CS 241: Systems Programming
Lecture 13. Slices

Fall 2023

Prof. Stephen Checkoway

String slices

String slices are a reference to a portion of a string

```
fn main() {
    let hello_world = String::from("hello world");
    let hi: &str = &hello_world[1..5];
    println!("{hi}");
}
```

```
Output:
ello
```

&str

Previously, we said &str was a reference to a string which is true, but it it's actually a reference to a portion of a string!

String literals are actually slices

```
let foo: &str = "This is a string literal";
```

&String -> &str

Rust will convert &String into &str automatically

```rust
let s = String::from("asdf");
let slice: &str = &s;
```

Passing strings to functions

```
fn foo(arg: String) {}
fn bar(arg: &str) {}

fn main() {
    let s = String::from("abc");
    foo(s);      // Valid, moves s into foo
    foo("abc"); // Invalid, foo() expects a String

    let t = String::from("xyz");
    bar(&t);     // Automatic conversion from &String to &str
    bar("xyz"); // Valid
}
```

Given a function

```rust
fn foo(s1: &str, s2: &str) { }
```

and some variables

```rust
let x = String::from("abc");
let y = "xyz";
```

What is the right way to pass x and y to foo()?

A. `foo(&x, &y)`

B. `foo(&x, y)`

C. `foo(x, &y)`

D. `foo(x, y)`

Many string methods defined on &str

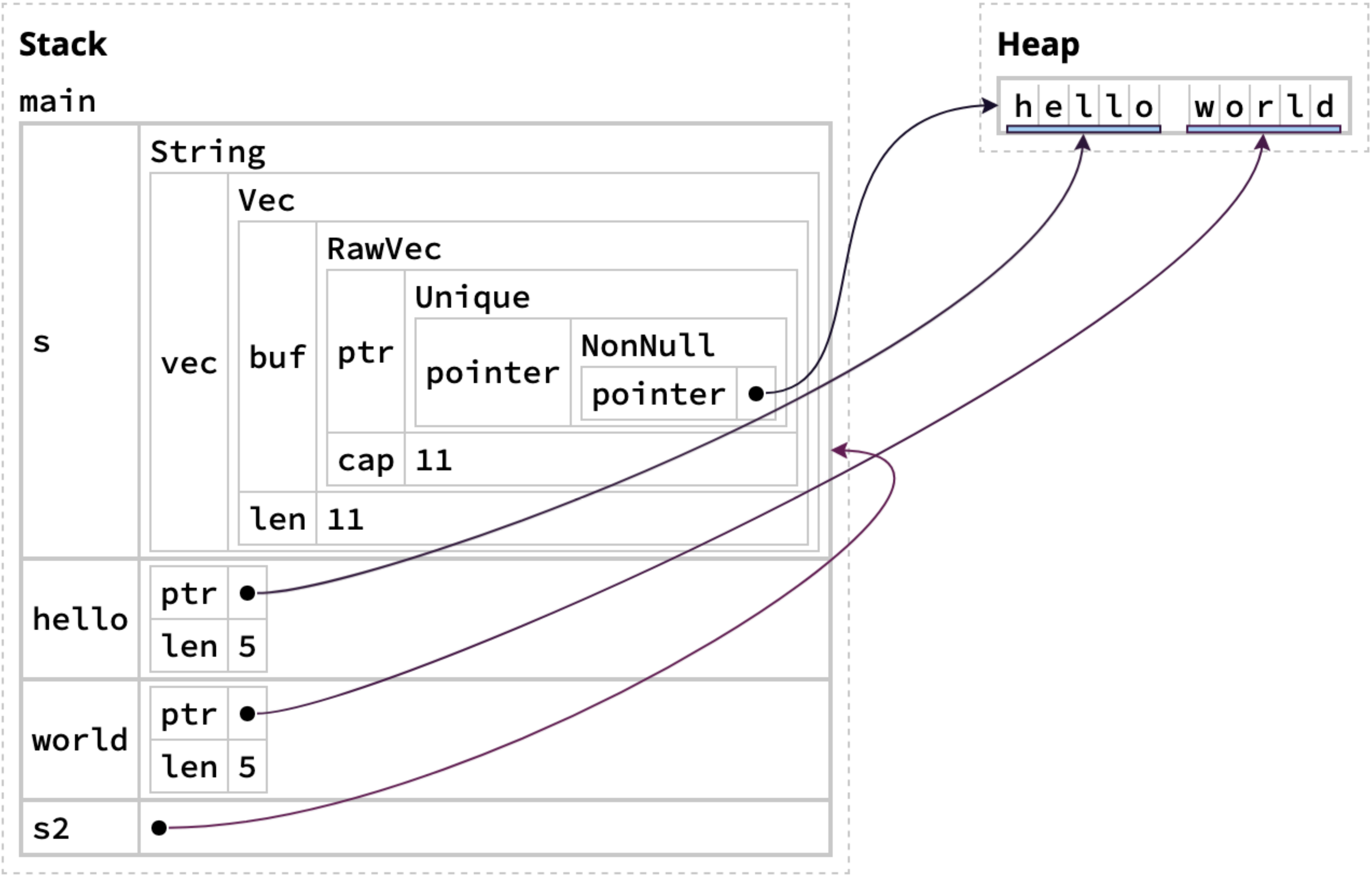
Because of the automatic conversion, many string methods actually operate on &str and not String

