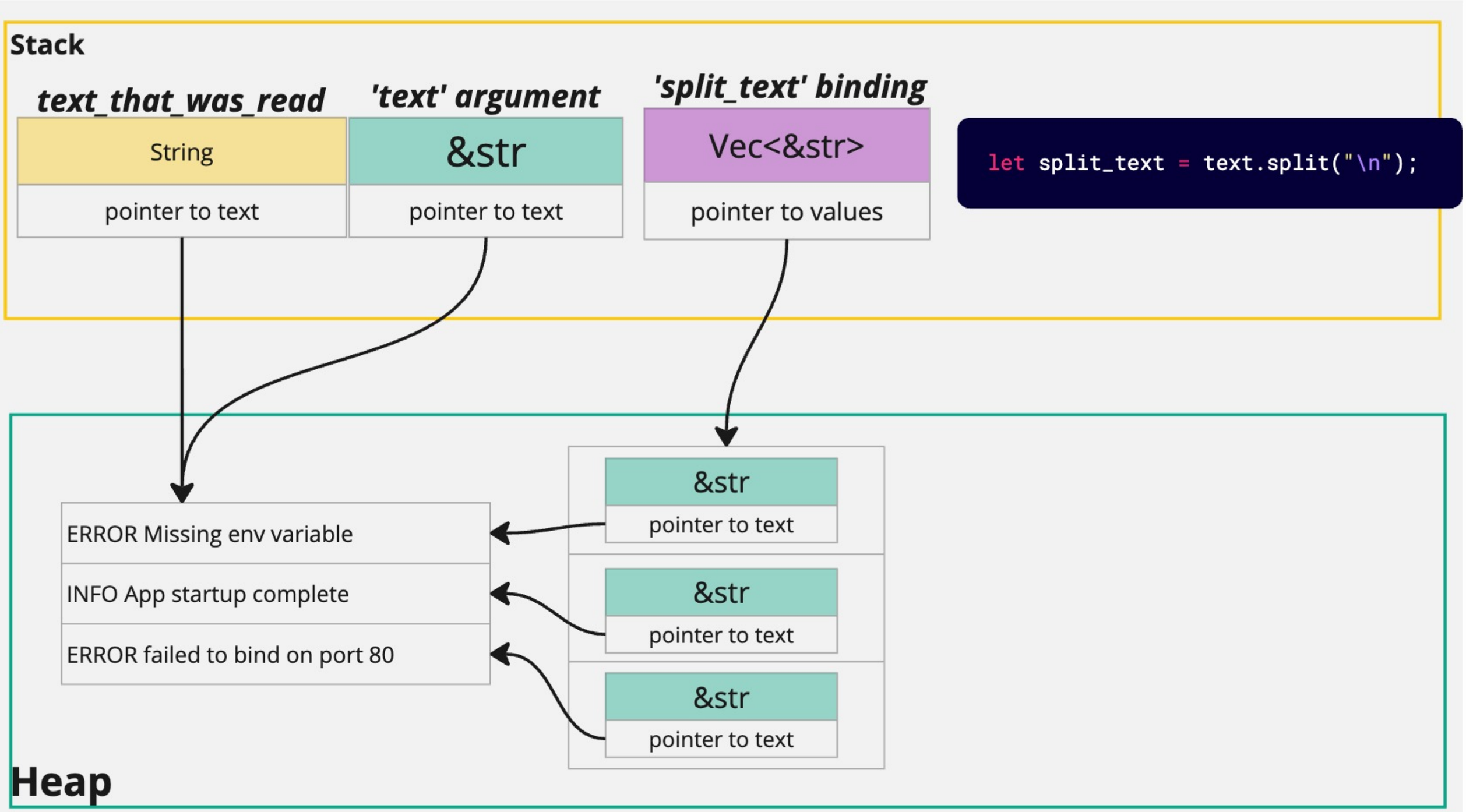
```
let text = "how are you"
let word_list = text.split(" ")
```

