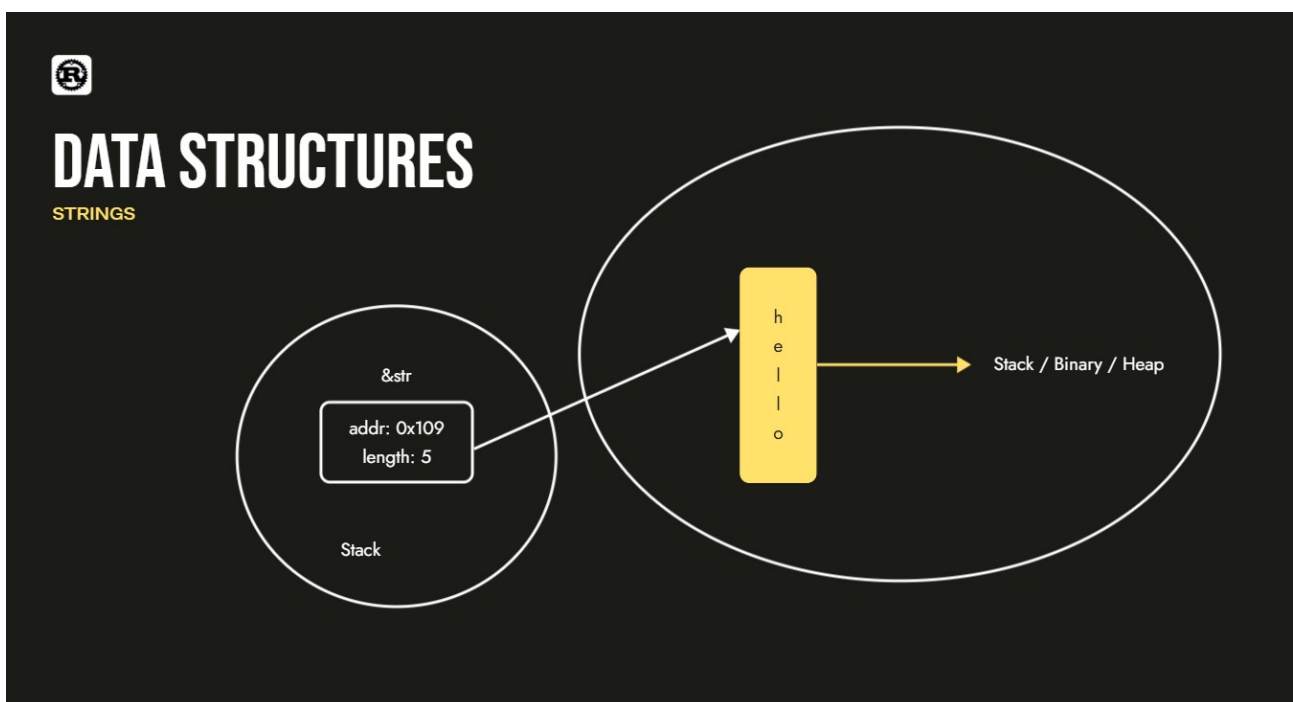


# Strings

When working with strings in Rust, there are two primary types:

- **String Slice (&str)**
- **String (String)**

These differ in terms of **storage** and **memory management** in the **stack** and **heap**. Let's analyze them using your example, "hello".



## 1. String Slice (&str)

A **string slice** is a reference to a part of a string. In Rust, string literals (like "hello") are stored in the binary's **read-only data section**.

### Storage in Memory

- **Stack:** Stores the **reference (&str)**, which consists of:
  - A pointer to the string data (which is in the binary's read-only section)
  - The length of the string (5 in this case)
- **Binary (Read-Only Memory):** Stores the actual characters **"hello"**.

