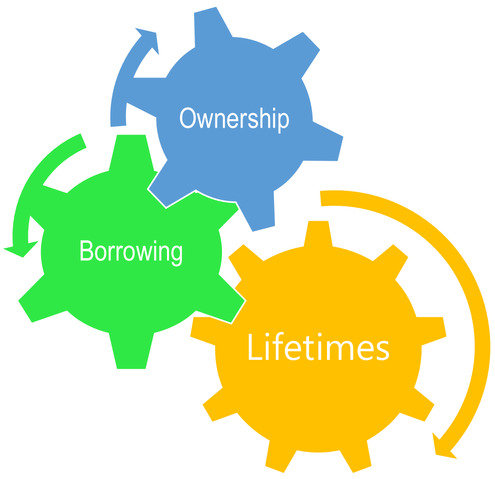
Ownerships

Ownership System

- One of Rust's most unique and compelling features
- Ownership is a way to achieve Rust's largest goal, **memory safety**.
- Three related concepts:
  - Ownership
  - Borrowing
  - Lifetimes



Ownership Rule

1. **Each value has a variable that's called its owner.**
2. **There can only be one owner at a time.**
3. **When the owner goes out of scope, the value will be dropped.**

Ownership is a set of rules that govern how a Rust program manages memory and its safety.

Variable Scope

- A scope is the range within a program for which an item is valid.
- The variable is valid from the point at which it's declared until the end of the current scope (*really?*).

"hello" itself is allocated on the heap

```
fn main() {
    // s is not valid here, it's not yet declared
    let s = String::from("hello"); // s is valid from this point forward

          // do stuff with s

} // this scope is now over, and s is no longer valid
  // s and "hello" are now dropped
```

allocated on the stack

Ownership

- **Variable bindings “have ownership” of what they’re bound to.**
- This means that when a binding goes out of scope, Rust will free the bound resources.

vector itself is allocated
on the heap

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

- When v goes out of scope at the end of foo( ), Rust will clean up everything related to the vector, even the heap-allocated memory. This happens deterministically, at the end of the scope.

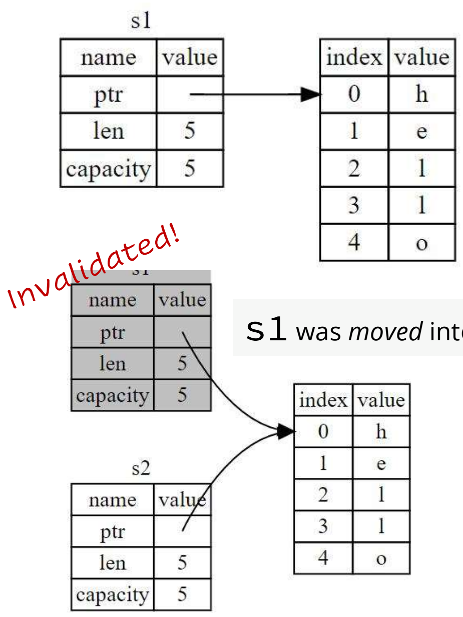
What will happen?

```
let s = String::from("hello");
let _s2 = s;
println!("s is: {s}");
```

```
error[E0382]: borrow of moved value: `s`
2 |   let s = String::from("hello");
  |       - move occurs because `s` has type `String`, which does not
implement the `Copy` trait
3 |  let _s2 = s;
  |            - value moved here
4 |  println!("s is: {s}");
  |                   ^ value borrowed here after move
  |
…
3 |  let _s2 = s.clone();
```

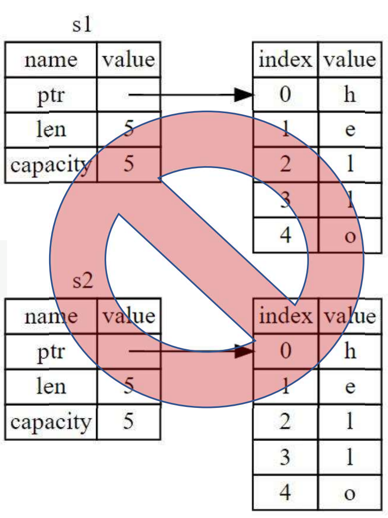
```
let s1: String = String::from("hello");
let s2: String = s1; // Move (not shallow copy)
```

Actually, any type that does not implement Copy trait is moved.

