Ownership

Mustakimur R. Khandaker

Stack and Heap

Both the stack and the heap are parts of memory available to program use at runtime.

- But they are structured in different ways.

The stack stores values in the order it gets them and removes the values in the opposite order.

- Referred to as ***last in, first out***. (LIFO)
- Adding data is called ***pushing onto the stack***, and removing data is called ***popping off the stack***.
- All data stored on the stack must have a known, fixed size.

The heap is less organized.

- We explicitly request a certain amount of space.
  - The memory allocator finds an empty spot in the heap that is big enough, marks it as being in use, and returns a pointer i.e. address of that location. ***(allocating)***

Stack allocation is faster than heap allocation.

- **Heap:** requires exhaustive search. **Stack:** on top of the current allocated memory.

Heap memory access is slower than stack memory access.

- A processor can access data faster that's close to other data (as it is on the stack) rather than farther away (as it can be on the heap).

Code

```c
#include <stdio.h>

void encrypts(char *key)
{
   while (key != '\0')
   {
       key = (char)(key + 10) % 128;
   }
   printf("%s\n", key);
}

int main()
{
   char key[16];
   strcpy(key, "whatever");
   encrypts(key);
   memset(key, 0, 16);
}
```

Source

```
0000000000400660 <main>:
 400660: 55                     push   rbp
 400661: 48 89 e5               mov    rbp,rsp
 400664: 48 83 ec 30            sub
rsp,0x30
 ...
 400685: e8 76 ff ff ff         call   400600
<encrypts>
 ...
 40069a: e8 61 fe ff ff         call   400500
<memset@plt>
 ...
 4006a6: 5d                     pop    rbp
 4006a7: c3                     ret
```

clang -O0

```
00000000004005a0 <main>:
 4005a0: 48 83 ec 18            sub
rsp,0x18
 ...
 4005ba: e8 a1 ff ff ff         call   400560
<encrypts>
 ...
 4005c1: 48 83 c4 18            add
rsp,0x18
```

clang -O1

Ownership

Ownership is Rust's most unique feature.

- It enables Rust to make memory safety guarantees without needing a garbage collector.

Memory is managed through a system of ownership with a set of rules that the compiler checks at compile time.

- None of the ownership features slow down the runtime.

Ownership rules:

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

String in Rust

Why am I able to mutate a string literal?!

string literal contents are known at compile time. ***(immutable)***

- The text is hardcoded directly into the final executable. ***(read-only memory)***
- This is why string literals are fast and efficient.

```rust
fn main() {
    // string literal are immutable
    let s1: &str = "hello";
    // but the pointer refers to string literals can be mutable
    let mut s: &str = s1;
    println!("{}", s);
    // so you can change a mutable pointer where refer to
    s = "there";
    // but you cannot change the literal itself
    // s1 = "Try";
    println!("{} {}", s1, s);

    // you cannot store a literal to a String type
    // let src: String = "hello";
}
```

Continue ....

String type is allocated on the heap.

- It is able to store an amount of text that is unknown to us at compile time. ***(user input)***
- You can create a String from a string literal using the from function.

```
let s: String = String::from("hello");
//s.push_str(", world!"); // without mut String, cannot borrow it
println!("{}", s);

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

The double colon (::) is an operator that allows us to namespace this particular from function under the String type.

- The memory must be requested from the memory allocator at runtime.
- Returning this memory to the allocator when execution is done with the String.
    - Unlike GC (garbage collector) or manual *alloc* and *free* of C, the memory is automatically returned once the variable that owns it goes out of scope.

Variable Scope

A scope is the range within a program for which an item is valid.

```
{
    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
```

When a variable goes out of scope, Rust calls a special function, called drop.

- Rust calls drop automatically at the closing curly bracket.
-

```
c++filt _ZN4core3ptr13drop_in_place17h2e6e11db72490ffcE
```

Function core::ptr::drop_in_place 1.8.0 [-][src]

[+] Expand attributes

