**Ownership in Rust:**

 Here are some of the types that implement Copy:

* All the integer types, such as u32.
* The Boolean type, bool, with values true and false.
* All the floating-point types, such as f64.
* The character type, char.
* Tuples, if they only contain types that also implement Copy. For example, (i32, i32) implements Copy, but (i32, String) does not.

Other type like str, vector, etc. don’t implement copy, we must clone () this value to make a copy for it, because if we assign string 2 = string 1, the first string will be move to second string and pointer of 1st string will be invalid.

When we assign string 2 = string 1, rust move string 1 into string 2, not using shallow copy or deep copy because shallow copy can cause memory leak when out of scope.

For example, of C and C++:

A screenshot of a computer code

AI-generated content may be incorrect.

When string 1 and string 2 go out of scope, they will both try to free the same memory. This is known as a *double free* error and is one of the memory safety bugs we mentioned previously. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

On the other hand, deep copy doesn’t have this problem, but the operation s2 = s1 could be very expensive in terms of runtime performance if the data on the heap were large. Also, it occupies lots of memory when copy the large array/string, etc.

But with rust when make an operation s2 = s1, the s1 value move to s2 so s1 value is no longer valid so this can prevent double free error when s1 and s2 out of scope. To copy value from s1 to s2, we must use clone().

A diagram of a number

AI-generated content may be incorrect.

Figure 1 Shallow copy s1 and s2 (s1 and s2 own same memory zone)

A diagram of a number

AI-generated content may be incorrect.

Figure 2 Deep copy (s1 memory zone is copied to s2)

Reference and Borrowing:

In Rust. A *reference* is like a pointer in that it’s an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference. This helps avoid unnecessary copying or moving of data while maintaining memory safety.

There are two types of reference:

Immutable reference:

* By Using it, you can read data but not modify.
* Multiple immutable references are allowed at the same time.
* The compiler guarantees that the data won’t change during the borrow.

Example:

A computer screen shot of a code

AI-generated content may be incorrect.

Mutable reference:

* You can modify the data it points to.
* Only one mutable reference is allowed at any given time to prevent data races.

Example:

A computer screen with green and blue text

AI-generated content may be incorrect.

Common Error:

Mixing mutable and immutable references:

A computer screen shot of a code

AI-generated content may be incorrect.

**Dangling References**

Rust **prevents** references that point to invalid memory:

A screen shot of a computer code

AI-generated content may be incorrect.

Because s is created inside dangle, when the code of dangle is finished, s will be deallocated. But we tried to return a reference to it. That means this reference would be pointed to an invalid String. That’s no good! Rust won’t let us do this.

Rust prevents this at compile time using **lifetimes**.

Explain lifetime:

A lifetime in Rust is a way for the compiler to ensure that references are always valid, they do not outlive the data they point to.

Example:

A computer screen with text and symbols

AI-generated content may be incorrect.

To prevent it, here is the solution:

A computer screen shot of a computer code

AI-generated content may be incorrect.

**The Slice Type:**

A **slice** in Rust is a reference to a *contiguous sequence of elements* in a collection, such as a String or an array. Slices enable efficient, zero-copy access to portions of a collection **without transferring ownership**. Because they are references, slices adhere to Rust’s borrowing rules and lifetimes.

Syntax: &collection[start..end].

Why Are Slices Important?

* Enable access to a subset of data without cloning or moving.
* Avoid manual index tracking, which can easily become invalid after mutation.
* Ensure memory safety through compile-time guarantees.
* Facilitate general and reusable APIs, especially when working with &str and &[T].

**How Slices Work**

Internally, a slice is represented as a **fat pointer**:

* One-part points to the first byte/element.
* The other holds the length.

This structure allows the compiler to enforce bounds checking and memory safety, while giving the developer precise control over partial data access.

A diagram of a number

AI-generated content may be incorrect.

Figure 3 How slice works

Problem Without Slices:

If you want to write a function to return the length of first word from a string:

A screen shot of a computer code

AI-generated content may be incorrect.

This function returns the index of the first space. However, if the string is modified afterward (e.g., cleared), this index becomes **invalid**, introducing a logical bug:

A screenshot of a computer code

AI-generated content may be incorrect.

Solution With Slices:

A screen shot of a computer code

AI-generated content may be incorrect.

* The function now returns a &str, a slice into the original string.
* Because the returned slice borrows the original string, Rust will prevent the original string from being mutated while the slice is still in use.
* This avoids invalidation bugs at compile time.

Example:

A computer code with text

AI-generated content may be incorrect.

**Flexible Signature**

Using &str for the parameter instead of &String allows the function to accept:

* A full String
* A string slice (&str)
* A literal string slice
* The literal itself (e.g., "hello")

Using this:



Instead of this:



Slices are not exclusive to strings. You can also slice arrays and other collections:

A close-up of a number

AI-generated content may be incorrect.

* Like &str, this is a non-owning reference.
* Type is &[T], where T is the element type.

**Advantages of Slices:**

* Avoid cloning or allocating memory for sub-data.
* Make functions more flexible with &str or &[T] parameters.
* Prevents use-after-free or dangling index bugs.
* Just a pointer + length — extremely lightweight abstraction.

**Common Pitfalls**

* UTF-8 boundary violation: Slicing in the middle of a multibyte character causes a panic.
* Mixing ownership and slices: Mutating a collection while holding a slice will lead to compiler errors.