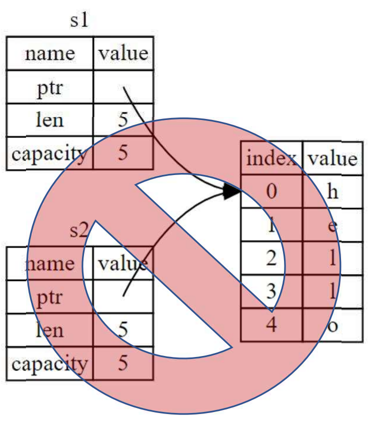


s1 was *moved* into s2

Deep copy



Shallow copy



Move semantics

- Rust ensures that there is exactly one binding to any given resource.

```
let s = String::from("hello");
let _s2 = s;
println!("s is: {s}");
```

- When we transfer ownership to something else, we say that we've '**moved**' the thing we refer to. It's the ***default thing*** that Rust does.
- When ownership is transferred to another binding, you cannot use the original binding.

Ownership and Functions

- Move also occurs when passing parameters to functions.

```
fn main() {
    let s = String::from("hello");  // `s` comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    println!("{s}");
}

fn takes_ownership(some_string: String) { // `some_string` comes into scope
    println!("{some_string}");
} // Here, `some_string` goes out of scope and `drop` is called. The backing
  // memory is freed.
```

Return Values and Scope

- Move occurs when returning values from functions, too.

```
fn main() {
  let s1 = gives_ownership();  // `gives_ownership` moves its return value into `s1`
} // Here, `s1` goes out of scope and is dropped.

fn gives_ownership() -> String { // `gives_ownership` will move its return value
                                 // into the function that calls it

  let some_string = String::from("yours"); // `some_string` comes into scope

    some_string  // `some_string` is returned and moves out to the calling function
}
```

```
fn main() {
  let s1 = gives_ownership();  // `gives_ownership` moves its return value into `s1`

  let s2 = String::from("hello");     // `s2` comes into scope

  let s3 = takes_and_gives_back(s2);  // `s2` is moved into `takes_and_gives_back`,
  println!("{s3}");                   // which also moves its return value into `s3`

} // Here, `s3` goes out of scope and is dropped. `s2` was moved, so nothing
  // happens. `s1` goes out of scope and is dropped.

fn gives_ownership() -> String {
  let some_string = String::from("yours"); // `some_string` comes into scope
  some_string
}

// This function takes a String and returns one
fn takes_and_gives_back(mut a_string: String) -> String { // `a_string` comes into scope
  a_string.push_str(", world");  // `a_string` is returned and moves out to
  a_string                       // the calling function
}
```

What has happened here?

```
let s = String::from("hello");
let _s2 = s;
println!("s is: {s}");
```

Why different behavior?

```
let x: i32 = 42;
let _y: i32 = x;
println!("x is: {x}");
```

```
// x is 42
```

Values with Known, Fixed Size at Compile Time

```rust
#[test]
fn copy_for_simple_values() {

    // integers are simple values with a known, fixed size
    let x: i32 = 5;
    let y: i32 = x; // copy

    assert_eq!(x, y);
}
```

Variables with Unknown Size at compile time

```rust
fn main() {
    let s1: String = String::from("hello");
    let _s2: String = s1; // move (not shallow copy)
    println!("{s1}, world!");
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from(”hello”);
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!(”{}, world!”, s1);
  |                            ^^ value borrowed here after move
```

Stack-allocatable Data: Copy

- Types like integers having a known size at compile time can be stored on the stack.
- Actually, for ***any type that implements the*** *Copy* ***trait***, variables that use it are just ***copied***, making them still valid after assignment to another variable.

```
let x: i32 = 5;
let _y: i32 = x; // copied
println!("x = {x}, y = {y}");
```

Rust won't let us annotate a type with Copy if the type, or any of its parts, has implemented the Drop trait.