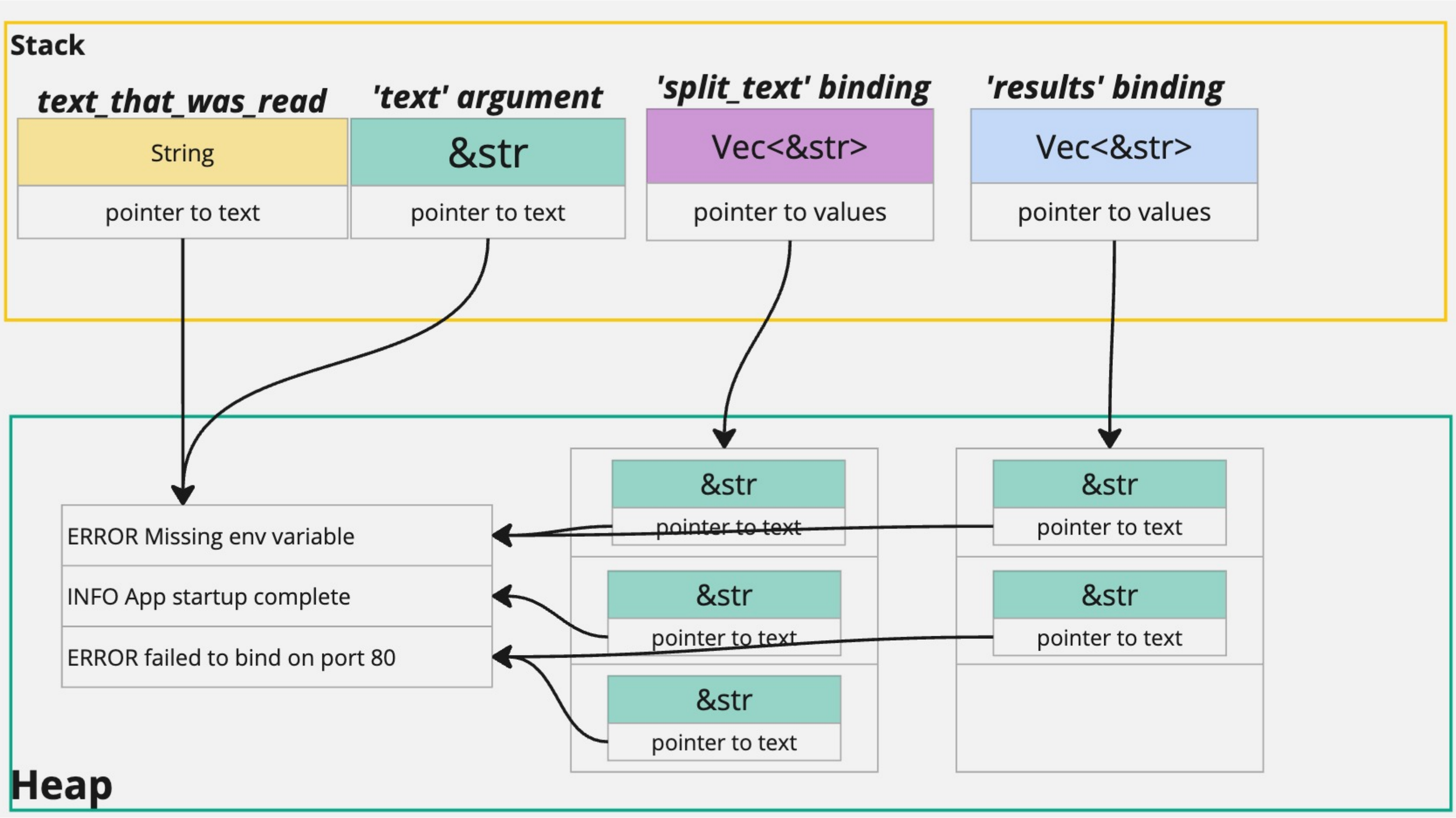
Memory - Ignoring stack/heap for simplicity