```
pub unsafe fn drop_in_place<T: ?Sized>(to_drop: *mut T)
```

[-] Executes the destructor (if any) of the pointed-to value.

Continue ....

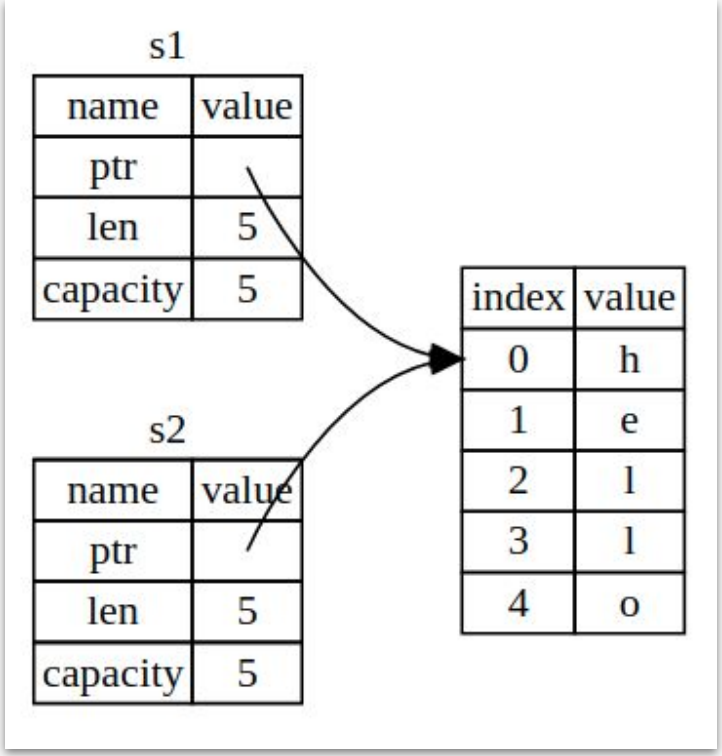
```
let s1 = String::from("hello");
let s2 = s1;
```

A String is made up of three parts (shown on the left):

- a pointer to the memory that holds the contents of the string, a length, and a capacity.
  - This group of data is **stored on the stack**.
- On the right is the **memory on the heap** that holds the contents.

- The length is how much memory, in bytes, the contents of the String is currently using.
- The capacity is the total amount of memory, in bytes, that the String has received from the allocator.

Assign s1 to s2, the String data is copied i.e. copy the pointer, the length, and the capacity (on the stack) except the data on the heap.



Code

```
let s1 = String::from("source string");
let s2 = s1;

/*
error[E0382]: borrow of moved value: `s1`
  --> src/main.rs:38:20
   |
35 |     let s1 = String::from("source string");
   |         -- move occurs because `s1` has type `String`,
which does not implement the `Copy` trait
36 |     let s2 = s1;
   |              -- value moved here
37 |
38 |     println!("{}", s1);
   |                    ^^ value borrowed here after move
*/

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

When s2 and s1 go out of scope, they will both try to free the same memory.

- known as a double free error.

To avoid double free:

Rust considers s1 to no longer be valid.

- Therefore, Rust doesn't need to free anything when s1 goes out of scope.

Rust called it Move instead of ***shallow copy***.

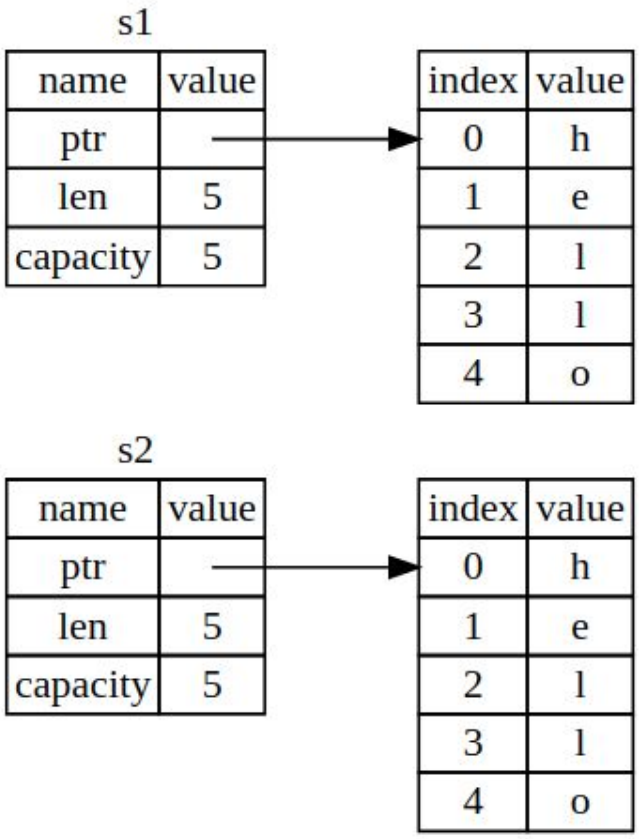
In this example, we would say that s1 was moved into s2.