Types that implements Copy trait

As a general rule, any group of simple scalar values, *nothing that requires some form of resource allocation* can implement `Copy`.

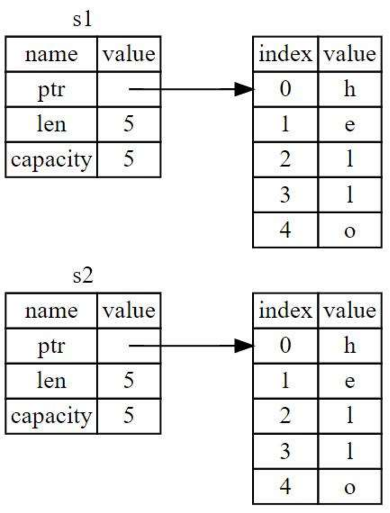
- All the numeric types, such as `u32` as `f64`, etc.
- Boolean type, `bool`, with values `true` and `false`.
- Character type, `char`.
- Tuples, if they only contain types that also implement `Copy`.
  - For example, `(i32, i32)` implements `Copy`, but `(i32, String)` does not.

Variables and Data Interacting with Clone

- If we do want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called `clone`.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("s1 = {s1}, s2 = {s2}");
}
```

Deep copy



Memory Management in Rust

- The heap allocated memory is automatically returned once the variable that owns it goes out of scope.

```
fn main() {
  { // `s` is not valid here, it’s not yet declared
     let s: String = String::from("hello"); // `s` is valid from this point forward

    // do stuff with `s`
  } // this scope is now over, and `s` is no longer valid
}
```

- Rust calls a special function, called `drop`, automatically at the closing curly bracket.

Note: In C++, this pattern of deallocating resources at the end of an item’s lifetime is sometimes called **Resource Acquisition Is Initialization (RAII).**

Ownership Summary

The ownership of a variable follows the same pattern every time:

- Assigning a value to another variable **moves** it *unless* it implements `Copy`.
- When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless ownership of the data has been moved to another variable.

What if we want to let a function use a value but not take ownership?

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(s1);

    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: String) -> i32 {
    let length = s.len(); // len() returns the length of a String

    length
}
```

What if we want our ownership back? (Solution?)

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

References (&)

- A **reference** is ***like a pointer*** in that it's an address we can follow to access the data owned by some other variable.
- *Unlike a pointer*, a reference is *guaranteed to point to a valid data* for the life of that reference. (**No dangling references**!)

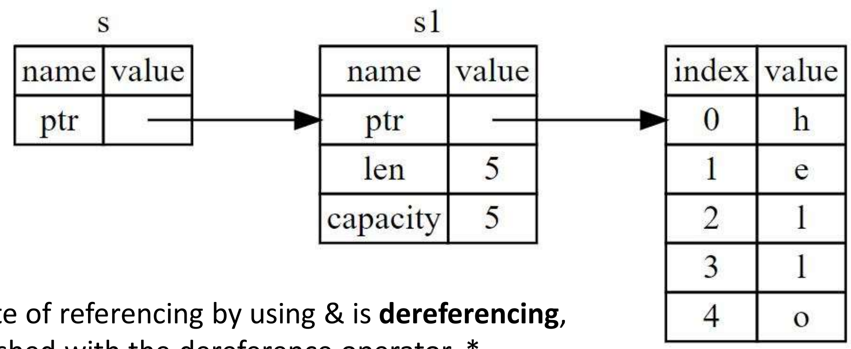
```
fn main() {
    let x: i32 = 42;
    let y: &i32 = &x;

    assert_eq!(*y, x);
    println!("x = {x}");
    println!("y = {:p}", y);
    println!("y = {y}");
}
```

```
fn main() {
    let s1 = String::from("hello");
    let len = calc_length(&s1);
    println!("Length of '{s1}' is {len}");
}

fn calc_length(s: &String) -> usize {
    s.len()
}
```

References and Borrowing

- The & represents a **reference**, and allows you to refer to some value without taking ownership of it, just *borrows* the ownership.

```
let s1: String = String::from("hello");
let s: &String = &s1;
```



Note: The opposite of referencing by using & is **dereferencing**, which is accomplished with the dereference operator, *

References and Borrowng (cont'd)

```
fn calc_length(s: &String) -> usize { // `s` is a ref. to a String
    s.len()
} // Here, `s` goes out of scope. But because it does not have ownership
  // of what it refers to, the referred value is not dropped.
```

- A binding that borrows something does not deallocate the resource when it goes out of scope.
- The scope of the variable s is still inside the function, but the value pointed to by the reference is not dropped when s gets out of scope, because s doesn't have ownership.
- We call the action of creating a reference *borrowing*.