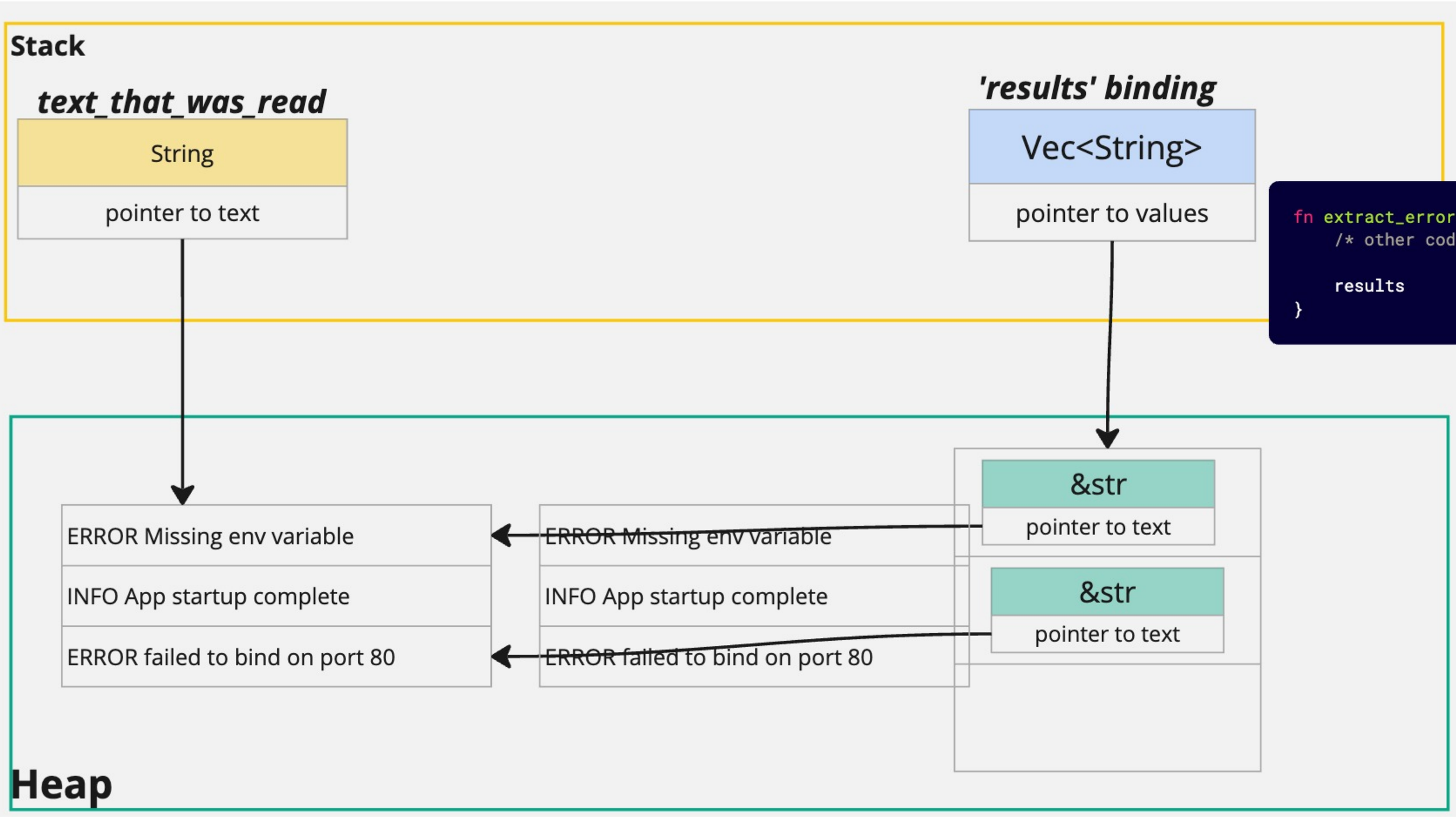


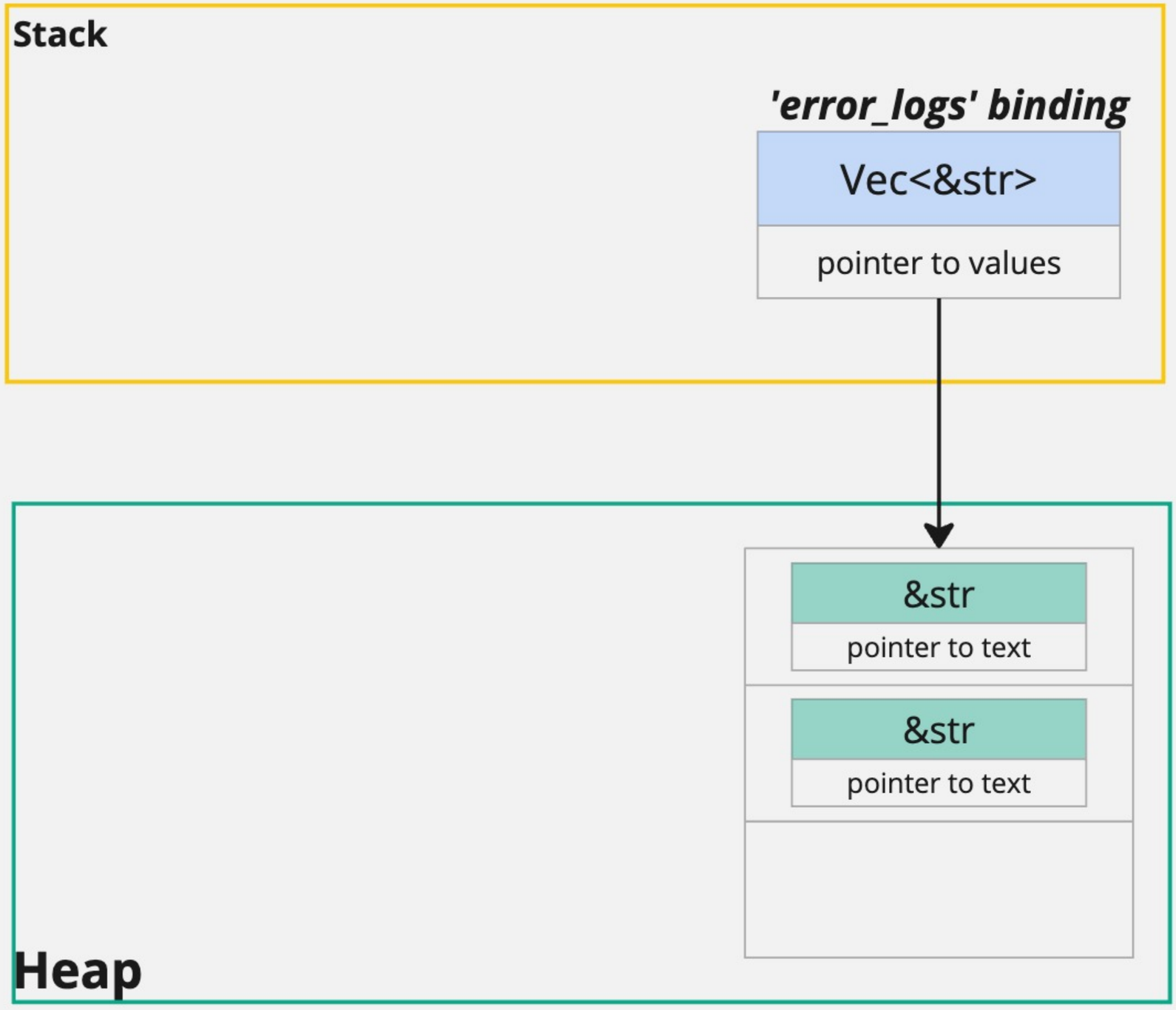

Stack
text_that_was_read
String
pointer to text
=> {
Heap
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80


Stack
fn extract_errors(text: &str) -> Vec<&str> {
text_that_was_read
String
pointer to text
'text' argument
&str
pointer to text
Heap
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80


Stack
text_that_was_read
String
pointer to text
'text' argument
&str
pointer to text
'split_text' binding
Vec<&str>
pointer to values
let split_text = text.split("\n");
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80
&str
pointer to text
&str
pointer to text
&str
pointer to text
Heap


Stack
text_that_was_read
String
pointer to text
'text' argument
&str
pointer to text
'split_text' binding
Vec<&str>
pointer to values
'results' binding
Vec<&str>
pointer to values
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80
&str
pointer to text
&str
pointer to text
&str
pointer to text
&str
pointer to text
&str
pointer to text
Heap


Stack
text_that_was_read
String
pointer to text
'results' binding
Vec<String>
pointer to values
fn extract_errors(
/* other code.
results
}
&str
pointer to text
&str
pointer to text
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80
ERROR Missing env variable
INFO App startup complete
ERROR failed to bind on port 80
Heap



