Strings at a glance

Rust basics · Strings at a glance

In PHP

There is one string type. `.=` appends. `strlen` counts bytes. `mb_strlen` counts characters. `$text[0]` reads the first byte, which for `héllo` happens to be `h`.

text.php

```
<?php

$text = "héllo";
$text .= " world";
echo $text, "\n";
echo strlen($text), " ", mb_strlen($text), "\n";
echo $text[0], "\n";

$name = "Ada";
echo "Hello, $name\n";
```

`strlen` prints `12`. `mb_strlen` prints `11`.

In Rust

`String` owns the text and can grow. `&str` is a borrowed view: a literal, or the type of a function parameter such as `name: &str`. `format!` builds a new `String`. `push_str` appends. Ownership and borrowing are chapter 3. This lesson only names the two types.

`.len()` counts bytes, like `strlen`. `.chars()` walks Unicode scalar values, like `mb_strlen`. `text[0]` does not compile. A string index is a range of bytes:

```
error[E0277]: the type `str` cannot be indexed by `{integer}`
 --> src/main.rs:3:25
  |
3 |     println!("{}", text[0]);
  |                         ^ string indices are ranges of `usize`
  |
  = help: the trait `SliceIndex<str>` is not implemented for `{integer}`
  = note: you can use `.chars().nth()` or `.bytes().nth()`
```

`&text[0..2]` on `héllo` compiles, then panics, because those two bytes split `é`:

```
 thread 'main' panicked at src/main.rs:3:25:
byte index 2 is not a char boundary; it is inside 'é' (bytes 1..3) of `héllo`
```

`.chars().next()` is the first character, as `Option`.

src/main.rs

```
 fn greet(name: &str) -> String {
    format!("Hello, {name}")
}

fn main() {
    let literal: &str = "héllo";
    let mut owned = String::from(literal);
    owned.push_str(" world");
    println!("{owned}");

    let message = greet("Ada");
    println!("{message}");
    let bytes = owned.len();
    let chars = owned.chars().count();
    println!("{bytes} {chars}");

    match owned.chars().next() {
        Some(first) => println!("{first}"),
        None => println!("empty"),
    }

    let raw = "  héllo world  ";
    let trimmed = raw.trim();
    let parts: Vec<&str> = trimmed.split(' ').collect();
    let joined = parts.join("-");
    let has_world = trimmed.contains("world");
    let swapped = trimmed.replace("world", "Rust");
    let upper = trimmed.to_uppercase();
    let lower = trimmed.to_lowercase();
    let starts = trimmed.starts_with("héllo");
    let parsed: Result<i32, _> = "42".parse();
    let left = String::from("mug");
    let right = String::from(" bowl");
    let joined_plus = left + &right;
    println!("{trimmed}");
    println!("{parts:?}");
    println!("{joined}");
    println!("{has_world} {starts}");
    println!("{swapped}");
    println!("{upper}");
    println!("{lower}");
    println!("{parsed:?}");
    println!("{joined_plus}");
```

```
}
```

`owned` prints `héllo world`. `greet` prints `Hello, Ada`. The lengths are `12` and `11`. The first character is `h`. `collect` gathers the pieces into the `Vec` named in the type. `trim` leaves `héllo world`, `split` then `join` prints `héllo-world`, `replace` prints `héllo Rust`, `to_uppercase` prints `HÉLLO WORLD`, `to_lowercase` prints `héllo world`, `parse` prints `Ok(42)`, and `+` prints `mug bowl`. After `left + &right`, `left` has moved: you cannot use it again. Prefer `format!` or `push_str` when you still need both sides.

> **Watch for:** `&text[0..2]` is a byte range. On `héllo` it panics inside `é`. Walk characters with `.chars()`.

> **Watch for:** `intval("abc")` returns `0`. `"abc".parse::<i32>()` returns `Err`, so the failure can't be missed.

> **Idiomatic Rust:** Take `&str` when a function only needs to read text. Return `String` from `format!` when the caller should own the result. `.chars()` when you mean characters.

Side by side

	PHP	Rust
Owned text	`string`	`String`
A view of text	`substr` copies	`&str`
Build a line	`"Hello, $name"`	`format!("Hello, {name}")`
Bytes	`strlen`	`.len()`
Characters	`mb_strlen`	`.chars().count()`

	PHP	Rust
First unit	`$text[0]` is a byte	`text[0]` does not compile. `.chars().next()`
Trim	`trim`	`trim()`
Split	`explode`	`split` , then `collect` into `Vec<&str>`
Join	`implode`	`join`
Contains	`str_contains`	`contains`
Replace	`str_replace`	`replace`
Upper case	`strtoupper`	`to_uppercase`
Lower case	`strtolower`	`to_lowercase`
Concatenate	`.`	`s1 + &s2` (moves `s1`), or `format!` / `push_str`
Prefix	`str_starts_with`	`starts_with`
Integer	`intval`	`parse::<i32>()`

Try it

Open `src/main.rs` in the Playground. Confirm `12 11` for `héllo world`. Change `"héllo"` to `"hello"` and run it again: the lengths become `11 11` . Put `héllo` back, then try `println!("{}", &owned[0..2]);` and read the char-boundary panic inside `é` .

Key takeaways

- `String` owns growable text. `&str` is a borrowed view, including literals and `name: &str`.
- `format!` builds a `String`. `push_str` appends. `s1 + &s2` concatenates and moves `s1`.
- `.len()` is bytes. `.chars()` is characters. `text[0]` does not compile.
- A range such as `&text[0..2]` is bytes, and it panics when it splits a character.

What's next: match and patterns