References are Immutable

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Already borrowed as immutable

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
```

- Just as variables, **references are immutable by default**. We're not allowed to modify something we have a reference to.

Mutable References

- We create a mutable reference with `&mut`.

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Mutable References

- We create a mutable reference with `&mut`.

```rust
fn main() {
     let s = String::from("hello");

     change(&mut s);
}

fn change(some_string: &String) {
     some_string.push_str(", world");
}
```

Mutable References

- We create a mutable reference with `&mut`.

```rust
fn main() {
     let s = String::from("hello");

     change(&mut s);
}

fn change(some_string: &mut String) {
     some_string.push_str(", world");
}
```

Mutable References

- We create a mutable reference with &mut.

Notice that `s` had to be marked **mut** as well. If it wasn't, we couldn't take a mutable borrow from immutable owner.

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Can we take immutable borrow from mutable owner?

Mutability Puzzle

- Let's consider the following:

```
let (mut x, mut y) = (5, 42);
let z = &mut x;

*z += 1;     // Is this OK?
z = &mut y; // Is this OK?
// z is an immutable binding to mutable reference (\(>o<)ノ)
```

You can mutate the thing that's bound to z (*z=5), but you can't bind z to something else (z = &mut y).

- Of course, if you need both:

```
let mut x = 5;
let mut z = &mut x;
```

Now z can be bound to another value, and the value itself can be changed.

Scope of a Reference

"Lifetime" is more appropriate term for references.

- In most languages, the scope denotes the region in which a variable is visible.
- In Rust, however, a reference's scope starts from *where it is introduced* and continues through the *last time that reference is used or dropped*.

```
fn non_overlapping_scopes() {
  let mut s = String::from("hello");
  let r2 = &mut s; // r2's scope starts
                   // and ends here
  {
    let r1 = &s; -----------------+
    println!("r1: {r1:?}"); -- --+
  } // `r1` goes out of scope here
} // `r2` goes out of scope here
```

```
fn overlapping_scopes() {
  let mut s = String::from("hello");
  let r2 = &mut s;  ----------------------+
  {                                       |
    let r1 = &s;  ---------------------+  |
    println!("r1: {r1:?}"); ----------+   |
  } // `r1` goes out of scope here        |
                                          |
  r2.push('!');                           |
  println!("r2: {r2:?}"); // ------------+
} // `r2` goes out of scope here
```

Restrictions of Mutable References

- If you have an "active" mutable reference to a value, you can have *no other "active" mutable references* to that value.
- By "active" references, I mean whose scopes are overlapped.

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
```

Why this restriction, then?

Preventing Data Races at Compile time

- This restriction allows us to prevent *data races* at compile time.
- A data race happens when these three behaviors occur:
  - Two or more pointers access the same data at the same time.
  - At least one of the pointers is being used to write to the data.
  - There's no mechanism being used to synchronize access to the data.

```
fn main() {
    let mut s = String::from("hello");
    {
        let r1 = &mut s;
        println!("r1: {r1:?}");
    } // `r1` goes out of scope here, so we can make
      // a new reference with no problems.
    let r2 = &mut s;
    r2.push_str(", world!");
    println!("r2: {r2:?}");
}
```

Multiple & and One &mut

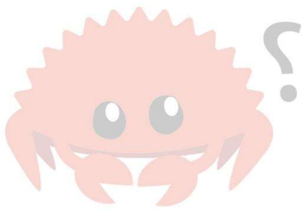
- We also cannot have a mutable reference while we have an immutable one to the same value.