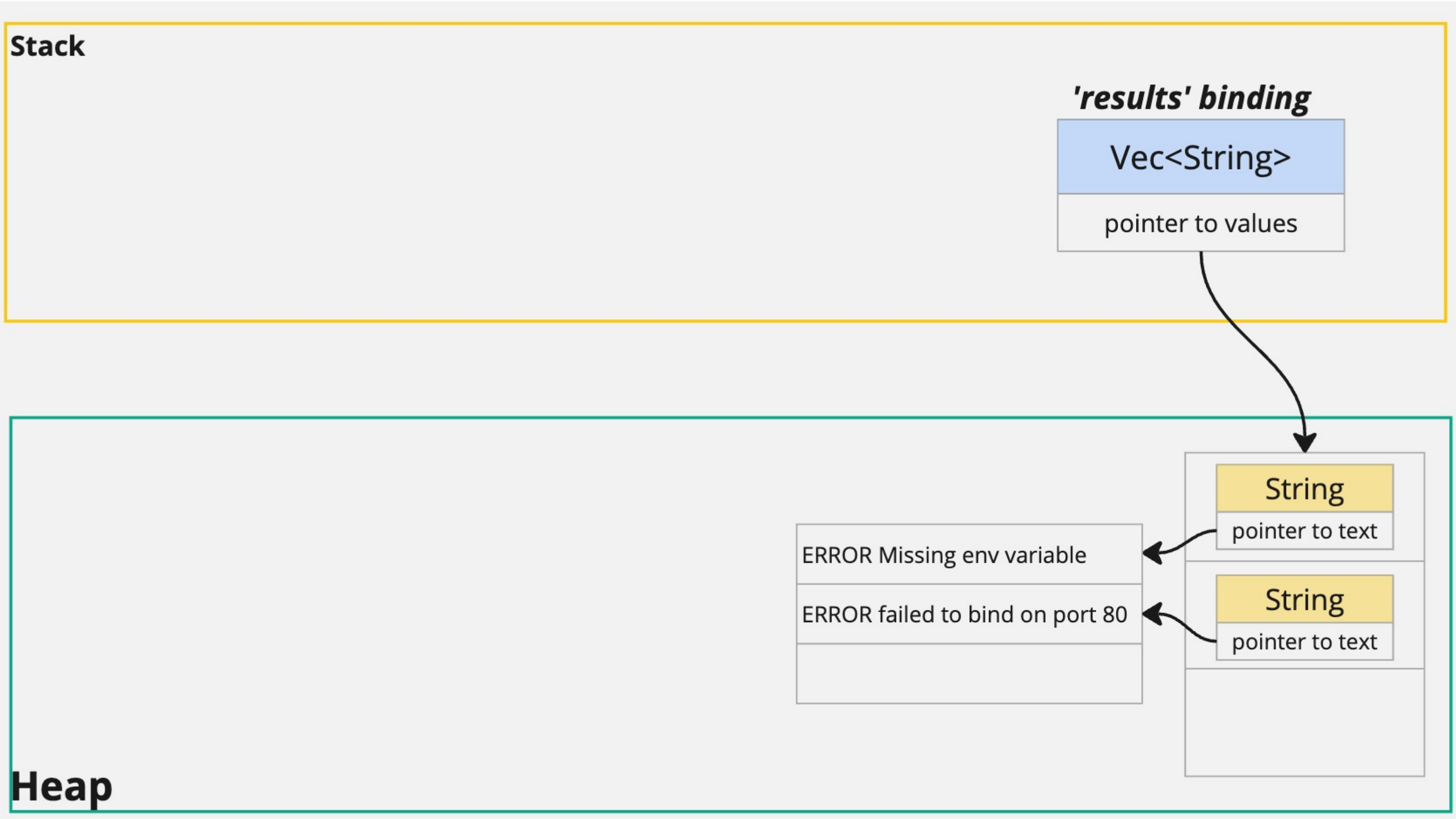

```rust
fn main() {
    let mut error_logs = vec![];

    match fs::read_to_string("logs
        Ok(text_that_was_read) =>
            error_logs = extract_e
                text_that_was_read
            );
        } // text_that_was_Read is
            go out of scope!!!
        // Its value will be dropp
        /* err case */
    }

    println!("{:#?}", error_logs);
}
```

Stack

Heap

Code we had that was working

```
fn main() {
    match fs::read_to_string("logs.txt") {
        Ok(text_that_was_read) => {
            let error_logs = extract_errors
                text_that_was_read.as_str()
            );

            println!("{:#?}", error_logs);
        } // error_logs go out of scope
        // text_that_was_read go out of
            scope
        /* err case */
    }
}
```