- .len()
- .is_empty()
- .find()
- .parse()
- .starts_with()
- .lines()
- .replace() [operates on &str, returns a String]

Slices are “fat” pointers

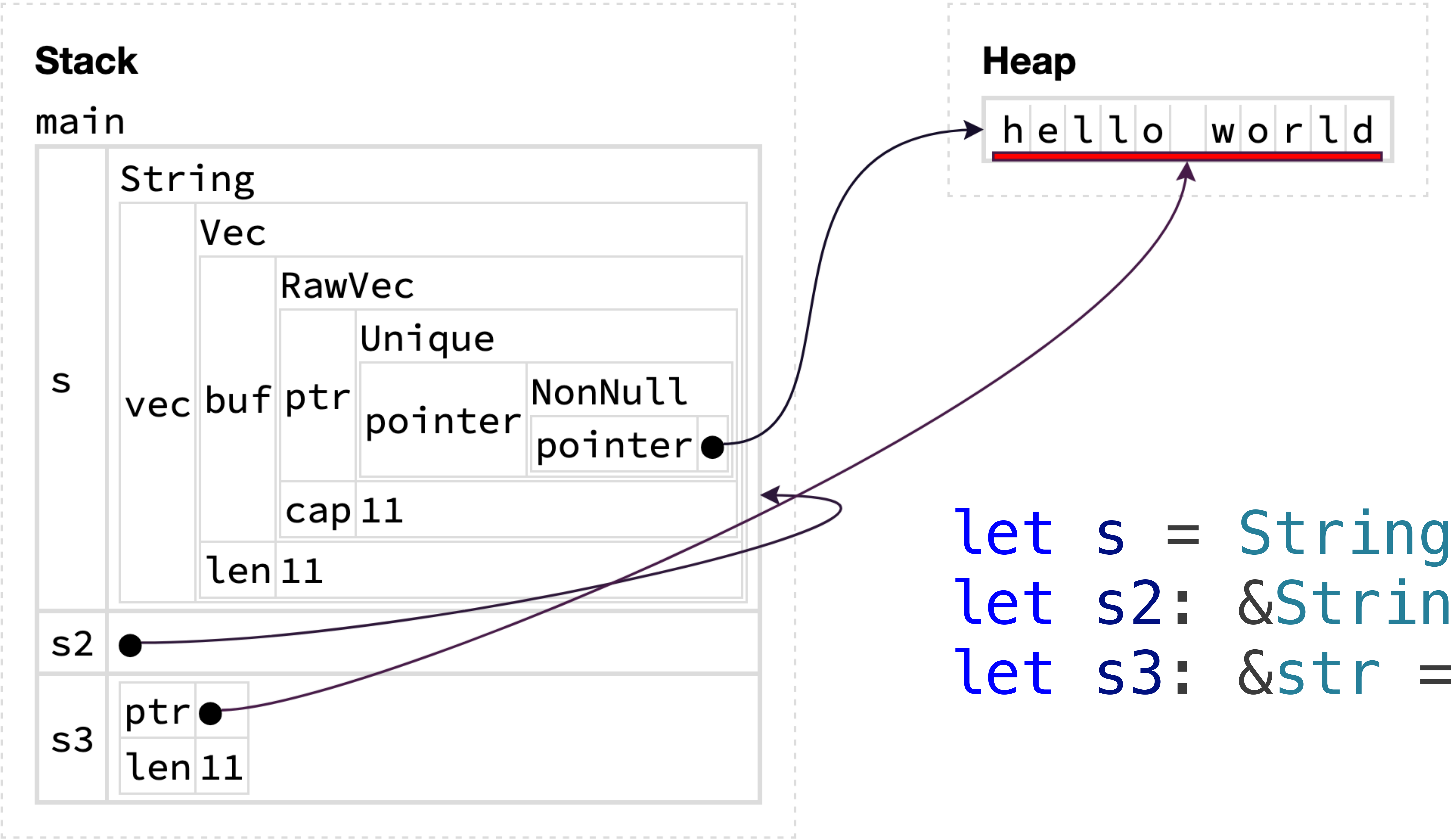
Slices are non-owning pointers with additional data, namely a length

```
let s = String::from("hello world");
let hello: &str = &s[0..5];
let world: &str = &s[6..11];
let s2: &String = &s;
```