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem

    println!("{r1}, {r2}");
}
```

Multiple & and One &mut (Cont’d)

- We also cannot have a mutable reference while we have an immutable one to the same value.

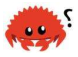
```rust
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // no problem yet


}
```



Multiple & and One &mut (Cont’d)

- We also cannot have a mutable reference while we have an immutable one to the same value.

```rust
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{r1}, {r2}, and {r3}");
}
```



Multiple & and One &mut (Cont'd)

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // variables `r1` and `r2` will not be used after this point

    let r3 = &mut s; // no problem
    println!("{r3}");
}
```

A dangling pointer

```
const int *my_int_ptr;
{
  int my_int = 5;
  my_int_ptr = &my_int;
}
cout << *my_int_ptr;
```

```
let my_int_ptr: &i32;
{
    let my_int: i32 = 5;
    my_int_ptr = &my_int;
}
dbg!(*my_int_ptr);
```

In Rust, …

```
let my_int_ptr: &i32;
{
    let my_int: i32 = 5;
    my_int_ptr = &my_int;
}
dbg!(*my_int_ptr);
```

```
 Compiling playground v0.0.1 (/playground)
error[E0597]: `my_int` does not live long enough
 --> src/main.rs:5:18
  |
5 |     my_int_ptr = &my_int;
  |                  ^^^^^^^ borrowed value does not live long enough
6 | }
  | - `my_int` dropped here while still borrowed
7 | dbg!(*my_int_ptr);
  |      ----------- borrow later used here
```

Dangling References

- In Rust, the compiler guarantees that references will never be dangling references.
- The compiler will ensure that the **reference will not outlive the data** it refers to.

```
fn main() {
    let reference_to_nothing = dangle();
    … // code using reference_to_nothing
}
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, `s`
} // Here, `s` goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

*Compiler (well, actually **borrow checker**) will not permit this to happen?*

One of Solutions

- If we change the return type from `&String` to `String`:

```
fn main() {
    let string = no_dangle();
}

fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

- This works without any problems. Ownership is moved out, and nothing is deallocated.

The Rules of References

- **Shared references** (aliasing), `&T`, like const pointers in C/C++
- **Mutable references**, `&mut T`, like non-const pointers in C/C++
- References don't keep things alive. There is no garbage collector.

BIG IDEA #1

– References are always valid. No dangling references.

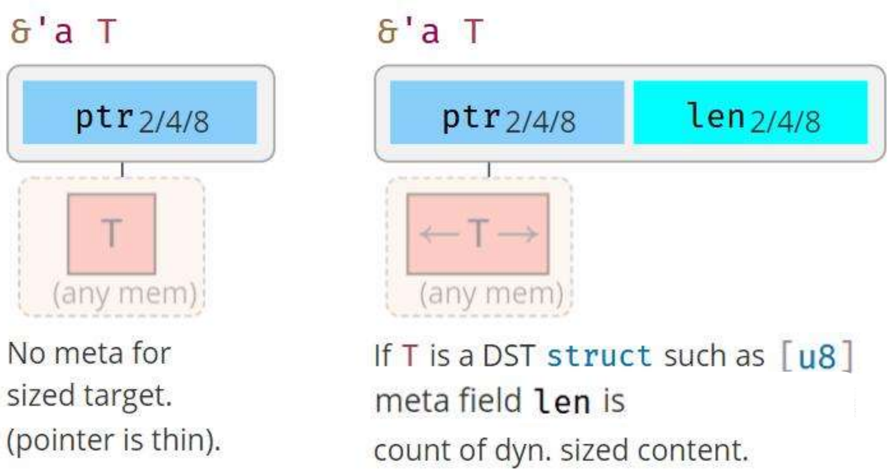
BIG IDEA #2

– Mutable references are **unique**. No mutable aliasing.

BIG IDEA #3

– Shared references* $\oplus$ Mutable Reference?

Memory Layout for References



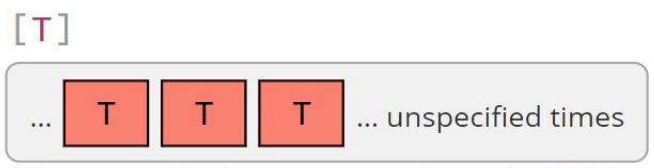
No meta for sized target. (pointer is thin).