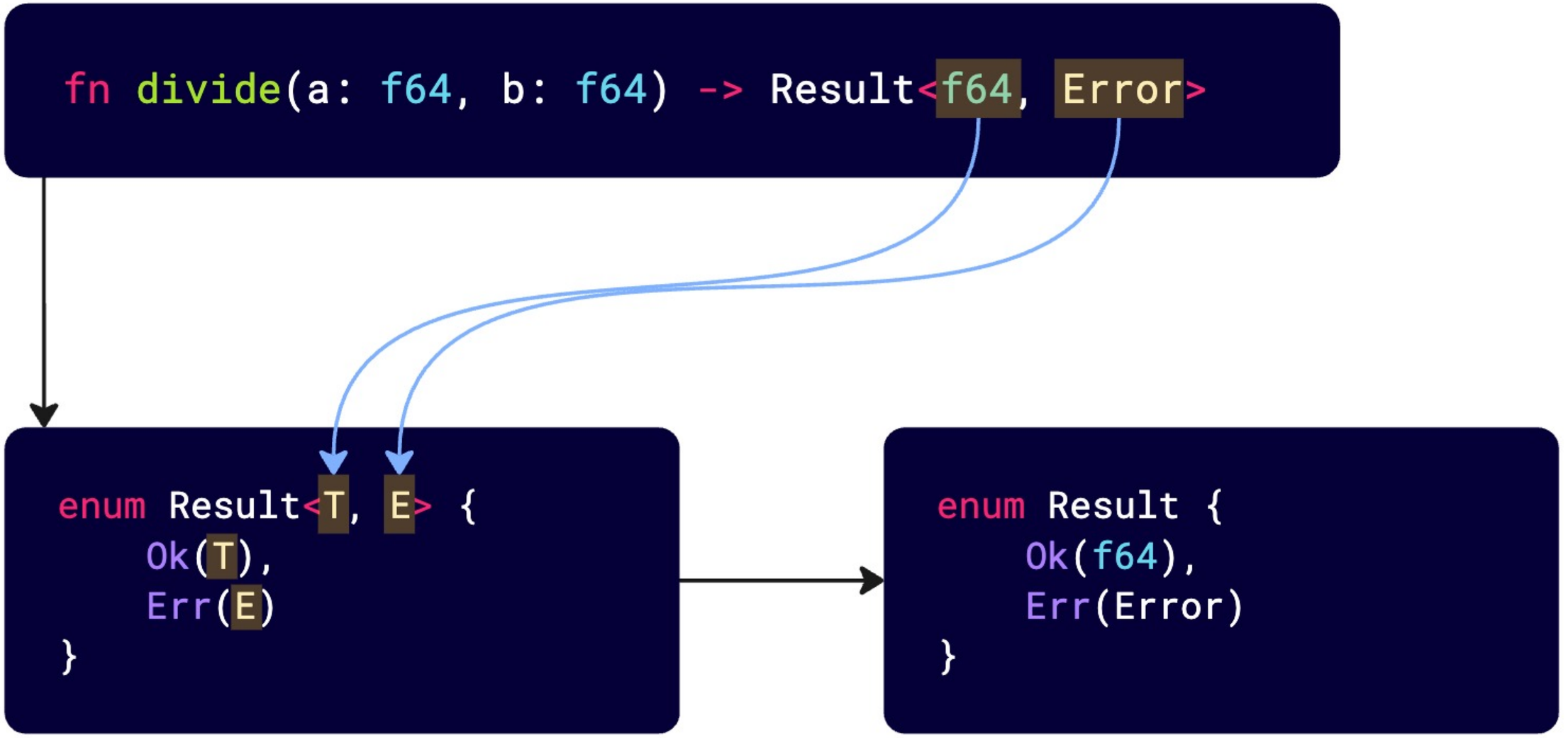

Stack
'results' binding
Vec<String>
pointer to values
String
pointer to text
String
pointer to text
ERROR Missing env variable
ERROR failed to bind on port 80
Heap

If our function receives some text and we need to return text, **should we always return a String?**

Depends!

