PROGRAMMING PARADIGMS

CS 3022 – Rust continued

This is an unsafe language you have to use Rust instead
NICE TEMPLATE <BRO>
All your programs are gonna be rewritten in Rust anyways you'll be jobless
NICE POINTER *BRO
CONSTEXPR
QUIT HAVING FUN
imgflip.com

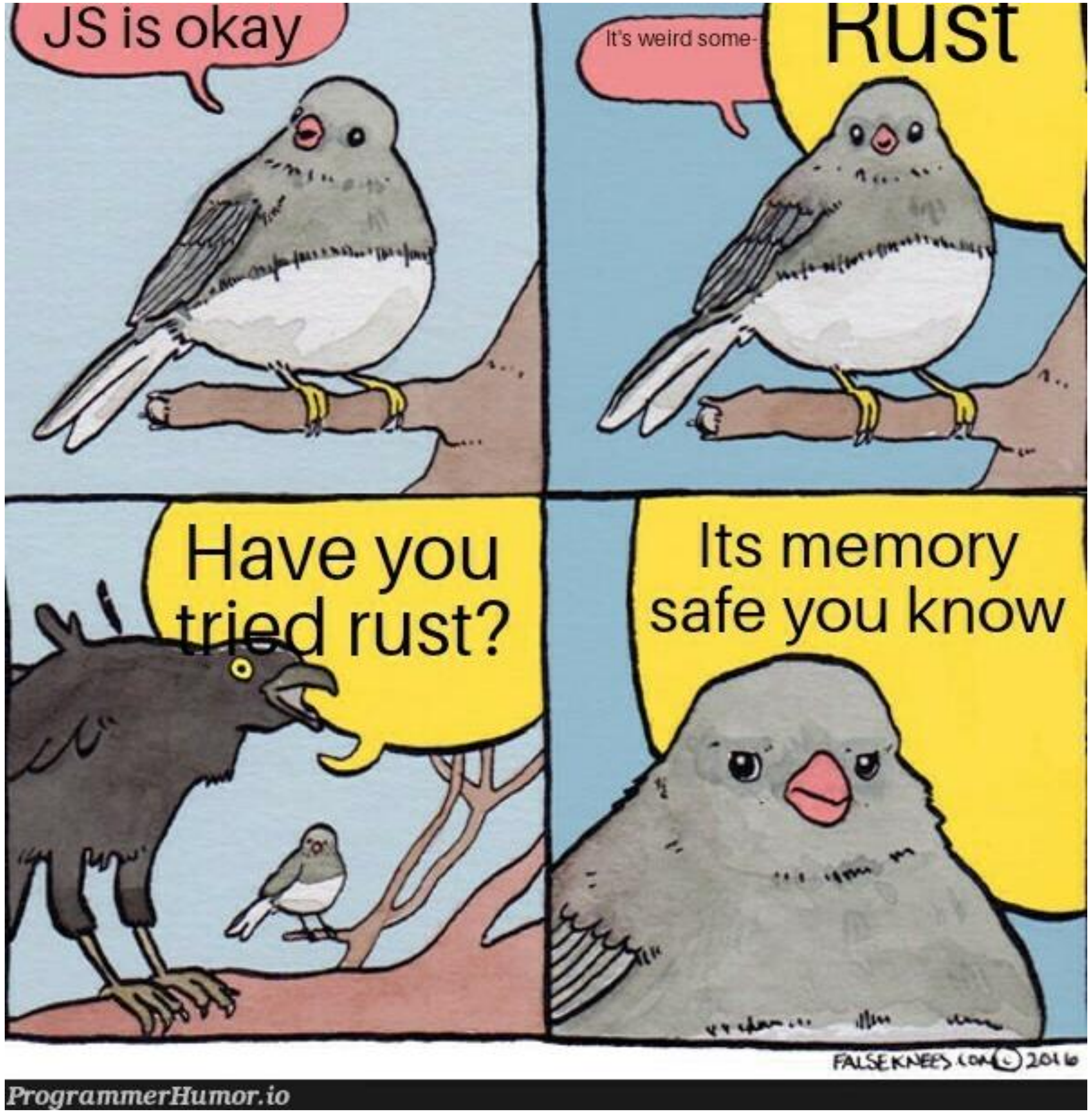
MEMORY SAFETY RULES IN RUST

* Each value in Rust has an *owner*.

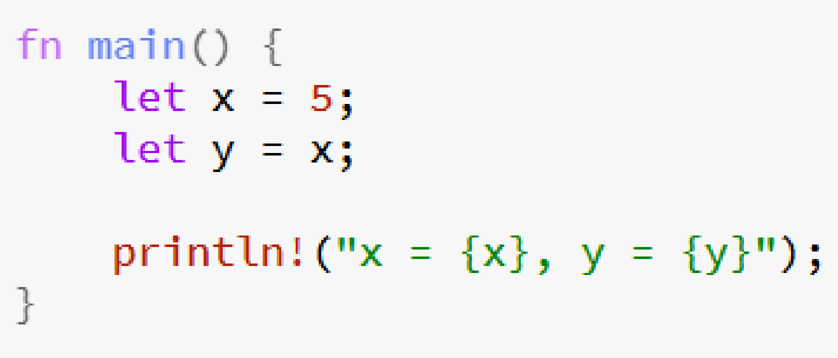
* There can only be one owner at a time.

* When the owner goes out of scope, the value will be dropped.

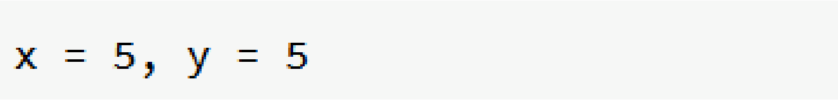
These rules can be annoying, but we don't need a garbage collector



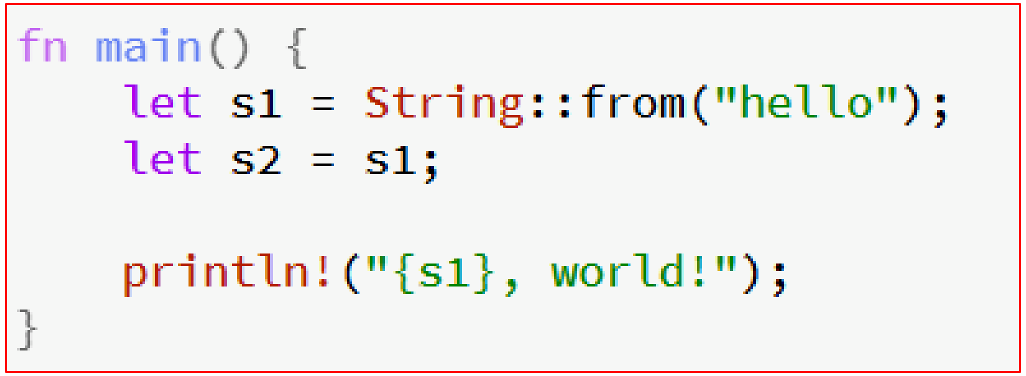
MEMORY RULES IN ACTION



```rust
fn main() {
    let x = 5;
    let y = x;

    println!("x = {x}, y = {y}");
}
```



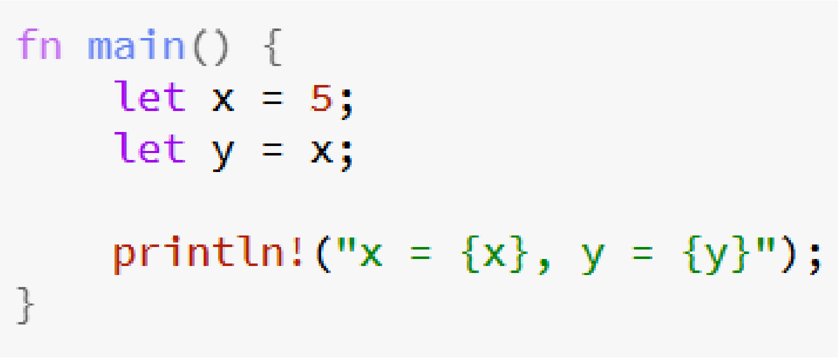
```
x = 5, y = 5
```



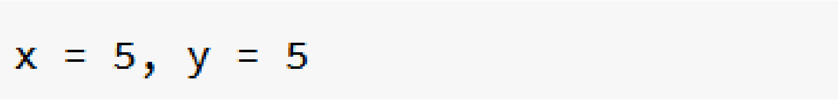
```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{s1}, world!");
}
```

ERROR! Huh? Whats going on?