Continue ....

```
let mut m1 = String::from("lets try mut string");
let mut m2 = m1;
/*
54 |     let mut m1 = String::from("lets try mut string");
   |         ----^^
   |         |
   |         help: remove this `mut`
   |
   = note: `#[warn(unused_mut)]` on by default

warning: variable does not need to be mutable
  --> src/main.rs:55:9
   |
55 |     let mut m2 = m1;
   |         ----^^
   |         |
   |         help: remove this `mut`

error[E0382]: borrow of moved value: `m1`
  --> src/main.rs:56:20
   |
54 |     let mut m1 = String::from("lets try mut string");
   |         ------ move occurs because `m1` has type `String`, which does not implement the `Copy` trait
55 |     let mut m2 = m1;
   |                  -- value moved here
56 |     println!("{}", m1);
   |                    ^^ value borrowed here after move
    */
// println!("{}", m1);
println!("{}", m2);
```

Deep Copy (Clone)

```rust
    println!("{}", m2);

    let s1 = String::from("hello");
    let s2 = s1.clone();

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



Regular Copy Behavior

Rust has a special annotation called the Copy trait.

- we can place on types like integers that are stored on the stack.
- Rust won't let us annotate a type with the Copy trait.
  - The type, or any of its parts, has implemented the Drop trait.

Here are some of the types that are 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 are also Copy. For example, (i32, i32) is Copy, but (i32, String) is not.

Code

```rust
{
    let mut x = 5;
    let y = x;
    println!("{} => {}", x, y);
    x = 10;
    println!("{} => {}", x, y);
}
```

```rust
{
    let mut x = 15;
    let y = &x;
    println!("{} => {}", x, y);
    /*
    error[E0506]: cannot assign to `x` because it is borrowed
       --> src/main.rs:112:7
        |
    110 |        let y = &x;
        |                -- borrow of `x` occurs here
    111 |        println!("{} => {}", x, y);
    112 |        x = 20;
        |        ^^^^^^ assignment to borrowed `x` occurs here
    113 |        println!("{} => {}", x, y);
        |                               - borrow later used
here
        */
    //x = 20;
    println!("{} => {}", x, y);
}
```

```rust
fn alt() {
  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

  /*******
  error[E0382]: borrow of moved value: `s`
     --> src/main.rs:176:19
      |
  171 |     let s = String::from("hello"); // s comes into scope
      |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
  172 |
  173 |     takes_ownership(s); // s's value moves into the function...
      |                     - value moved here
  ...
  176 |     println!("{}", s);
      |                    ^ value borrowed here after move
  ********/
  //println!("{}", s);

  let x = 5; // x comes into scope

  makes_copy(x); // x would move into the function,
                 // but i32 is Copy, so it’s okay to still
                 // use x afterward
  println!("{}", x);
} // Here, x goes out of scope, then s. But because s's value was moved, nothing
 // special happens.
```

```rust
fn takes_ownership(some_string: String) {
  // some_string comes into scope
  println!("{}", some_string);

  /******
  error[E0596]: cannot borrow `some_string` as mutable, as it is not declared as mutable
     --> src/main.rs:203:4
      |
  199 | fn takes_ownership(some_string: String) {
      |                    ----------- help: consider changing this to be mutable: `mut some_string`
  ...
  203 |     some_string.push_str("world");
      |     ^^^^^^^^^^^ cannot borrow as mutable
  ******/

  // some_string.push_str("world");
} // Here, some_string goes out of scope and `drop` is called. The backing
 // memory is freed.
```

```rust
fn makes_copy(some_integer: i32) {
  // some_integer comes into scope
  println!("{}", some_integer);
  /*******
     error[E0384]: cannot assign to immutable argument `some_integer`
     --> src/main.rs:222:4
      |
  219 | fn makes_copy(some_integer: i32) {
      |               ------------ help: make this binding mutable: `mut some_integer`
  ...
  222 |    some_integer += 10;
      |    ^^^^^^^^^^^^^^^^^^ cannot assign to immutable argument
  ********/
  //some_integer += 10;
} // Here, some_integer goes out of scope. Nothing special happens.
```

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

  println!("{}", s1);

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

  let s2 = takes_and_gives_back(s2); // s2 is moved into
                                     // takes_and_gives_back, which also
                                     // moves its return value into s3
  println!("{}", s2);
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was
 // moved, so nothing happens. s1 goes out of scope and is dropped.
```

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

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

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

```rust
// takes_and_gives_back will take a String and return one
fn takes_and_gives_back(a_string: String) -> String {
  // a_string comes into
  // scope

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

Multiple Return

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

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

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

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

  (s, length)
}
```