Returning a String required extra allocations on the heap

We would have been fine returning &str if we didn't expect it to have to outlive the input text



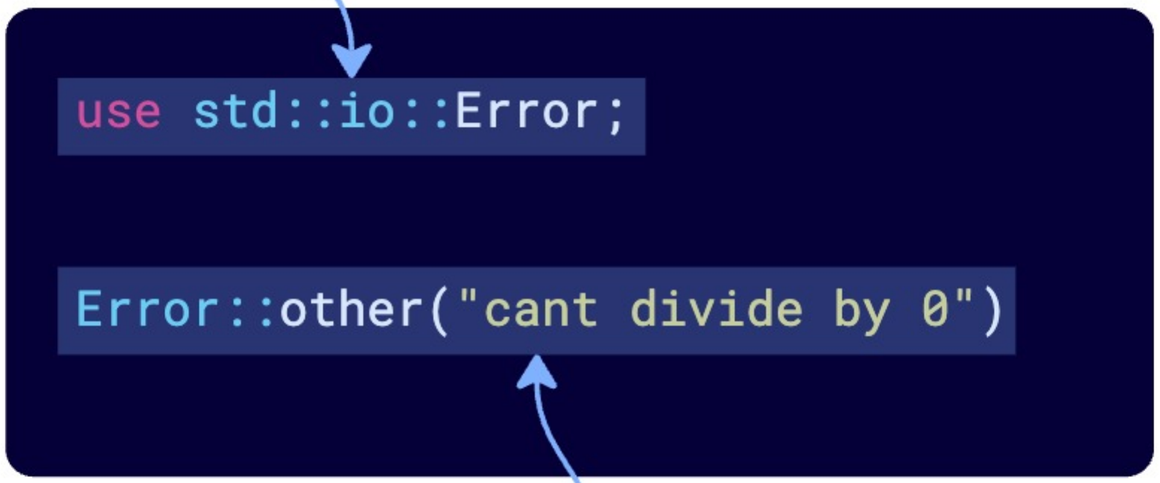
This is the actual definition of 'Result' in the Rust source code

Remember, enums are *kind of* like a shortcut for defining multiple structs

```
fn divide(a:f64, b:f64) -> Result<f64, Error> {
    if b == 0.0 {
        Err(Error::other("Oops..."))
    }

    Ok(1.33);
}
```

```
struct OkResult {
    value_that_was_calculated: f64
}

struct ErrResult {
    the_error_that_occured: Error
}

fn divide(a:f64, b:f64) -> OkResult or ErrResult {
    if b == 0.0 {
        ErrResult {
            the_error_that_occured: Error::other("Oops...")
        }
    }

    OkResult {
        value_that_was_calculated: 1.0
    }
}
```

Imports a struct defined in the std lib.
Used to represent an error



Creates an instance of the Error struct

```rust
use std::str::Utf8Error
```

```rust
use std::string::FromUtf8Error
```

```rust
use std::num::ParseIntError
use std::num::ParseFloatError
use std::num::TryFromIntError
```

```rust
use std::thread::JoinError
```

```rust
use std::io::Error
```

Many modules in the std lib have their own custom error types

You can also create your own custom types of errors

There isn't really a general-purpose catch-all type of error

Javascript has 'Error'
Python has 'Exception'

```rust
fn divide(a: f64, b: f64) -> Result<f64, Error> {
    if b == 0.0 {
        Err(Error::other("cant divide by 0"))
    } else {
        Ok(a / b)
    }
}
```

Value we put in the 'Ok' is the result of the successful operation

```rust
fn divide(a: f64, b: f64) -> Result<f64, Error> {
    if b == 0.0 {
        Err(Error::other("cant divide by 0"))
    } else {
        Ok(a / b)
    }
}
```

Value we put in the 'Ok' is the result of the successful operation

What should we put here if we have a successful operation that doesn't really give us any value?