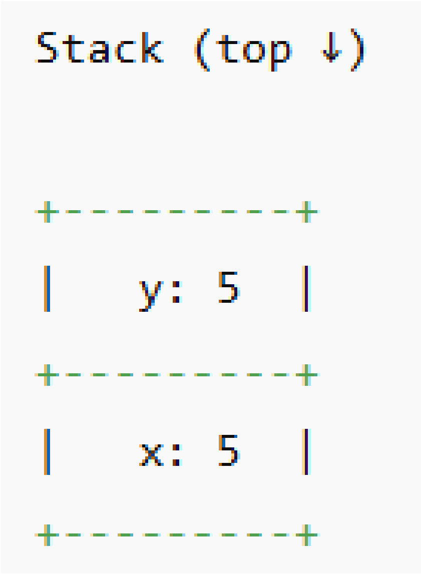
MEMORY RULES IN ACTION – STACK STORAGE



```rust
fn main() {
    let x = 5;
    let y = x;

    println!("x = {x}, y = {y}");
}
```

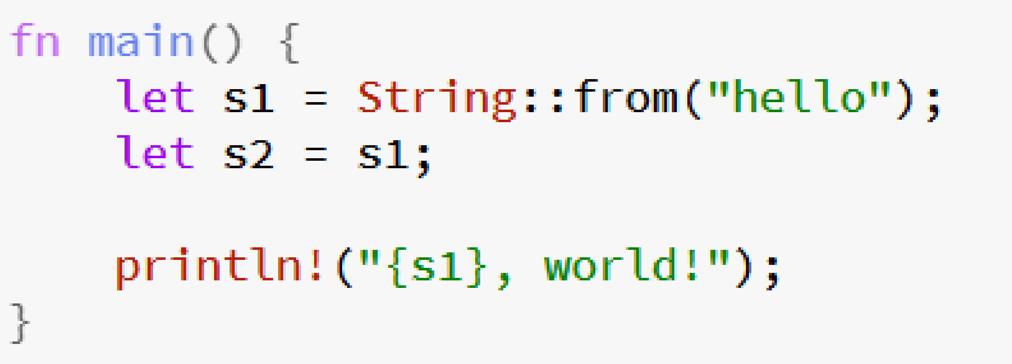


```
x = 5, y = 5
```

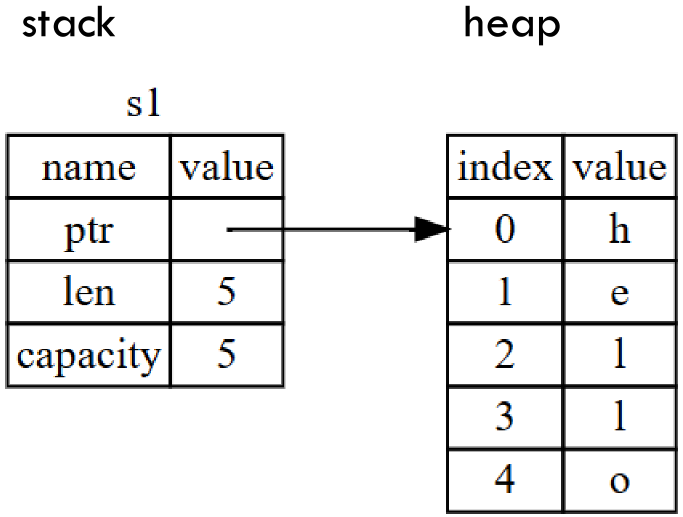


```
Stack (top ↓)


+----------+
|   y: 5   |
+----------+
|   x: 5   |
+----------+
```

MEMORY RULES IN ACTION – HEAP STORAGE



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

    println!("{s1}, world!");
}
```



stack

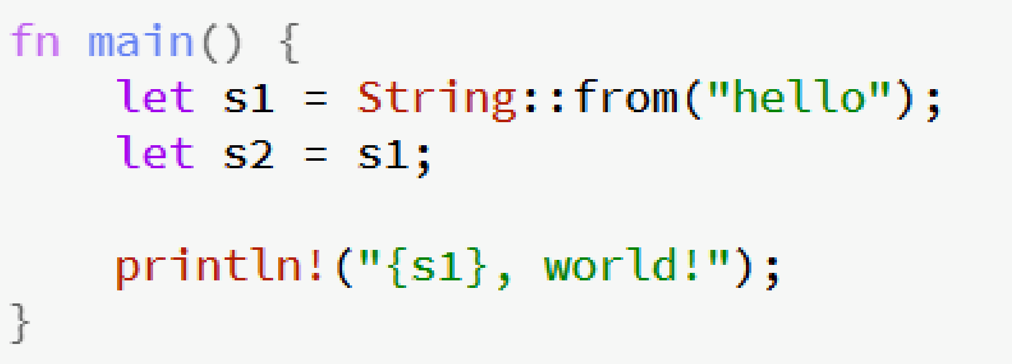
s1

name	value
ptr	
len	5
capacity	5

heap

index	value
0	h
1	e
2	l
3	l
4	o

MEMORY RULES IN ACTION – HEAP STORAGE

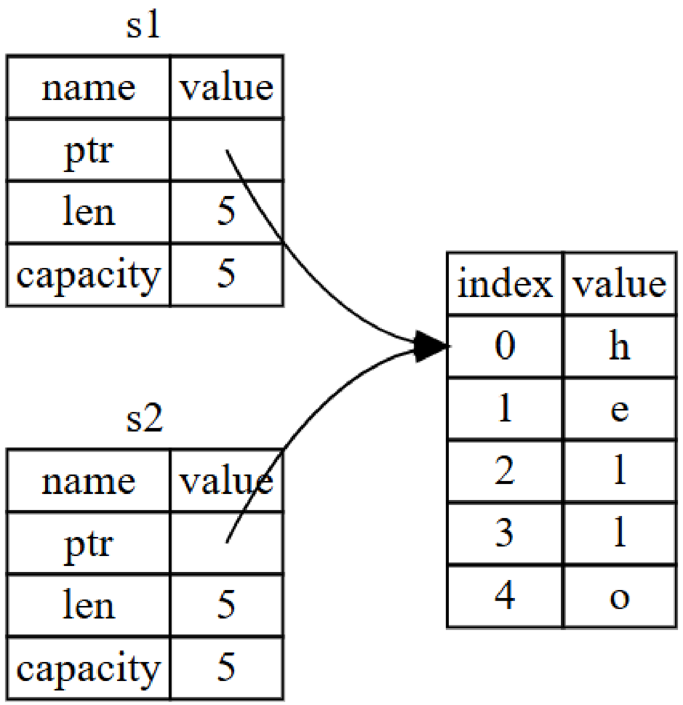


```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{s1}, world!");
}
```

What happens in many programming languages

stack

heap



s1

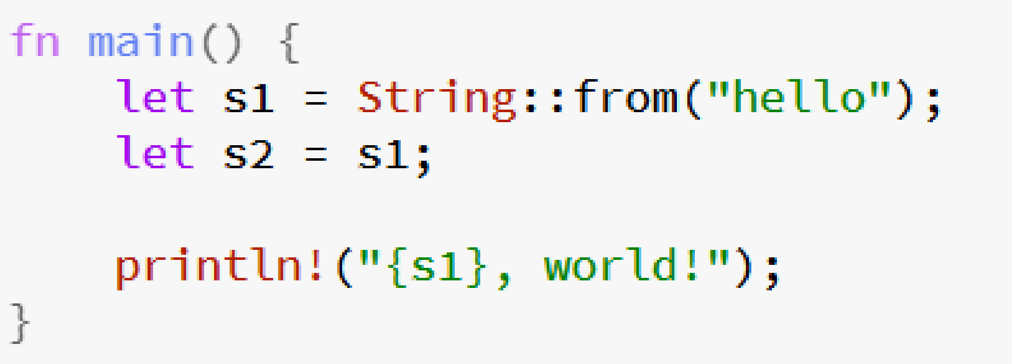
name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

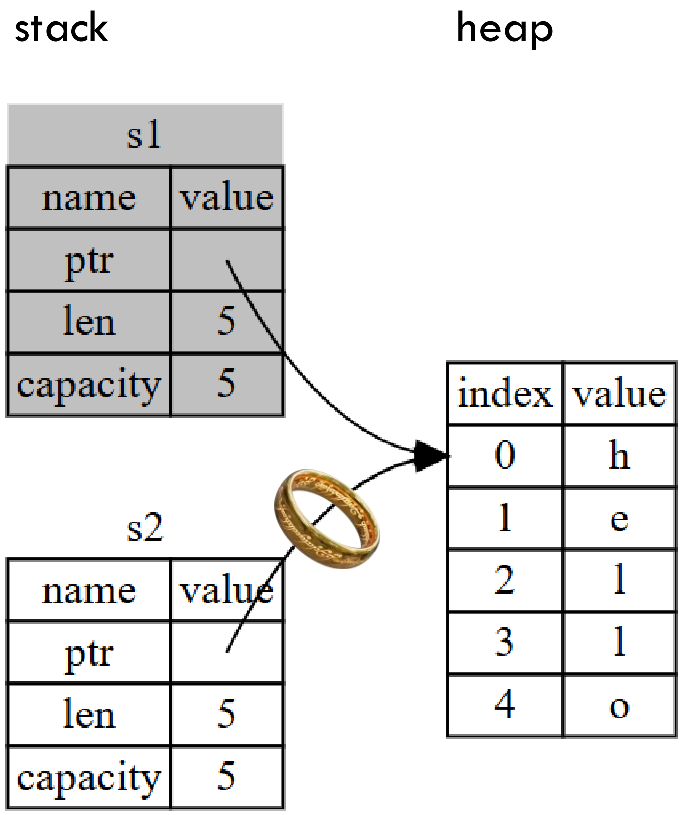
index	value
0	h
1	e
2	l
3	l
4	o

ONE POINTER TO RULE THEM ALL …



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

    println!("{s1}, world!");
}
```