### Example:

```
let s: &str = "hello";
```

## Memory Representation

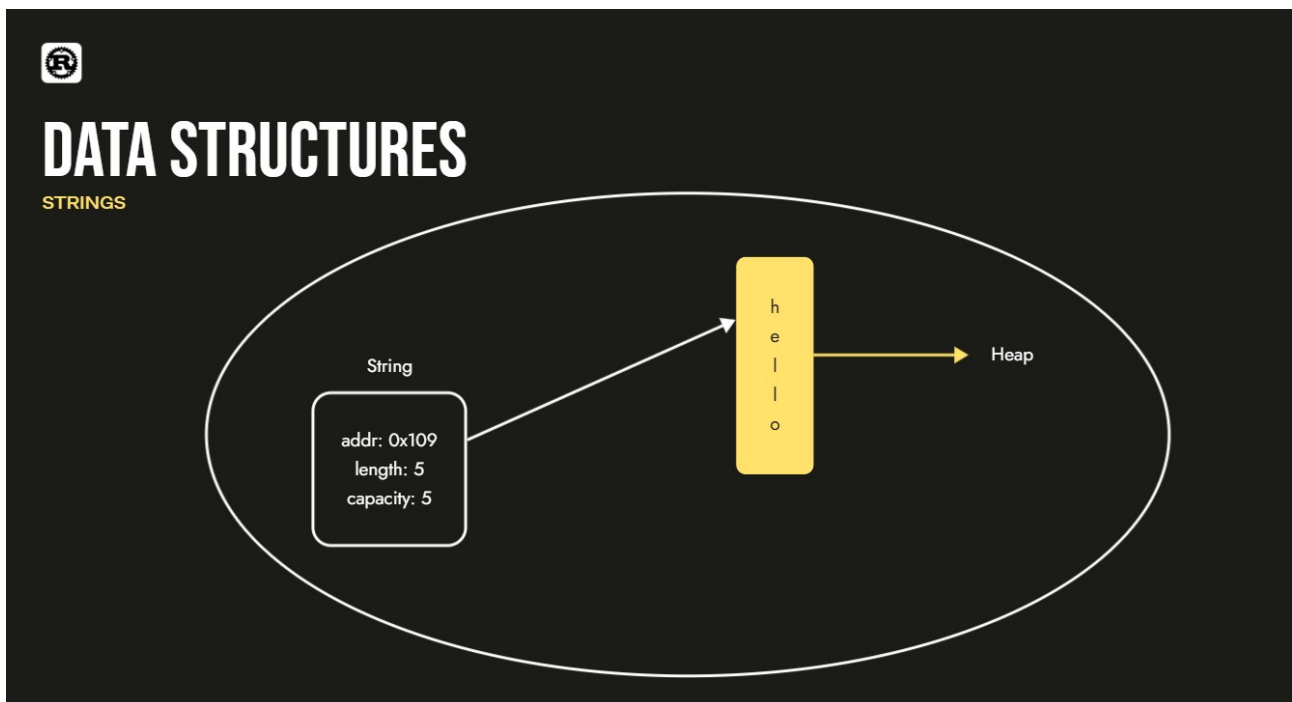
Memory	Content
Stack	Pointer → "hello" (read-only section)
Binary (Read-Only Data Section)	"hello" (not modifiable)

### Advantages:

- No heap allocation.
- Efficient, as it only stores a reference and a length.

### Disadvantages:

- Immutable.
- Lifetime is tied to the original data.



## 2. String (String)

A **String** is a **heap-allocated, growable string**.

### Storage in Memory

- **Stack:** Stores the **String struct**, which consists of:
  - A **pointer** to the heap-allocated "hello"
  - The **length** (5)
  - The **capacity** (typically  $\geq 5$ )
- **Heap:** Stores the actual string **"hello"**, allocated dynamically.

### Example:

```
let s: String = String::from("hello");
```

## Memory Representation

Memory	Content
Stack	Pointer → Heap "hello"
Heap	"hello" (modifiable, stored dynamically)

### Advantages:

- Growable (can be modified).
- Heap allocation allows dynamic size changes.

### Disadvantages:

- Heap allocation makes it slightly slower than `&str`.
  - Needs explicit memory management (ownership, borrowing).
- 

## Comparison Summary

Feature	<code>&amp;str</code> (String Slice)	<code>String</code> (Owned String)
Stored in	Stack (pointer) + Read-Only Binary	Stack (pointer, len, cap) + Heap (data)
Mutability	Immutable	Mutable
Size Known at Compile Time?	Yes	No
Heap Allocation?	No	Yes
Performance	Faster	Slightly slower (due to heap allocation)

## String Methods

`String` is a growable, heap-allocated data structure used to store and manipulate text.