&String vs. &str

&String is a pointer to the String
&str is a pointer + length to the actual string data




```rust
let s = String::from("hello world");
let s2: &String = &s;
let s3: &str = &s;
```

```rust
let mut sentence = String::from("This is sample sentence.");
// Get a reference to the first word.
let orig_first_word: &str = sentence.split_whitespace().next().unwrap();

sentence.make_ascii_uppercase(); // Convert to upper case letters in place (no reallocation)
// Get a reference to the new first word.
let new_first_word: &str = sentence.split_whitespace().next().unwrap();
println!("{orig_first_word} -> {new_first_word}");
```

```
error[E0502]: cannot borrow `sentence` as mutable because it is also borrowed as immutable
  |
4 | let orig_first_word: &str = sentence.split_whitespace().next().unwrap();
  |                             ---------------------------- immutable borrow occurs here
5 |
6 | sentence.make_ascii_uppercase();
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
...
9 | println!("{orig_first_word} -> {new_first_word}");
  |           ----------------- immutable borrow later used here
```

This error

A. Prevented undefined behavior

B. Prevented a logic bug

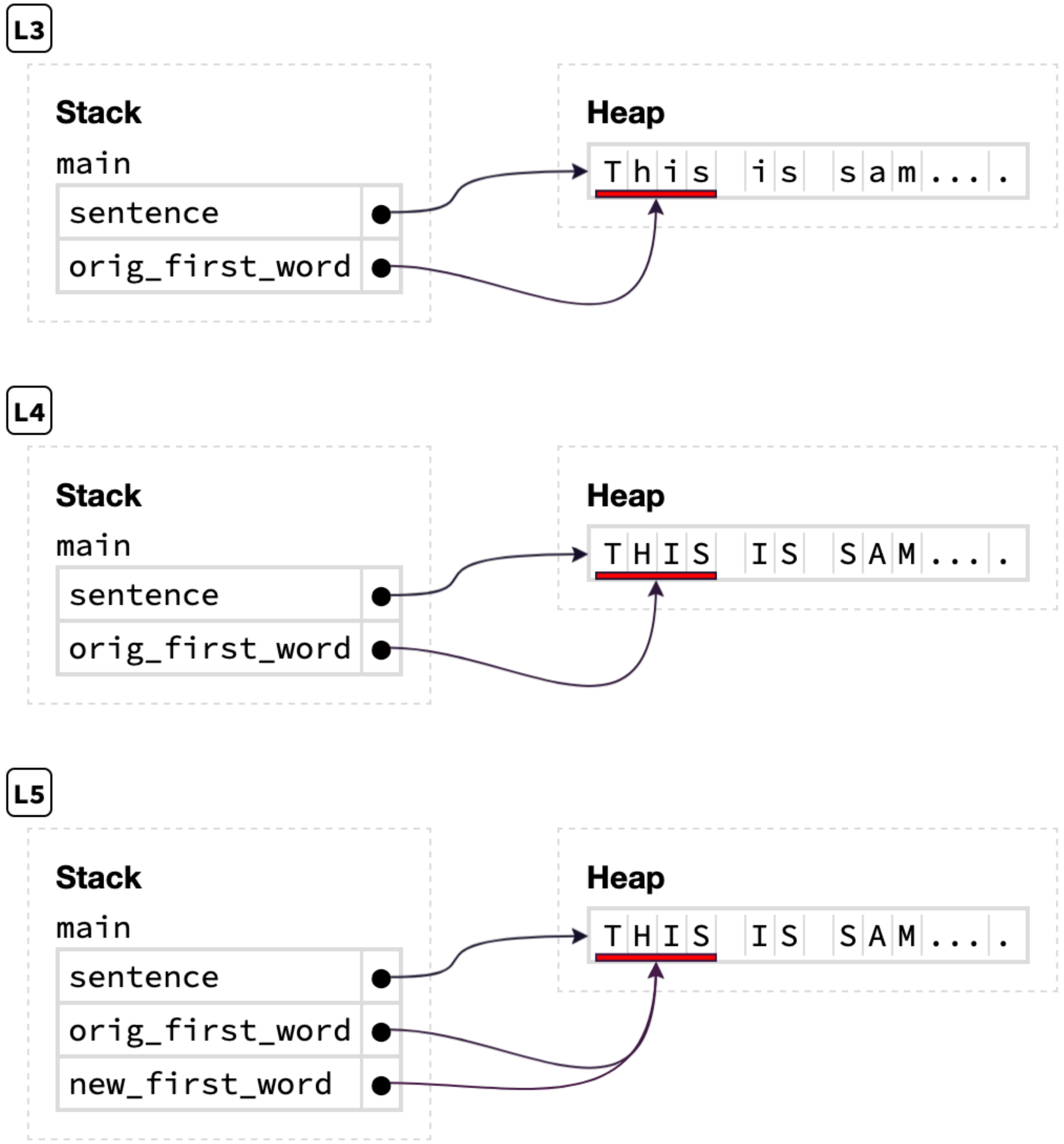
C. Is due to a limitation in Rust's analysis

