**String Types In Rust**

[[Zackary Troop](https://medium.com/@ztroop?source=post_page-----b4188e1b6182--------------------------------)](https://medium.com/@ztroop?source=post_page-----b4188e1b6182--------------------------------)

[Zackary Troop](https://medium.com/@ztroop?source=post_page-----b4188e1b6182--------------------------------)

·

Follow

3 min read

·

3 days ago

6

Rust offers a variety of types to handle text, each bringing its unique characteristics and use cases to the table. It’s important to get a handle on these types —String, &str, str, and string literals — when diving into Rust programming. There’s a chapter in [“Rust By Example” that also covers this topic pretty well](https://doc.rust-lang.org/rust-by-example/std/str.html).

Just a heads-up, this article is going to assume you’re already familiar with *ownership* and the basics of stack / heap memory allocation.

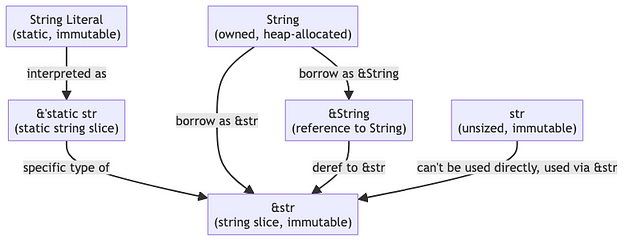


Diagram depicting different string type relationships, by Zackary Troop

First up, we have String. Think of String as a growable, heap-allocated data structure. It’s super flexible and just what you need when you’re looking to modify or take ownership of some text data. Its size isn’t fixed and can change during runtime. Since it’s stored in the heap, it has room to grow.

fn main() {  
 let mut fruit\_salad = String::from("Fruit Salad");  
  
 // Since String is growable, we can add more text to it  
 fruit\_salad.push\_str(", yummy, yummy!");  
  
 // Print the modified String  
 println!("{}", fruit\_salad); // Outputs: "Fruit Salad, yummy, yummy!"  
}

Then there’s &String, which is essentially a read-only reference to a String. Imagine it as a way to glimpse at a String without claiming it as your own. If you wondering, yes, you can have a mutable reference as well.

fn main() {  
 let mut fruit\_salad = String::from("Fruit Salad");  
  
 // Use the immutable reference to access the String  
 println!("Accessed via immutable reference: {}", &fruit\_salad);  
  
 // Create a mutable reference to the String  
 let fruit\_salad\_mut\_ref = &mut fruit\_salad;  
  
 // Use the mutable reference to modify the String  
 fruit\_salad\_mut\_ref.push\_str(", yummy, yummy!");  
  
 // Now the original String is modified  
 println!("Original String after modification: {}", fruit\_salad);  
}

The &str type is the topic of interest. It’s a slice or view into a string and is one of the most commonly used string types in Rust. It’s a borrowed type, so it doesn’t own the data it refers to. The thing about &str is that it’s immutable — you can’t change it. It’s perfect for passing string data around when you don’t need ownership. Its size is known at compile time.

*What is a slice exactly? A slice lets you access a contiguous sequence of elements within a collection, like an array or a string, without owning it.*

fn main() {  
 let fruit\_salad = String::from("Fruit Salad");  
  
 // Borrowing the String as an immutable &str  
 let fruit\_borrowed = &fruit\_salad[..];  
  
 // Passing the &str to a function  
 print\_str(fruit\_borrowed);  
  
 // Demonstrating that &str is a slice of String  
 // Let's take a slice of the first 5 characters  
 let fruit\_slice = &fruit\_salad[..5];  
 println!("Slice of the first 5 characters: {}", fruit\_slice);  
}  
  
// A function that takes &str  
// This is efficient because it does not take ownership of the string data  
fn print\_str(s: &str) {  
 println!("String slice: {}", s);  
 // Note that the size of &str is known and can be used  
 println!("Length of the string slice: {}", s.len());  
}

Now, str is a bit of a different beast. It’s an unsized string type that represents a sequence of UTF-8 characters. You don’t really use str directly; instead, you’ll encounter it as &str. Once a str is created, it can’t be altered.

Lastly, &’static str are your string literals. These are the hard-coded, immutable strings that are embedded right into your code. They’re always there, throughout the entire run of the program.

So, there you have it — a quick guide around string types in Rust. Each has its place, and knowing which to use when is a key part of becoming proficient in Rust.