What happens in Rust. s1 invalidated!



s1

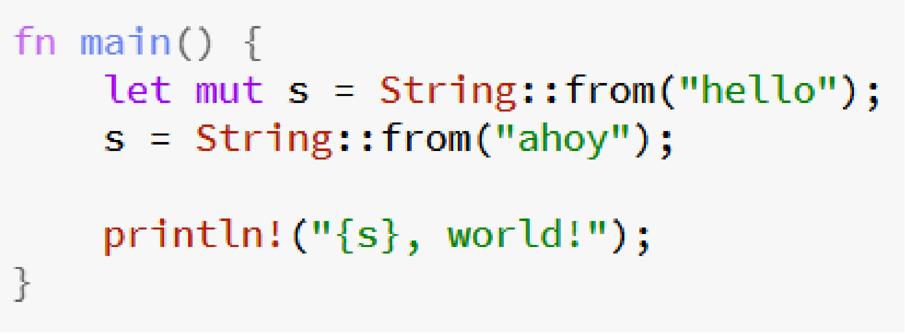
name	value
ptr	
len	5
capacity	5

s2

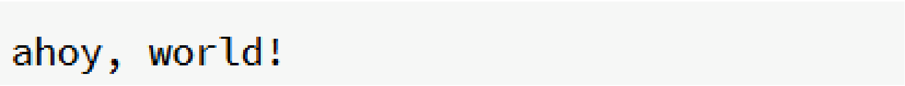
name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

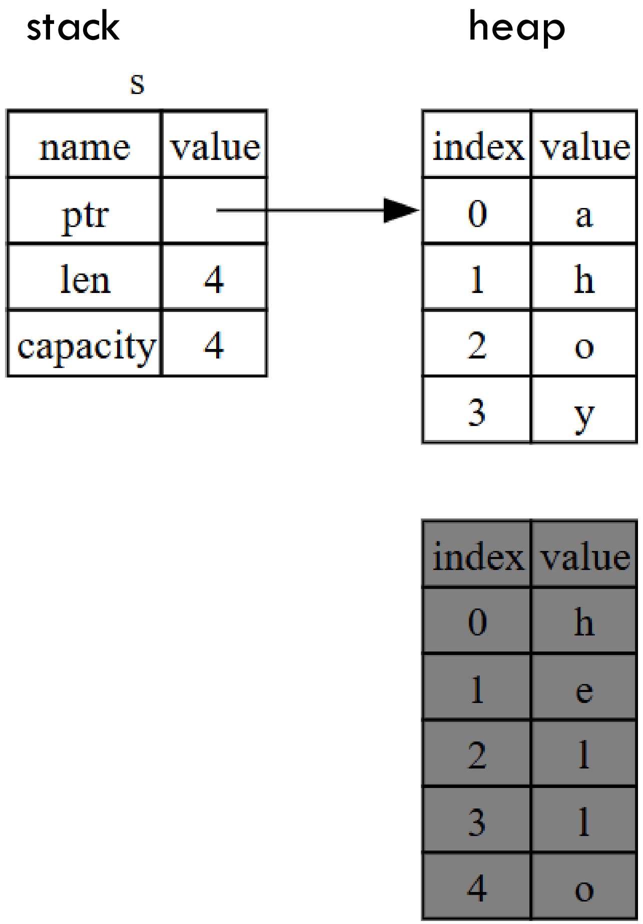
REASSIGNMENT TRIGGERS IMMEDIATE FREEING OF MEMORY



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

    println!("{s}, world!");
}
```



```
ahoy, world!
```



stack

heap

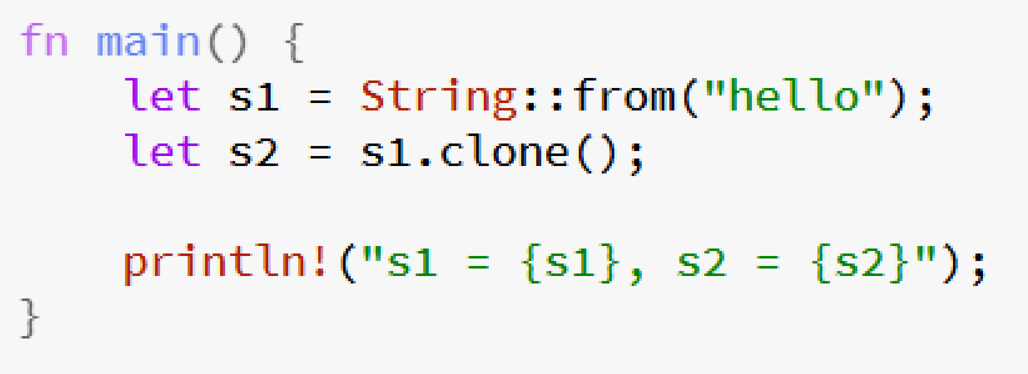
s

name	value
ptr	
len	4
capacity	4

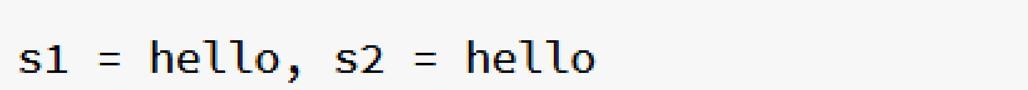
index	value
0	a
1	h
2	o
3	y

index	value
0	h
1	e
2	l
3	l
4	o

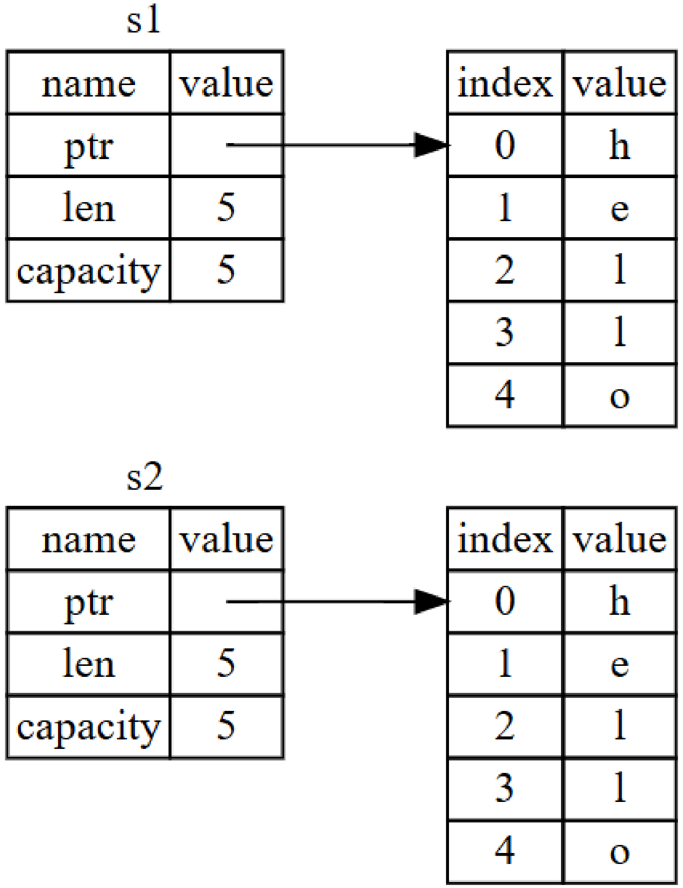
DEEP COPY TO FIX IT– COPY MADE ON THE HEAP



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

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