Stack/heap from clicker question

L3 shows the stack/heap after creating the orig_first_word slice

L4 shows the stack/heap after uppercasing the string

L5 shows the stack/heap after creating the new_first_word slice



How the Borrow Checker caught this

```rust
let mut sentence = String::from("This is sample sentence.");

// Get a reference to the first word.
let orig_first_word: &str = sentence.split_whitespace()
    .next()
    .unwrap();


sentence.make_ascii_uppercase();
// Get a reference to the new first word.
let new_first_word: &str = sentence.split_whitespace()
    .next()
    .unwrap();
println!("{orig_first_word} -> {new_first_word}");
```

sentence	↥	+R	+W	+O

sentence	→	R	~~W~~	~~O~~
orig_first_word	↥	+R	–	+O
*orig_first_word	↥	+R	–	–

R W

Write permission is expected, but the path does not have the permission.

Fixing the code

The problem: We’re changing the string we have a reference to

The solution: Create a new string holding the original contents of the word

```
let orig_first_word = String::from(sentence.split_whitespace().next().unwrap());
```

String slices are slightly annoying

```rust
/// Return a slice referencing the first
/// two characters of s
fn first_two(s: &str) -> &str {
    &s[..2]
}

fn main() {
    let ascii = String::from("ASCII text");
    let s = first_two(&ascii);
    println!("{s}");

    let emoji = String::from("🦀🦑🦐");
    let t = first_two(&emoji);
    println!("{t}");
}
```

Output

```
AS

thread 'main' panicked at 'byte index 2 is not a char
boundary; it is inside '🦀' (bytes 0..4) of `🦀🦑🦐`',
slice.rs:11:6
```

String slices must be on UTF-8 boundaries

Strings are UTF-8 encoded

- Each Unicode “code point” is encoded in 1–4 bytes
- String slices must start and end on valid UTF-8 boundaries
- Some characters (e.g., some emoji) require multiple code points like 🏴‍☠️ which requires 4 code points and 13 bytes!
- Some characters (mostly those with accents) have (at least) two different encodings: a “precomposed” version like ÿ (1 code point, 2 bytes) and a decomposed version consisting of y and ¨ (2 code points, 3 bytes)

Text is **hard**

&[T; n] -> &[T]
&Vec<T> -> &[T]

Rust will convert a reference to an array [T; n] or a reference to a Vec<T> into an array slice &[T]

```
let arr: [bool; 4] = [true, false, false, true];
let v: Vec<u8> = vec![128, 64, 32, 16, 8, 4, 2, 1];

let slice1: &[bool] = &arr;
let slice2: &[u8] = &v;
```

Array slices

```rust
fn sum(data: &[i32]) -> i32 {
    let mut result = 0;
    for x in data {
        result += *x;
    }
    result
}

fn main() {
    let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    let v = vec![3, -72, 42, 100];

    println!("{}", sum(&arr[1..3]));
    println!("{}", sum(&arr));
    println!("{}", sum(&v[2..]));
    println!("{}", sum(&v));
}
```

Many methods are defined on slices rather than the array or Vec

Examples

- .len()
- .first()
- .last()
- .get()   Returns a reference to the item or slice wrapped in an Option
- .get_mut() Same but returns a mutable reference
- .contains()
- .starts_with()
- .binary_search()
- .sort()

Ranges

We create a slice by giving a range [start, end) as start..end

- &foo[..end] is the same as &foo[0..end]
- &foo[start..] is the same as &foo[start..foo.len()]

Ranges are more generally useful

```
    for x in 0..4 {
        println!("{x}");
    }
Output:
0
1
2
3
```

Inclusive ranges

The syntax start..=end gives a range [start, end] (so it includes end)

```
    for x in 0..=4 {
        println!("{x}");
    }
Output:
0
1
2
3
4
```

Range start and end

The start and end of a range can be variables or expressions

```
let x = 10;
let y = 20;
for num in x+3..2*y {
    println!("{num}");
}
```

Prints out 13, 14, …, 39

Reversing a range

Ranges are a type of reversible iterator so we can use .rev() to get an iterator in the reverse order

```
let x = 10;
let y = 20;
for num in (x+3..2*y).rev() {
    println!("{num}");
}
```

Prints 39, 38, …, 13

How do you construct an iterator that returns the values 20, 19, …, 11?

A. (11..20).rev()

B. (10..20).rev()

C. (10..21).rev()

D. (11..21).rev()

E. (9..19).rev()