### 1. Creating Strings

```
let mut s = String::new();
let s = String::from("hello");
let s = "hello".to_string();
```

### 2. Appending to a String

```
let mut s = String::from("hello");
s.push_str(" world"); // Appends a string slice
s.push('!');          // Appends a character
```

### 3. Concatenation

```
let s1 = String::from("hello");
let s2 = String::from(" world");
let s3 = s1 + &s2; // Note: s1 is moved here and can no longer be used
let s4 = format!("{}", s3, " again");
```

### 4. Length and Capacity

```
let s = String::from("hello");
println!("Length: {}", s.len());
println!("Capacity: {}", s.capacity());
```

## 5. Checking if Empty

```
let s = String::new();
println!("Is empty: {}", s.is_empty());
```

## 6. Accessing Characters

```
let s = String::from("hello");
for c in s.chars() {
    println!("{}", c);
}
```

## 7. Accessing Bytes

```
let s = String::from("hello");
for b in s.bytes() {
    println!("{}", b);
}
```

## 8. Substring Extraction

```
let s = String::from("hello");
let slice = &s[0..2]; // "he"
```

## 9. Replacing and Trimming

```
let s = String::from("hello world");
let s = s.replace("world", "Rust");
let s = s.trim();
```

## 10. Splitting Strings

```
let s = String::from("hello world");
for word in s.split_whitespace() {
    println!("{}", word);
}
```

# String Slice (&str) Methods

String slices are immutable views into a string.

### 1. Length

```
let s = "hello";
println!("Length: {}", s.len());
```

### 2. Checking if Empty

```
let s = "";
println!("Is empty: {}", s.is_empty());
```

### 3. Accessing Characters

```
let s = "hello";
for c in s.chars() {
```

```
    println!("{}", c);
}
```

#### 4. Accessing Bytes

```
let s = "hello";
for b in s.bytes() {
    println!("{}", b);
}
```

#### 5. Substring Extraction

```
let s = "hello";
let slice = &s[0..2]; // "he"
```

#### 6. Finding Substrings

```
let s = "hello world";
if let Some(pos) = s.find("world") {
    println!("Found 'world' at position: {}", pos);
}
```

#### 7. Checking Prefixes and Suffixes

```
let s = "hello world";
println!("Starts with 'hello': {}", s.starts_with("hello"));
println!("Ends with 'world': {}", s.ends_with("world"));
```

#### 8. Replacing and Trimming

```
let s = " hello world ";
let s = s.trim();
let s = s.replace("world", "Rust");
```

#### 9. Splitting Strings

```
let s = "hello world";
for word in s.split_whitespace() {
    println!("{}", word);
}
```

#### 10. Joining Strings

```
let words = ["hello", "world"];
let s = words.join(" ");
println!("{}", s); // Output: "hello world"
```

### Examples with More Context

#### Concatenation and Formatting

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = String::from(", world!");
    let s3 = s1 + &s2; // s1 is moved here and can no longer be used
    println!("{}", s3);

    let s4 = format!("{}", s3, " How are you?");
    println!("{}", s4);
}
```

```
}
```

## Iterating Over Characters

```
fn main() {  
    let s = "hello";  
    for c in s.chars() {  
        println!("{}", c);  
    }  
  
    for b in s.bytes() {  
        println!("{}", b);  
    }  
}
```

## Splitting and Trimming

```
fn main() {  
    let s = " hello world ";  
    let trimmed = s.trim();  
    println!("Trimmed: '{}'", trimmed);  
  
    for word in trimmed.split_whitespace() {  
        println!("{}", word);  
    }  
}
```

## Using find and replace

```
fn main() {  
    let s = "hello world";  
    if let Some(pos) = s.find("world") {  
        println!("Found 'world' at position: {}", pos);  
    }  
  
    let replaced = s.replace("world", "Rust");  
    println!("Replaced: {}", replaced);  
}
```

## split Method on &str

The `split` method in Rust is used to split a string slice (`&str`) based on a delimiter. It returns an iterator over substrings.

## Basic split

Splits the string slice by a given delimiter and returns an iterator over the substrings.

```
fn main() {  
    let s = "hello world";  
    let words: Vec<&str> = s.split(' ').collect();  
    println!("{}", words); // Output: ["hello", "world"]  
}
```

```
}
```

## **split\_whitespace**

Splits the string slice by any whitespace and returns an iterator over the substrings.

```
fn main() {  
    let s = "hello    world \n Rust";  
    let words: Vec<&str> = s.split_whitespace().collect();  
    println!("{:?}", words); // Output: ["hello", "world", "Rust"]  
}
```

## **splitn**

Splits the string slice by a given delimiter but limits the number of substrings to a specified number.

```
fn main() {  
    let s = "a b c d e";  
    let parts: Vec<&str> = s.splitn(3, ' ').collect();  
    println!("{:?}", parts); // Output: ["a", "b", "c d e"]  
}
```

## **rsplit**

Splits the string slice from the end of the string (reverse split).