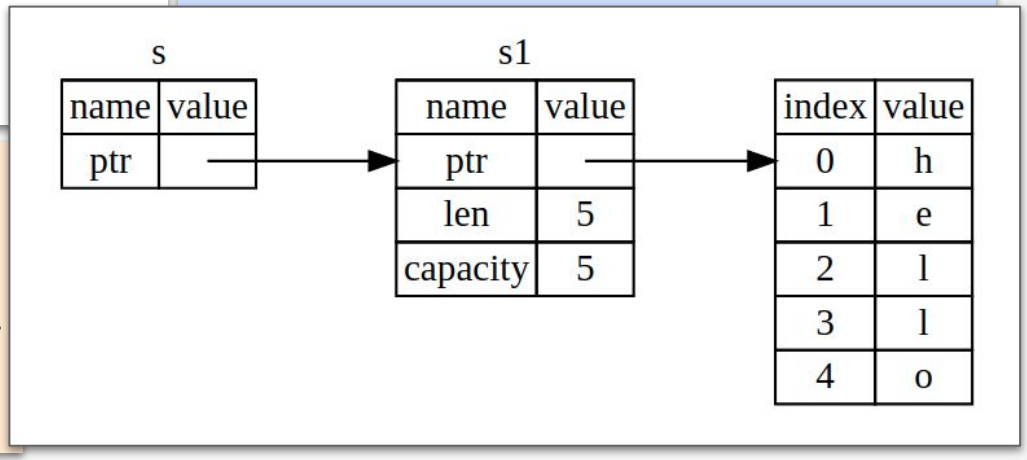
Borrowing

Send a reference of the object to the calling function can avoid transfer of ownership twice if the callsite function wants to keep it.

```rust
fn code15() {
  let s1 = String::from("hello");
  let len = calculate_length(&s1);
  println!("The length of '{}' is {}.", s1, len);
}

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

code15() pass &s1 into calculate_length() and, in its definition, we take &String rather than String.

- These ampersands are references.
  - They allow us to refer to some value without taking ownership of it.



We call having references as function parameters borrowing.

- As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back.

Continue ....

```
fn code16() {
  let s = String::from("hello");

  change(&s);
}

fn change(some_string: &String) {
  /*****************
  error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
     --> src/main.rs:320:4
      |
  319 | fn change(some_string: &String) {
      |                        ------- help: consider changing this to be a mutable reference: `&mut String`
  320 |     some_string.push_str(", world");
      |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as
mutable
  *****************/
  //some_string.push_str(", world");
}
```

Continue ....

Mutable References:

- They have one big restriction: you can have only one mutable reference to a particular piece of data in a particular scope.

```
fn code18() {
  let mut s = String::from("hello");
  let r1 = &mut s;
  /*
  error[E0499]: cannot borrow `s` as mutable more than once at a time
     --> src/main.rs:358:13
      |
  357 |     let r1 = &mut s;
      |               ------ first mutable borrow occurs here
  358 |     let r2 = &mut s;
      |               ^^^^^^ second mutable borrow occurs here
  359 |
  360 |     println!("{}, {}", r1, r2);
      |                        -- first borrow later used here
  */

  //let r2 = &mut s;
  //println!("{}, {}", r1, r2);
}
```

```
fn code17() {
  let mut s = String::from("hello");
  println!("Before: {}", s);
  change2(&mut s);
  // change2(& s);
  println!("After: {}", s);
}

fn change2(some_string: &mut String) {
  some_string.push_str(", world");
}

/*fn change2(some_string: & String) {
  some_string.push_str(", world");
}*/
```

This restriction allows for mutation but in a very controlled fashion.

Data Races

The benefit of having mutable reference restriction is that Rust can prevent data races at compile time.

- A data race is similar to a race condition.
- 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.
- Data races cause undefined behavior.
- Data races can be difficult to diagnose and fix at runtime.
  - Rust prevents the problem at compile time!

It is valid to use curly brackets to create a new scope, allowing for multiple mutable references, just not simultaneous ones.

Code

Restrictions:

- Multiple living mutable reference within a scope.
- Immutable reference uses for read would cause data races even if there is a mutable reference (not multiple).

Allows:

- Easy to create short-live scope using curly brackets.
- The last use of a reference defines its scope of lifetime. So, scope is not necessary to be within curly brackets.