```
s1 = hello, s2 = hello
```



s1

name	value
ptr	
len	5
capacity	5

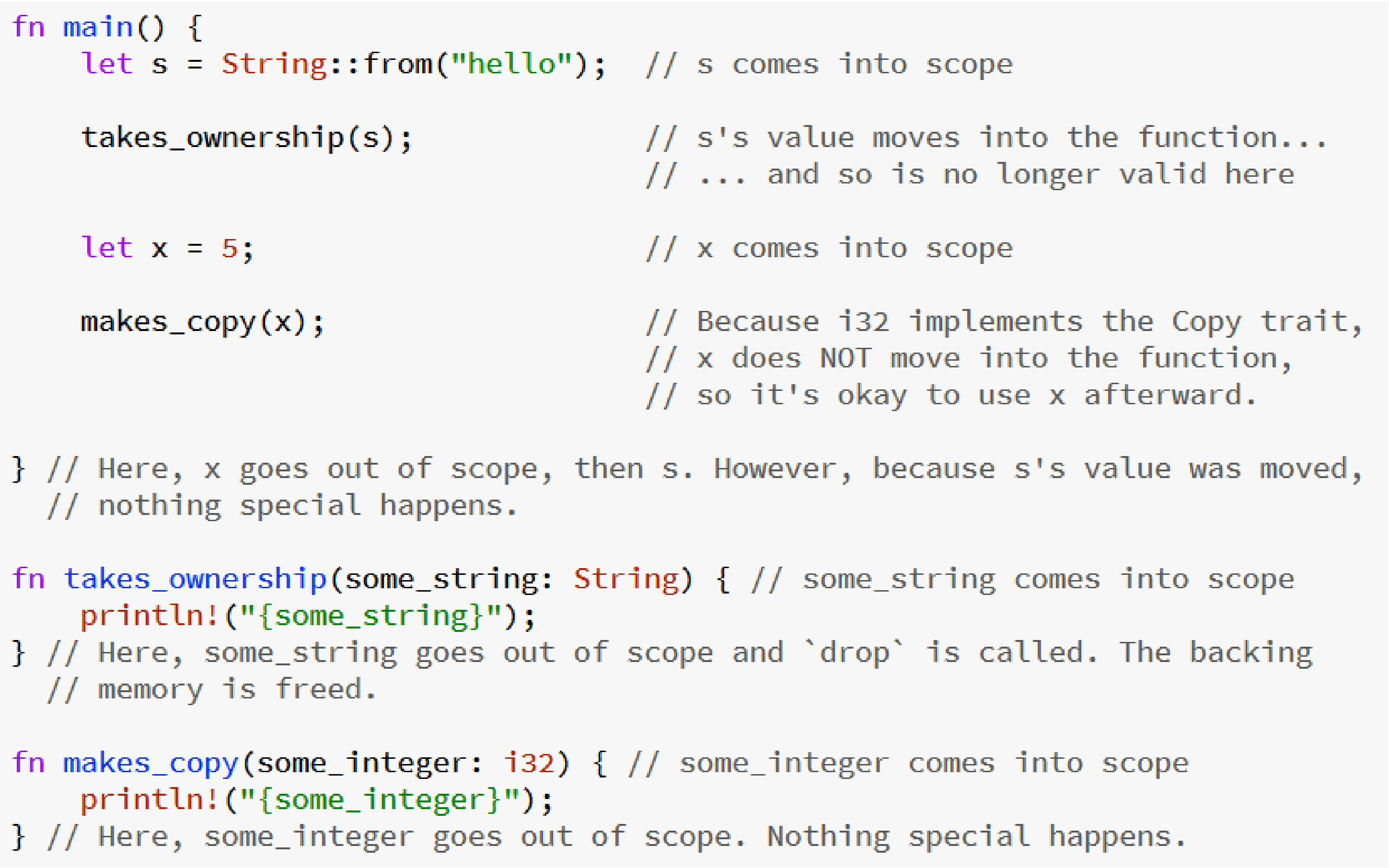
index	value
0	h
1	e
2	l
3	l
4	o

s2

name	value
ptr	
len	5
capacity	5

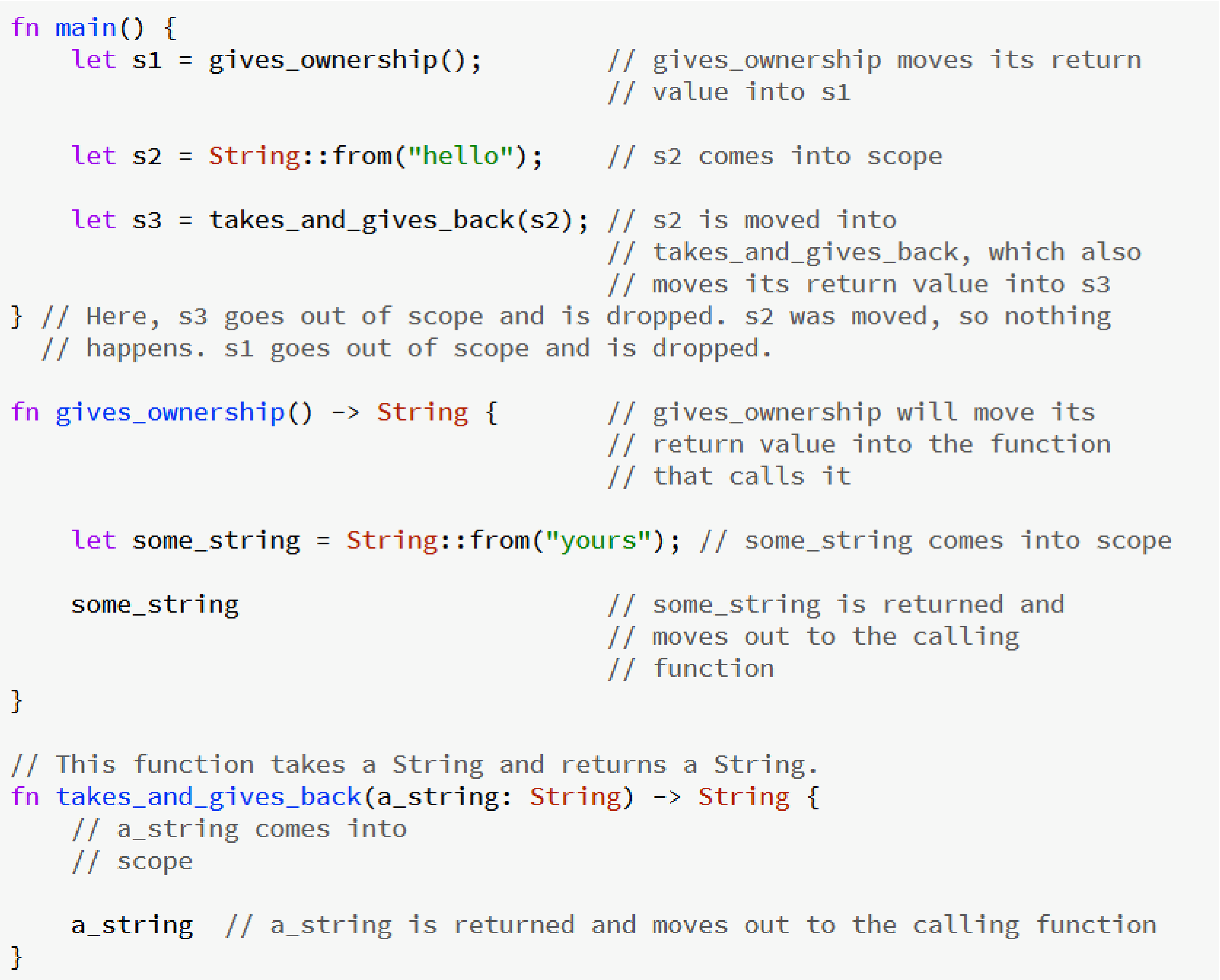
index	value
0	h
1	e
2	l
3	l
4	o

FUNCTIONS, SCOPE, AND OWNERSHIP



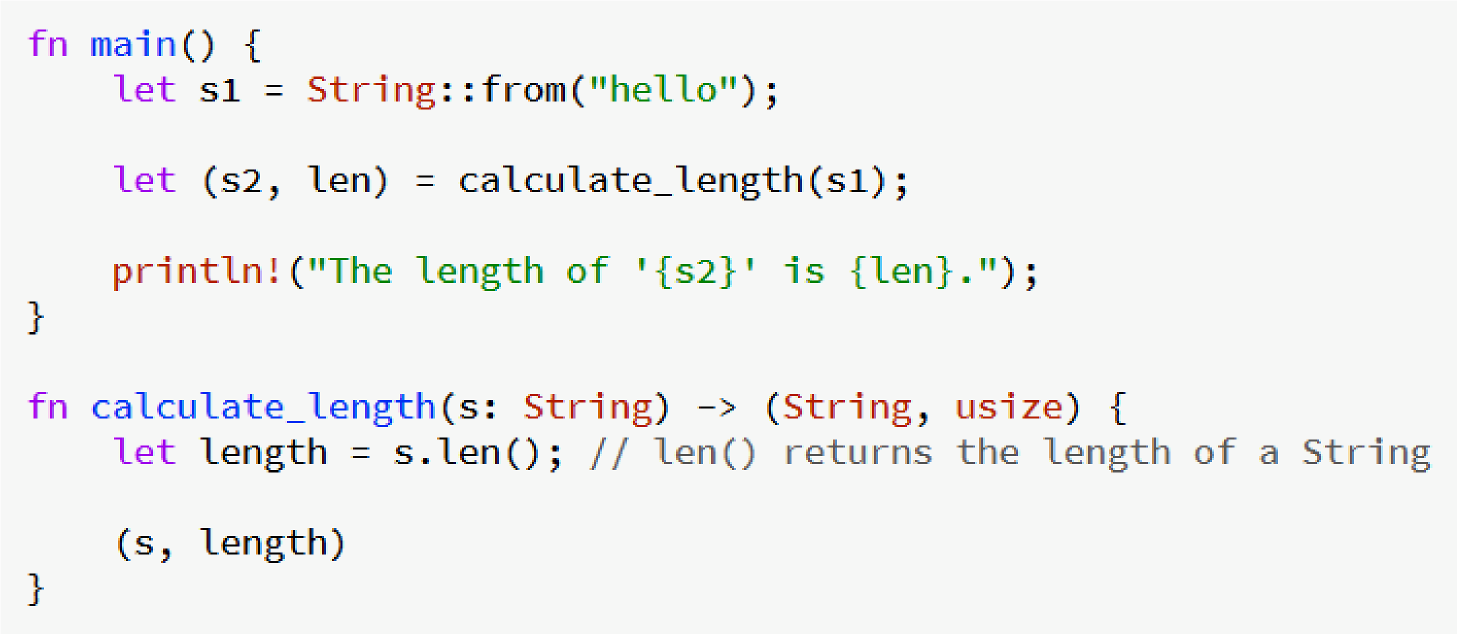
```
fn main() {
    let s = String::from("hello");  // s comes into scope

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

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // Because i32 implements the Copy trait,
                                    // x does NOT move into the function,
                                    // so it's okay to use x afterward.