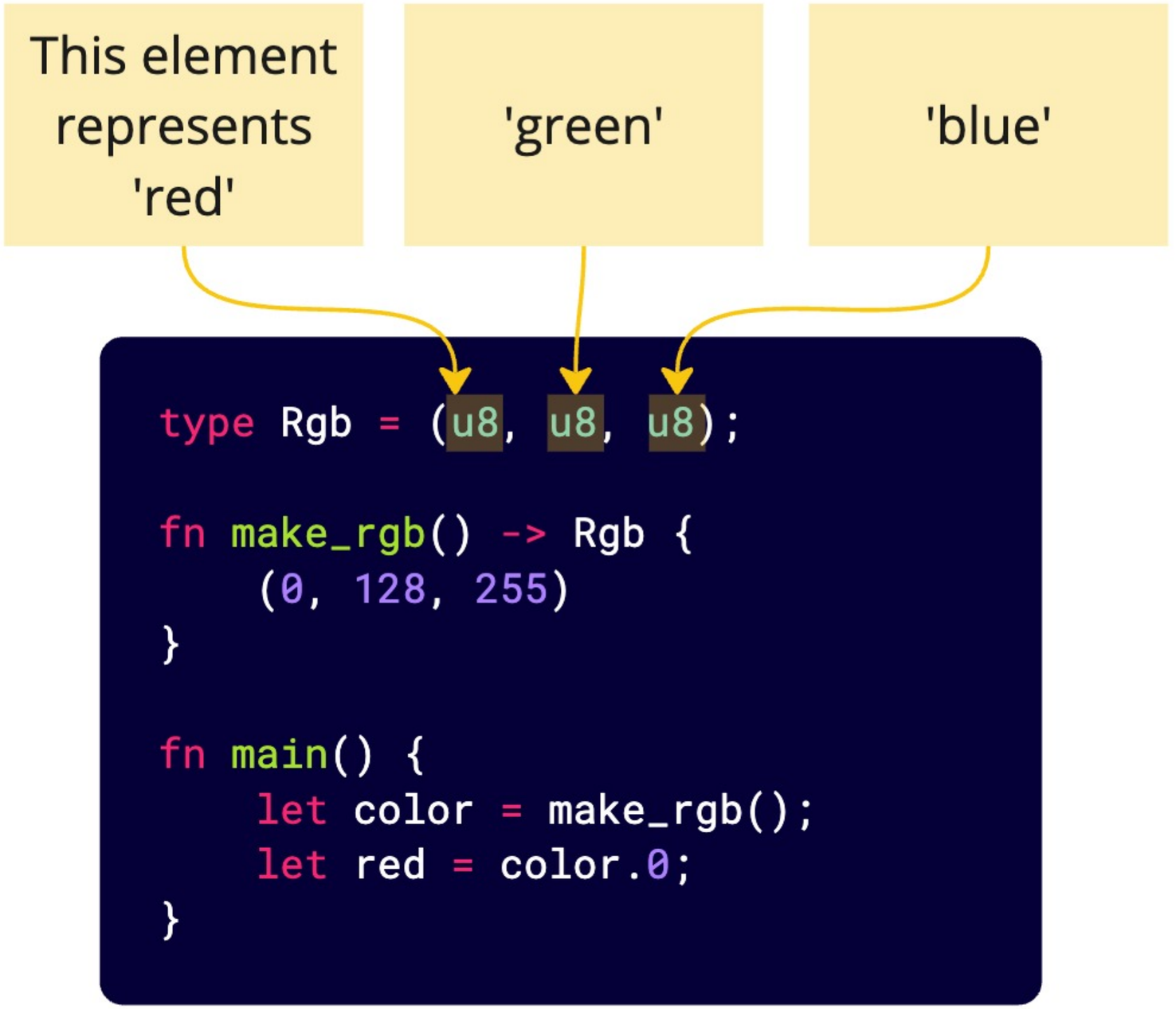
- Writing data to a file
- Removing a directory
- Permission check
- Validating a string

Nothing worth returning in 'Ok' variant?

Use an empty tuple in the Ok

```
fn validate_email(email: String) -> Result<(), Error> {
    if email.contains("@") {
        Ok(())
    } else {
        Err(Error::other("emails must have an @"))
    }
}
```

```
struct Rgb {
    red: u8,
    blue: u8,
    green: u8,
}

fn make_rgb() -> Rgb {
    Rgb {
        red: 0,
        blue: 128,
        green: 255,
    }
}

fn main() {
    let color = make_rgb();
    let red = color.red;
}
```


This element represents 'red'
'green'
'blue'
type Rgb = (u8, u8, u8);
fn make_rgb() -> Rgb {
(0, 128, 255)
}
fn main() {
let color = make_rgb();
let red = color.0;
}