```rust
fn code19() {
  let mut s = String::from("hello");

  {
     let r1 = &mut s;
  } // r1 goes out of scope here, so we can make a new reference with no problems.

  let r2 = &mut s;

  mixed();
  mixedScope();
}
```

You may now start hating Rust 😅

Continue ....

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

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

  /*
  error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
     --> src/main.rs:396:13
      |
  394 |     let r1 = &s; // no problem
      |              -- immutable borrow occurs here
  395 |     let r2 = &s; // no problem
  396 |     let r3 = &mut s; // BIG PROBLEM
      |              ^^^^^^ mutable borrow occurs here
  */
  // let r3 = &mut s;

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

Continue ....

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

  let r1 = &s; // no problem
  let r2 = &s; // no problem
  println!("{} and {}", r1, r2);
  // r1 and r2 are no longer used after this point

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

Dangling References

A dangling pointer is a pointer that references a location in memory that may have been given to someone else, by freeing some memory while preserving a pointer to that memory.

In Rust, the compiler guarantees that references will never be dangling references.

- If we have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

Code

```
/*
error[E0106]: missing lifetime specifier
  --> src/main.rs:433:16
   |
433 | fn dangle() -> &String {
   |                ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
help: consider using the `'static` lifetime
   |
433 | fn dangle() -> &'static String {
*/

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!
```

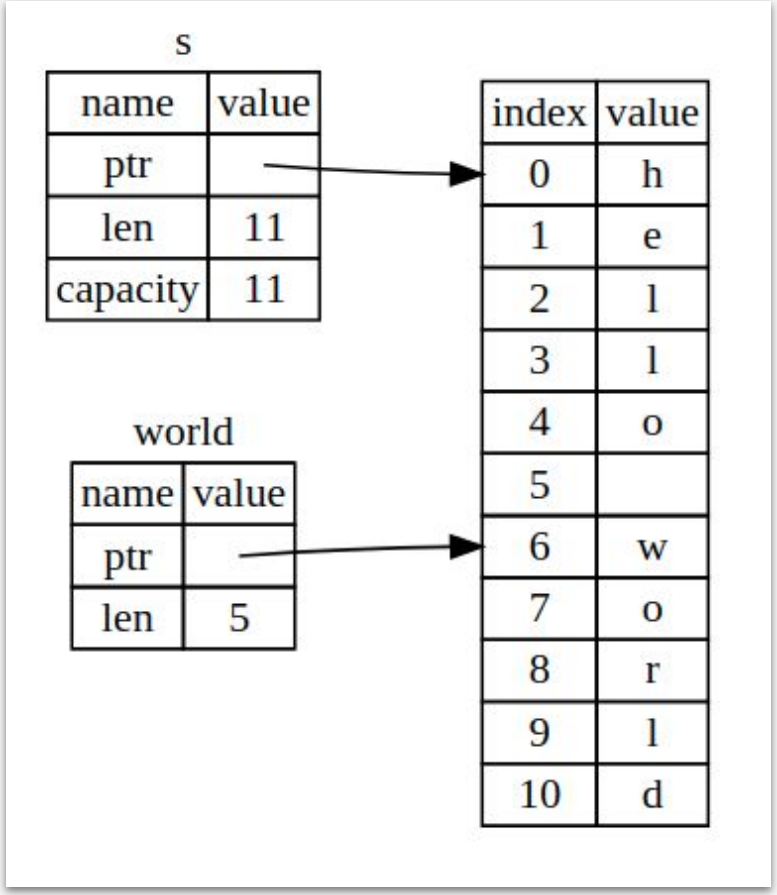
Continue ....

```rust
fn code20() {
  //let reference_to_nothing = dangle();
  let reference_to_nothing = no_dangle();
}
```

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


  s
}
```

Ownership is moved out, and nothing is deallocated.

Slices [String]




```rust
fn code21() {
  let mut s = String::from("hello world");
  let len = s.len();

  let hello = &s[0..5];
  // let hello = &s[..5];
  let world = &s[6..len];
  // let world = &s[6..];
}
```

```rust
fn code21() {
  let mut s = String::from("hello world");

  let word = first_word(&s);
  /*****
     error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
     --> src/main.rs:581:4
      |
  580 |     let word = first_word(&s);
      |                            -- immutable borrow occurs here
  581 |     s.clear();
      |     ^^^^^^^^^ mutable borrow occurs here
  582 |     println!("{} => {}", s, word);
      |                              ---- immutable borrow later used here
     ******/
  //s.clear();
  println!("{} => {}", s, word);

  let my_string_literal = "hello world";

  // Because string literals *are* string slices already,
  // this works too, without the slice syntax!
  let word = first_word_str(my_string_literal);
  println!("{} => {}", s, word);
}
```

```rust
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[..]
}
```

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

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

  &s[..]
}
```

< Ownership />