} // Here, x goes out of scope, then s. However, because s's value was moved,
  // nothing special happens.

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.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.
```

OWNERSHIP WITH RETURN VALUES

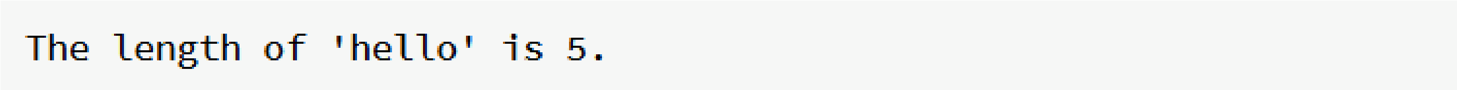


```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

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

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, 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 {        // 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
}

// This function takes a String and returns a String.
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
}
```

RETURNING TUPLES FOR RE-USE



```rust
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)
}
```



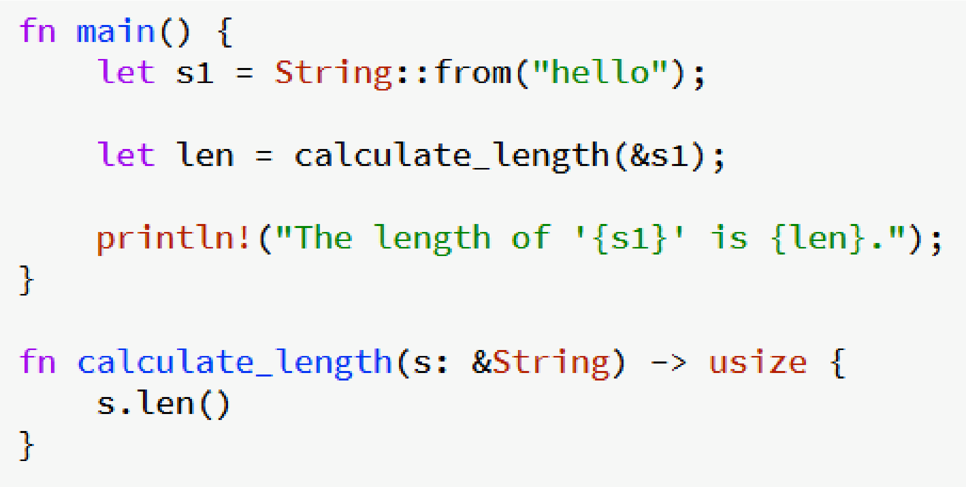
```
The length of 'hello' is 5.
```

What if you want to use a variable after passing it to a function?

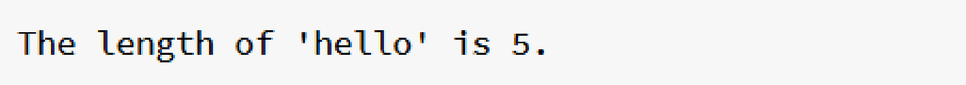
You could use a tuple.

That could get annoying.

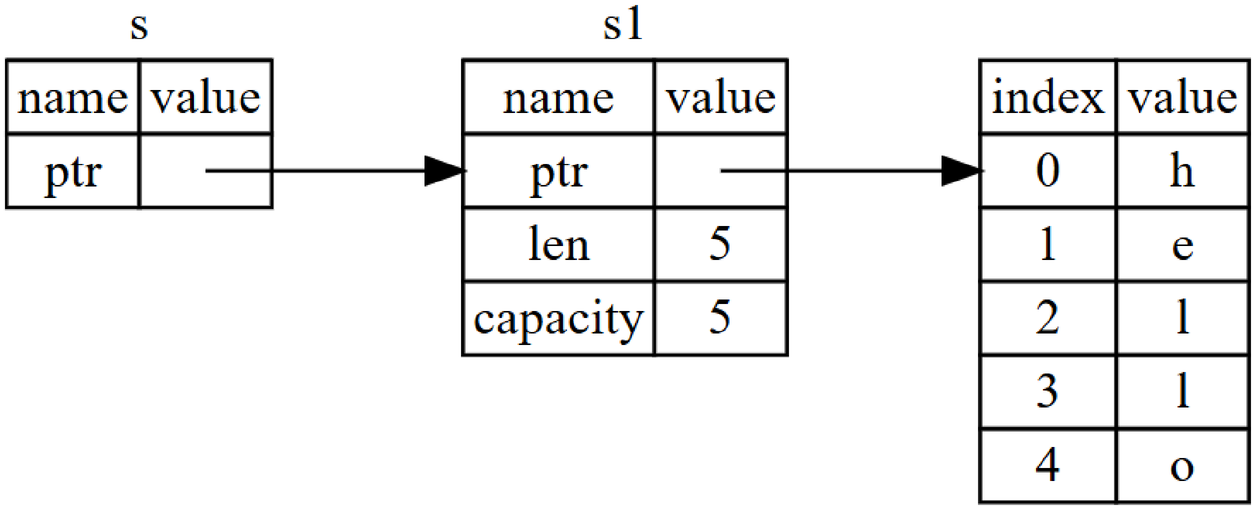
...A BETTER WAY – REFERENCE PASSING



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

    let len = calculate_length(&s1);

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

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