```
fn main() {  
    let s = "a b c d e";  
    let parts: Vec<&str> = s.rsplit(' ').collect();  
    println!("{:?}", parts); // Output: ["e", "d", "c", "b", "a"]  
}
```

## **rsplitn**

Reverse splits the string slice by a given delimiter but limits the number of substrings to a specified number.

```
fn main() {  
    let s = "a b c d e";  
    let parts: Vec<&str> = s.rsplitn(3, ' ').collect();  
    println!("{:?}", parts); // Output: ["e", "d", "c b a"]  
}
```

## **split\_terminator**

Similar to `split`, but does not produce an empty string slice if the string slice ends with the delimiter.

```
fn main() {
```

```

    let s = "hello world!";
    let words: Vec<&str> = s.split_terminator('!').collect();
    println!("{:?}", words); // Output: ["hello world"]
}

```

## Using find and replace

```

fn main() {
    let s = "hello world";
    if let Some(pos) = s.find("world") {
        println!("Found 'world' at position: {}", pos);
    }

    let replaced = s.replace("world", "Rust");
    println!("Replaced: {}", replaced);
}

```

## split Method on &str

The `split` method in Rust is used to split a string slice (`&str`) based on a delimiter. It returns an iterator over substrings.

### Basic split

Splits the string slice by a given delimiter and returns an iterator over the substrings.

```

fn main() {
    let s = "hello world";
    let words: Vec<&str> = s.split(' ').collect();
    println!("{:?}", words); // Output: ["hello", "world"]
}

```

### split\_whitespace

Splits the string slice by any whitespace and returns an iterator over the substrings.

```

fn main() {
    let s = "hello    world \n Rust";
    let words: Vec<&str> = s.split_whitespace().collect();
    println!("{:?}", words); // Output: ["hello", "world", "Rust"]
}

```

### splitn

Splits the string slice by a given delimiter but limits the number of substrings to a specified number.

```

fn main() {
    let s = "a b c d e";
    let parts: Vec<&str> = s.splitn(3, ' ').collect();
    println!("{:?}", parts); // Output: ["a", "b", "c d e"]
}

```



## rsplit

Splits the string slice from the end of the string (reverse split).

```
fn main() {
    let s = "a b c d e";
    let parts: Vec<&str> = s.rsplit(' ').collect();
    println!("{:?}", parts); // Output: ["e", "d", "c", "b", "a"]
}
```

## rsplitn

Reverse splits the string slice by a given delimiter but limits the number of substrings to a specified number.

```
fn main() {
    let s = "a b c d e";
    let parts: Vec<&str> = s.rsplitn(3, ' ').collect();
    println!("{:?}", parts); // Output: ["e", "d", "c b a"]
}
```

## split\_terminator

Similar to `split`, but does not produce an empty string slice if the string slice ends with the delimiter.

```
fn main() {
    let s = "hello world!";
    let words: Vec<&str> = s.split_terminator('!').collect();
    println!("{:?}", words); // Output: ["hello world"]
}
```

## split\_inclusive

Splits the string slice, including the part of the delimiter in each substring.

```
fn main() {
    let s = "abc1def2ghi";
    let parts: Vec<&str> = s.split_inclusive(char::is_numeric).collect();
    println!("{:?}", parts); // Output: ["abc1", "def2", "ghi"]
}
```

## Example: Using Various `split` Methods

```
fn main() {
    let s = "hello world! hello Rust!";

    // Basic split
    let words: Vec<&str> = s.split(' ').collect();
```

```

println!("{:?}", words); // ["hello", "world!", "hello", "Rust!"]

// Split by whitespace
let words_whitespace: Vec<&str> = s.split_whitespace().collect();
println!("{:?}", words_whitespace); // ["hello", "world!", "hello", "Rust!"]

// Split with limit
let parts: Vec<&str> = s.splitn(3, ' ').collect();
println!("{:?}", parts); // ["hello", "world!", "hello Rust!"]

// Reverse split
let rparts: Vec<&str> = s.rsplit(' ').collect();
println!("{:?}", rparts); // ["Rust!", "hello", "world!", "hello"]

// Reverse split with limit
let rparts_limit: Vec<&str> = s.rsplitn(3, ' ').collect();
println!("{:?}", rparts_limit); // ["Rust!", "hello", "hello world!"]

// Split terminator
let term_split: Vec<&str> = s.split_terminator('!').collect();
println!("{:?}", term_split); // ["hello world", " hello Rust"]

// Split inclusive
let inclusive_split: Vec<&str> = s.split_inclusive(' ').collect();
println!("{:?}", inclusive_split); // ["hello ", "world! ", "hello ", "Rust!"]
}

```