If T is a DST struct such as [u8] meta field len is count of dyn. sized content.

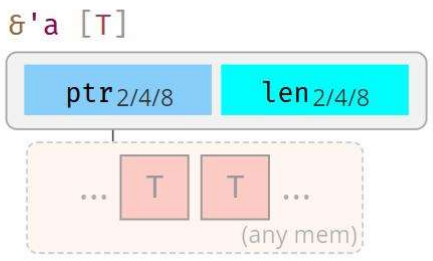
Next, we’ll look at a different kind of reference: **slices**.

The Slice Type: [T]

- **Slices** let you reference a contiguous sequence of elements of type `T` in a collection rather than the whole collection.
- Most often lives behind **slice reference**, `&[T]`
- A slice reference(`&[T]`) does <u>not</u> have ownership.



Slice type of unknown-many elements. Neither `Sized` (nor carries `len` information), and most often lives behind reference as `&[T]`.



Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.

String type

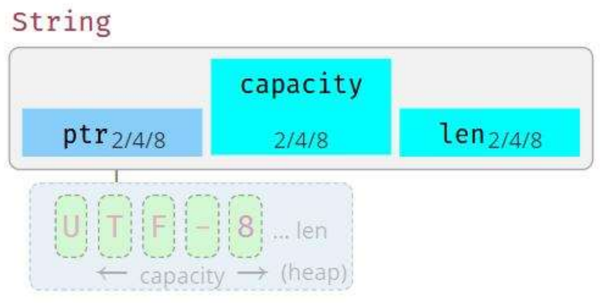
- A growable and/or shrinkable sequence of characters, which is *mutable*.
- Stored on the *heap* instead of the stack due to its unknown size at compile time.

A string literal is of type **&slice** which is **immutable**!

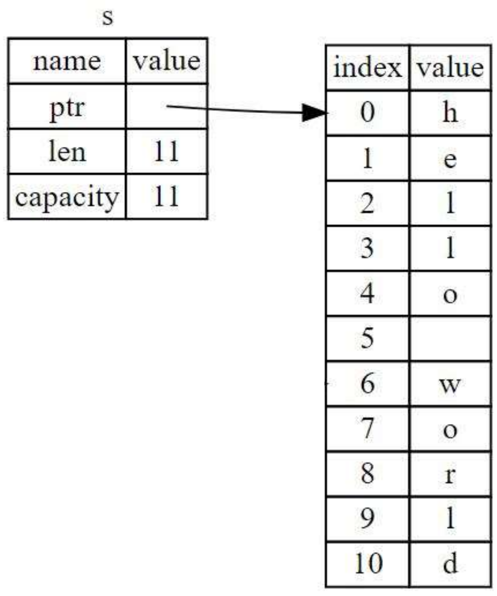
```
fn main() {
    let mut s = String::from("hello");
    s.push_str(", world!"); // push_str() appends a literal to a String
    println!("{s}"); // This will print `hello, world!`
}
```

(Owned) String type

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



Observe how `String` differs from `&str` and `&[char]`.



name	value
ptr	
len	11
capacity	11

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

Problem Statement for Slice

Write a function

- **Input**: a string of words separated by spaces
- **Output**: the first word it finds in that string.

If the function doesn’t find a space in the string, the whole string must be one word, so the entire string should be returned.

```
fn first_word(s: &String) -> ???
```

Return the index of the end of the first word

```
fn first_word(s: &String) -> usize {
    let bytes: &[u8] = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}

fn second_word(s: &String) -> (usize, usize) { … }
```

What if …

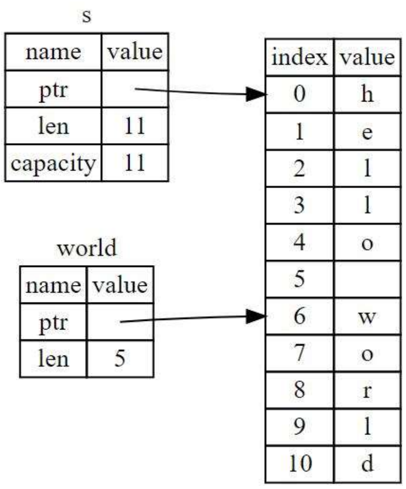
```
fn main() {
  let mut s = String::from("hello world");

  let index = first_word(&s); // word will get the value 5

  s.clear(); // this empties the String, making it equal to ""

  // What if we try to extract the first word later?

  // `index` still has the value 5 here, but there's no more string that
  // we could meaningfully use the value with. `index` is now totally invalid!

}
```

`index` isn't connected to the state of `s` at all.
That is `index` may be out of sync with `s`.