```
The length of 'hello' is 5.
```



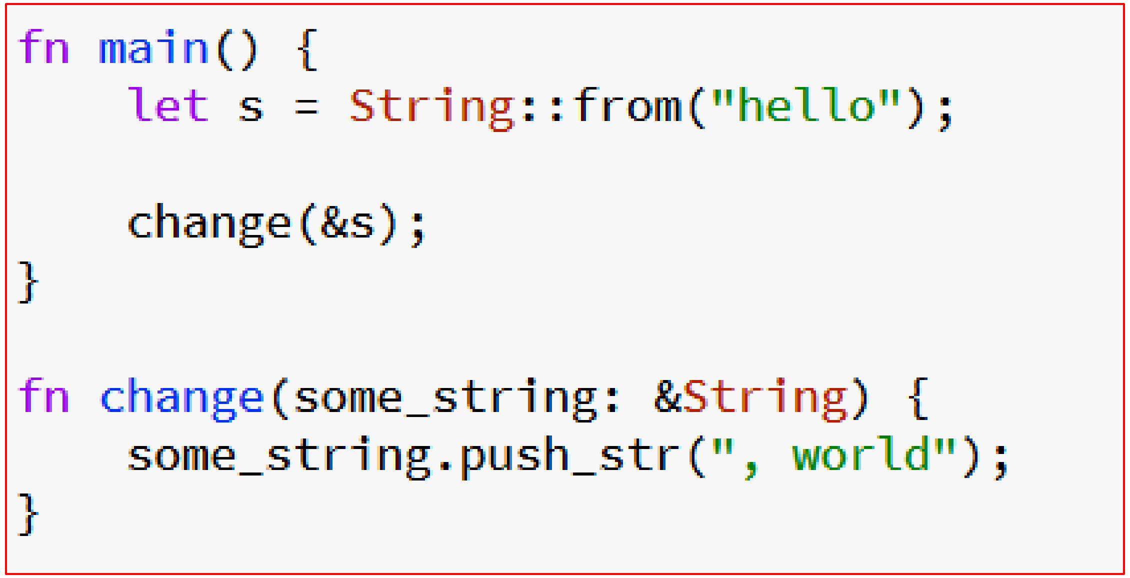
name	value
ptr	

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

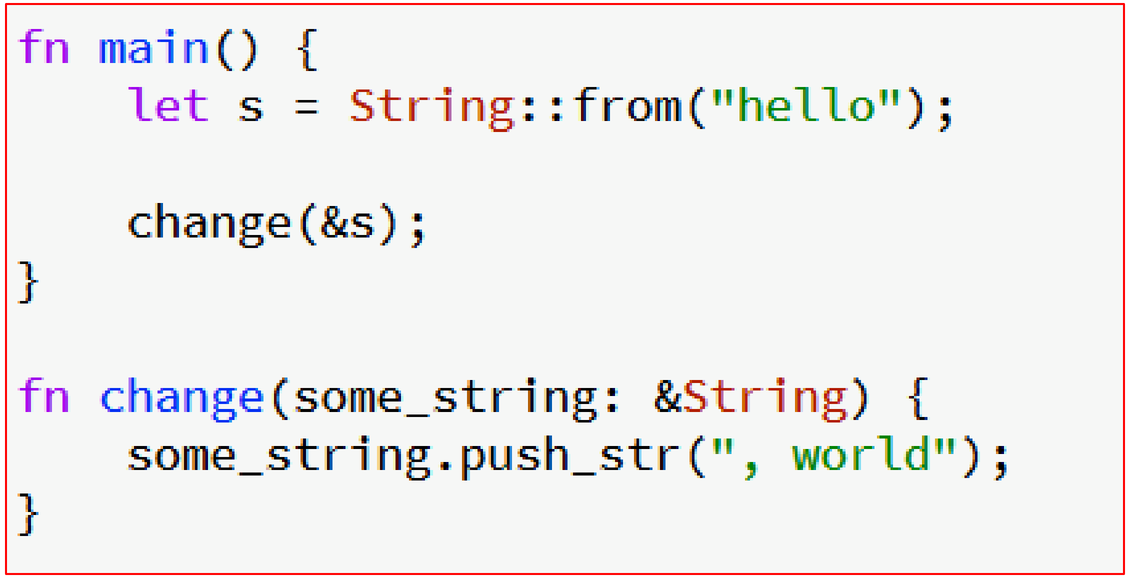
Note the automatic de-referencing (deref coercion). (*s).len() is the same as s.len(). Not all types have this ‘trait’.

WHAT IS THE ERROR IN THIS CODE?



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

    change(&s);
}

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

WHAT IS THE ERROR IN THIS CODE?

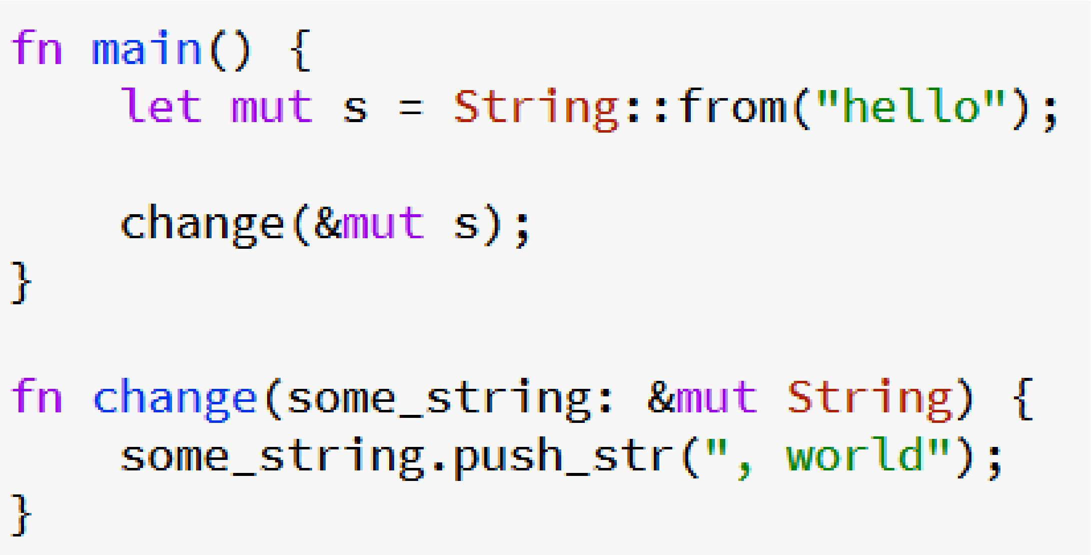


```
fn main() {
    let s = String::from("hello");

    change(&s);
}

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

There are two errors:

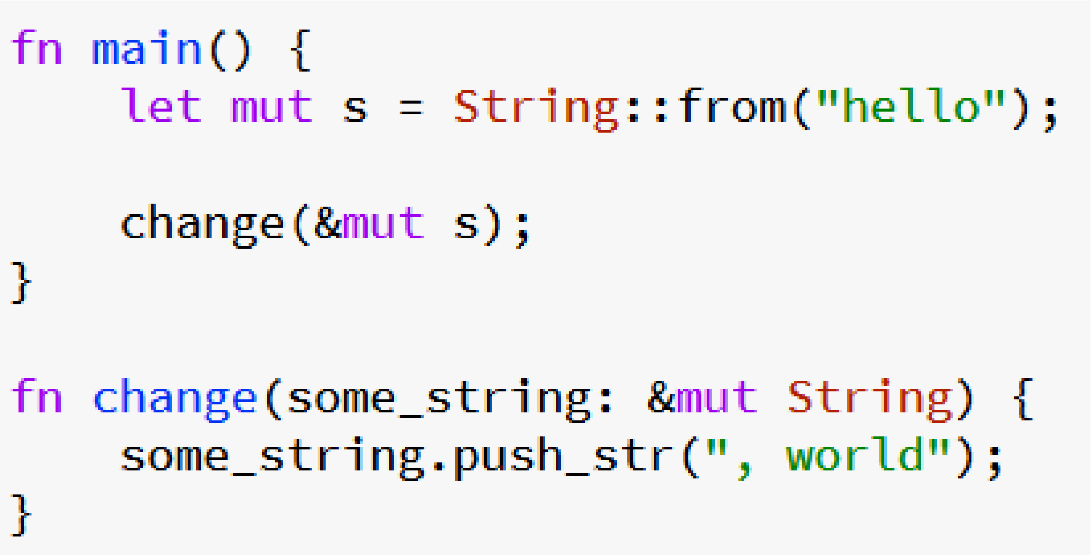
1. Variable ‘s’ is not mutable.
2. The reference ‘some_string’ is not mutable.

THE FIX? MUTABLE REFERENCES



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

    change(&mut s);
}

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

THE FIX? MUTABLE REFERENCES



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

    change(&mut s);
}

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

SIDEBAR – RACE CONDITIONS

Race condition - Any bug where the program's *behavior depends on the timing or order of events.*

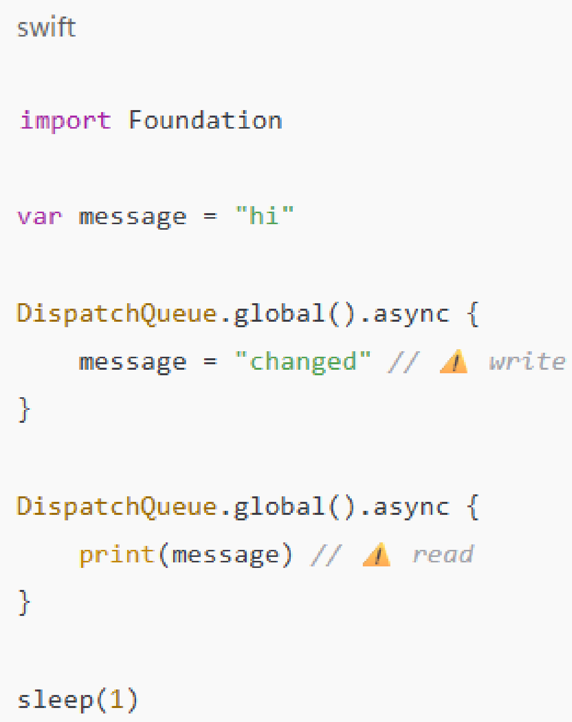
A *data race* happens when these three behaviors occur:

- 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.

Data races cause ***undefined*** behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime

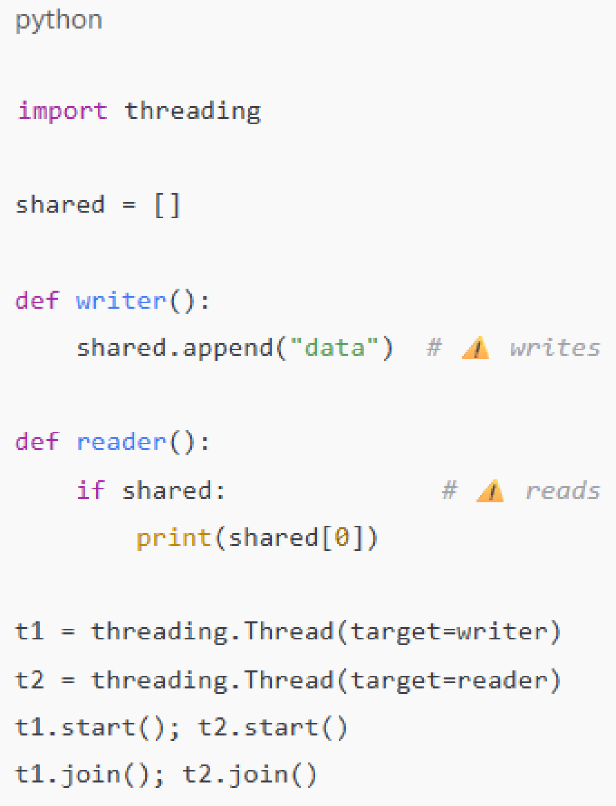
SIDEBAR – RACE CONDITIONS

Race condition - Any bug where the program's *behavior depends on the timing or order of events.*