String slices (str and &str)

- A string slice (reference) is a reference to part of a `String`.

```
fn main() {
    let s = String::from("hello world");

    let hello: &str = &s[0..5];
    let world: &str = &s[6..11];
}
```



Note: String slice range indices must occur at valid UTF-8 character boundaries. If you attempt to create a string slice in the middle of a multibyte character, your program will exit with an error.

Return a String slice

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Compiler will ensure the references into the String remain valid

```
fn main() {
    let mut s = String::from("hello world");

    let word: &str = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {word}");
}
```

```
pub fn clear(&mut self)
{ … }
```

Not only has Rust made our API easier to use, but it has also eliminated an entire class of errors at compile time!

String Literals as Slices

- Recall that we talked about string literals being stored inside the binary.

```
let s: &str = "Hello, world!";
```

- The type of `s` here is `&str`: it's a slice pointing to that specific point of the binary. This is also why **string literals are immutable**.
- `&str` **is an immutable reference**.

String Slices as Parameters

- ***Define a function using a string slice instead of a reference to a*** *`String`* makes our API more general and useful without losing any functionality.

```
fn first_word(s: &String) -> &str {
```

⇓

```
fn first_word(s: &str) -> &str {
```

String Slices as Parameters Demo (1)

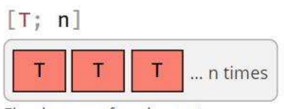
```
fn main() {
  let my_string_literal: &str = "hello world";

  // `first_word` works on slices of string literals, whether partial or whole
  let word = first_word(&my_string_literal[0..6]);
  let word = first_word(&my_string_literal[..]);

  // Because string literals *are* string slices already,
  // this works too, without the slice syntax!
  let word = first_word(my_string_literal);
}
```

String Slices as Parameters Demo (2)

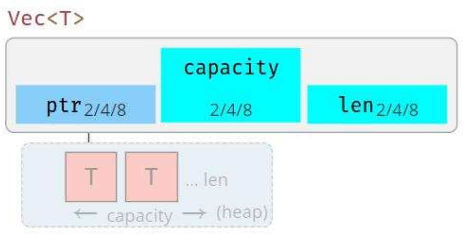
```
fn main() {
  let my_string: String = String::from("hello world");

  // `first_word` works on slices of `String`s, whether partial or whole
  let word = first_word(&my_string[0..6]);
  let word = first_word(&my_string[..]);

  // `first_word` also works on references to `String`s, which are equivalent
  // to whole slices of `String`s
  let word = first_word(&my_string);
}
```

implicit ***deref coercion***!
&String => &str

Other Slices



Fixed array of **n** elements.



Regular *growable array* vector of single type.

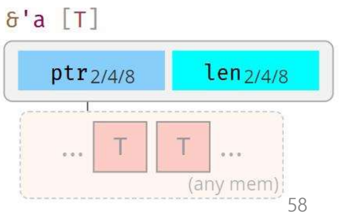
- Array slices

```
let a = [1, 2, 3, 4, 5];
let slice: &[i32] = &a[1..3];
assert_eq!(slice, &[2, 3]);
```

- Vector slices

```
let vs: Vec<i32> = vec![1, 2, 3, 4, 5];
let slice: &[i32] = &vs[1..3];
assert_eq!(slice, &[2, 3]);
```

Works the same way as string slices do, by storing a reference to the first element and a length.



Summary

- The concepts of ownership, borrowing, and slices ensure memory safety in Rust programs at compile time.
- Having the owner of data automatically clean up that data when the owner goes out of scope means you don't have to write and debug extra code to get memory management control.
- Ownership affects how lots of other parts of Rust work.