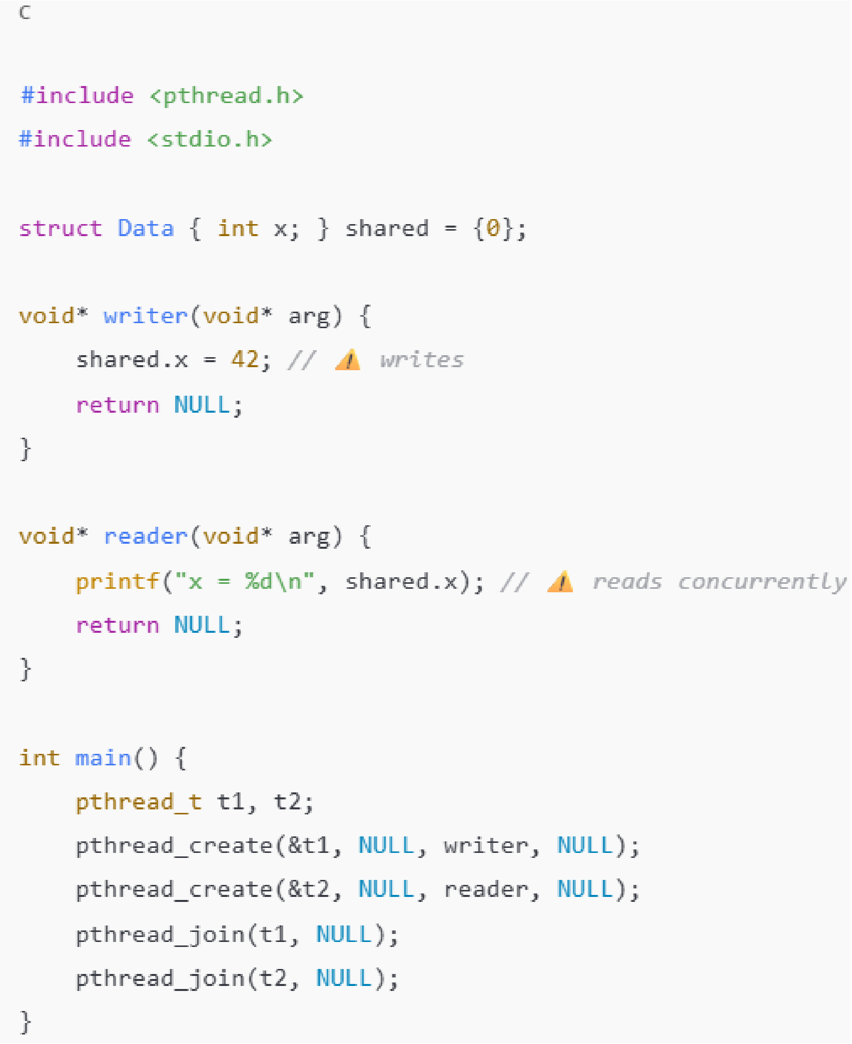
swift

```swift
import Foundation

var message = "hi"

DispatchQueue.global().async {
    message = "changed" // ⚠ write
}

DispatchQueue.global().async {
    print(message) // ⚠ read
}

sleep(1)
```



python

```python
import threading

shared = []

def writer():
    shared.append("data")  # ⚠ writes

def reader():
    if shared:             # ⚠ reads
        print(shared[0])

t1 = threading.Thread(target=writer)
t2 = threading.Thread(target=reader)
t1.start(); t2.start()
t1.join(); t2.join()
```



c

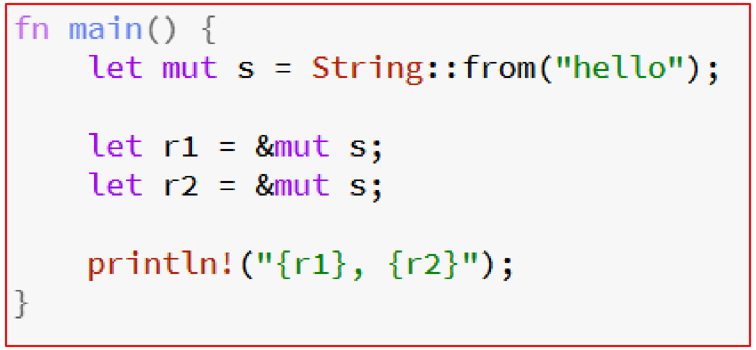
```c
#include <pthread.h>
#include <stdio.h>

struct Data { int x; } shared = {0};

void* writer(void* arg) {
    shared.x = 42; // ⚠ writes
    return NULL;
}

void* reader(void* arg) {
    printf("x = %d\n", shared.x); // ⚠ reads concurrently
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, writer, NULL);
    pthread_create(&t2, NULL, reader, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

RUST MEMORY RULES PREVENT DATA RACES

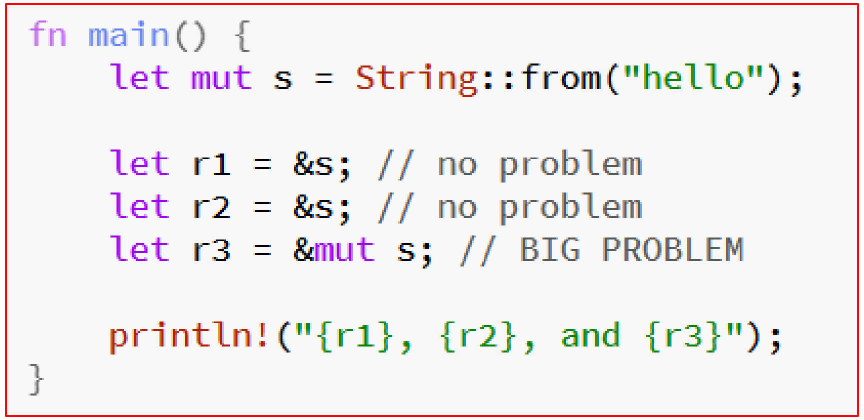
Rules:

- At any given time, you can have either
    - One mutable reference
    - Any number of immutable references.
- References must always be valid.

Bad code! These examples will not compile.



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

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

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



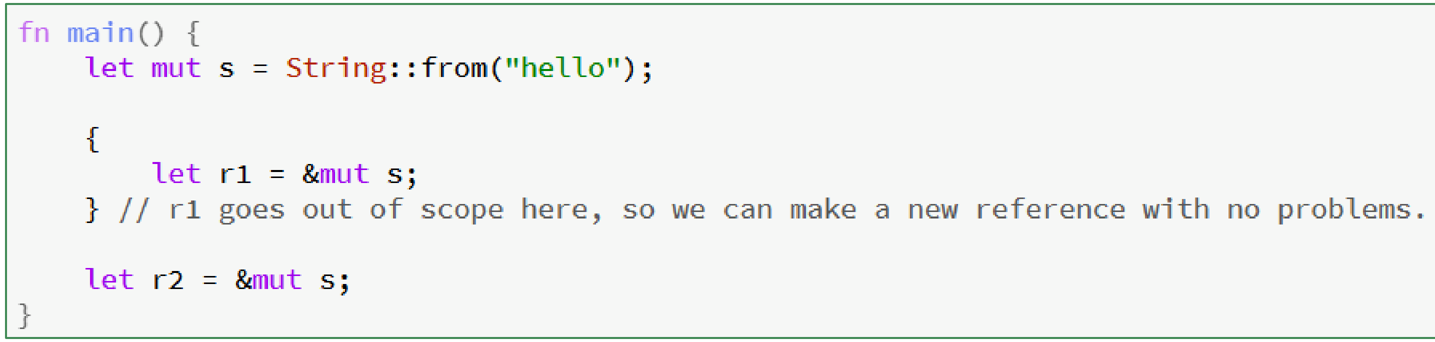
```rust
fn main() {
    let mut s = String::from("hello");

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

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

RUST MEMORY RULES PREVENT DATA RACES

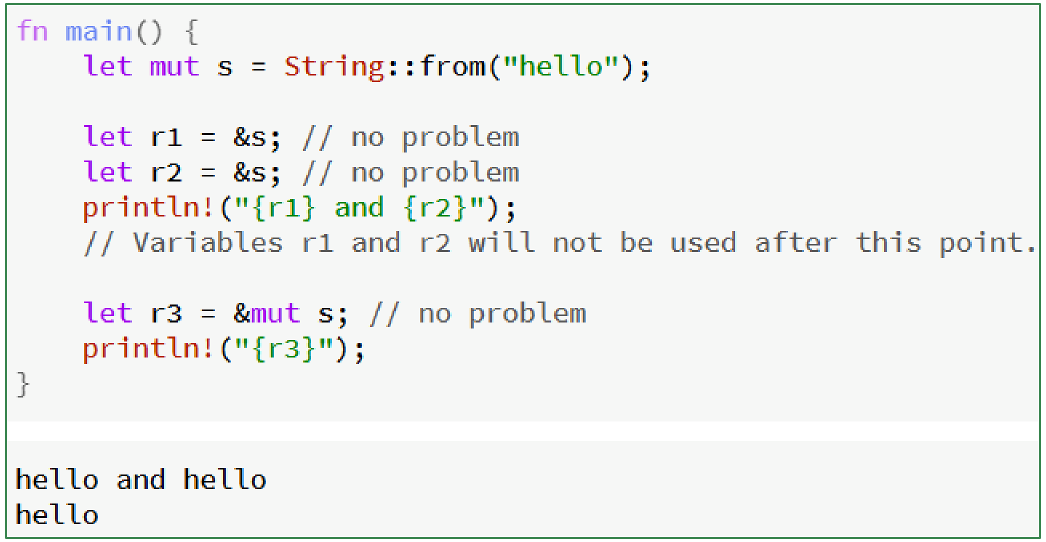
Rules:

- At any given time, you can have either
  - One mutable reference
  - Any number of immutable references.
- References must always be valid.

Good code! These examples will compile.



```
fn main() {
    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;
}
```



```
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}");
}
```

```
hello and hello
hello
```

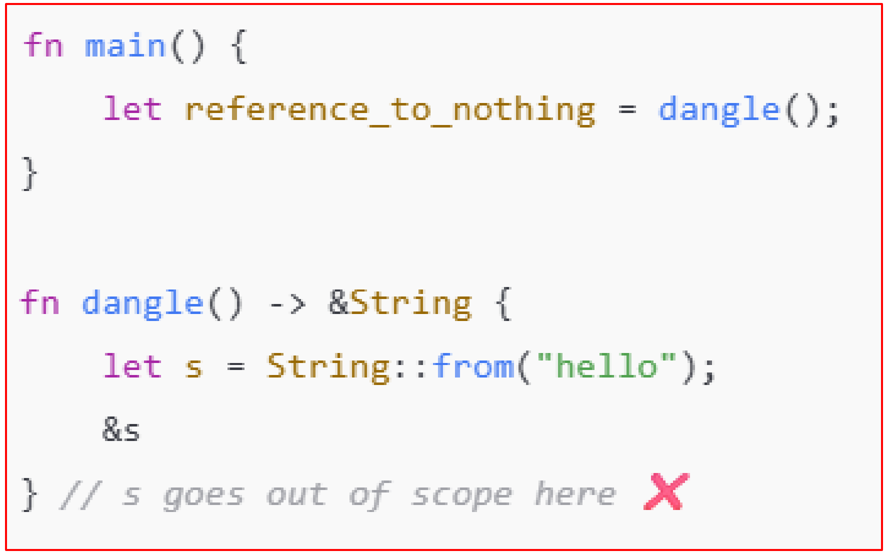
DANGLING REFERENCES

Because s goes out of scope and memory is released,

`reference_to_nothing`

points to nothing.

Error: Will not compile

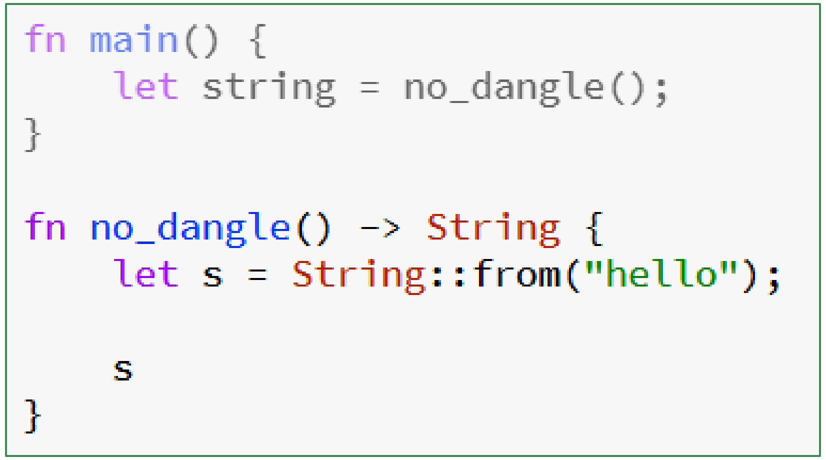


```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
} // s goes out of scope here ✗
```

DANGLING REFERENCES – A BETTER WAY

Ownership of ‘s’ is moved out, and nothing is deallocated.

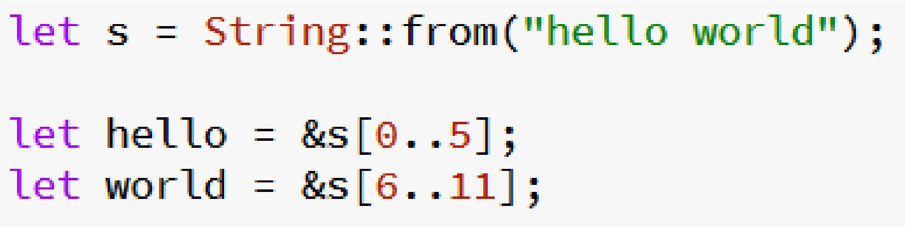
Will compile ☺



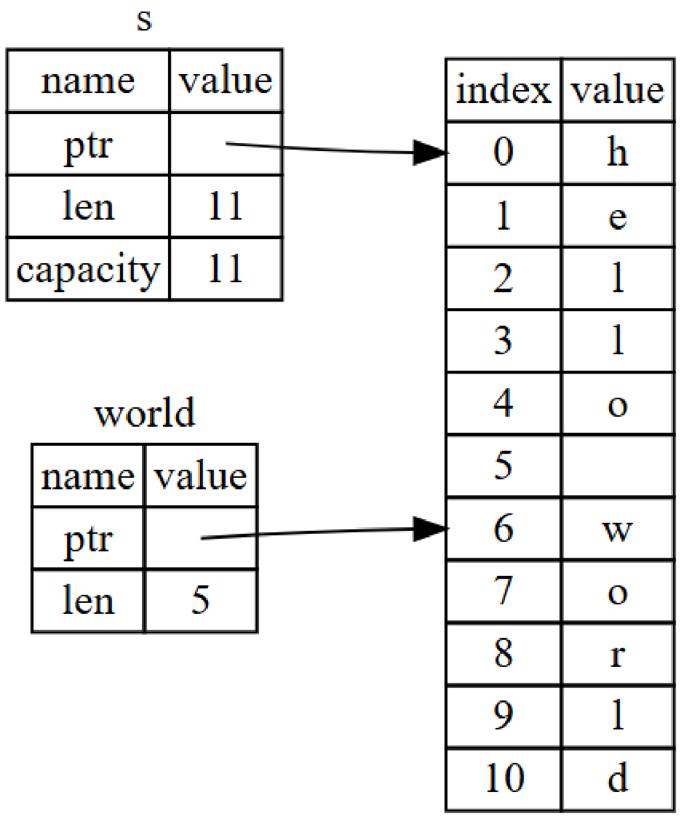
```
fn main() {
    let string = no_dangle();
}

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

    s
}
```

SLICES

A *slice* is a reference to a contiguous sequence of the elements.



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

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



s

name	value
ptr	
len	11
capacity	11

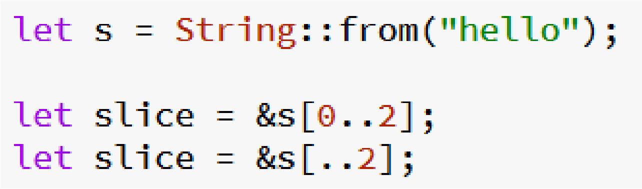
world

name	value
ptr	
len	5

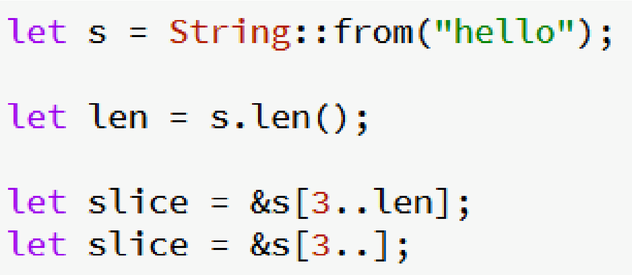
index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

SLICES – SYNTACTIC SUGAR

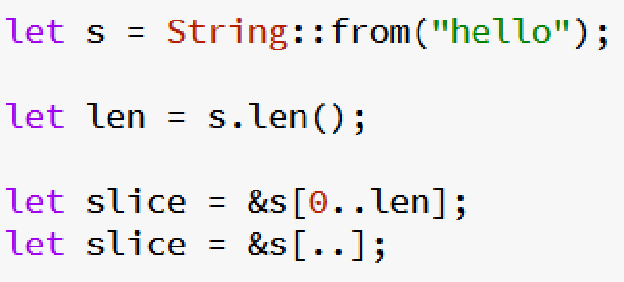
These examples have identical ways of taking slices:



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

let slice = &s[0..2];
let slice = &s[..2];
```



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

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```



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

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```