CS 241: Systems Programming
Lecture 8. Introduction to Rust

Fall 2025

Prof. Stephen Checkoway

Why Rust?

Systems programming usually taught in C, but C is (nearly) impossible to write correctly!

Bad, unsafe C code leads to very bad vulnerabilities

Rust focuses on safety

Rust isn’t just for academics! US government issued report urging use of memory-safe languages

Rust is starting to appear in operating system kernels and is replacing critical software like sudo and Python’s cryptography module

Hello, World!

```rust
fn main() {
    println!("Hello world!");
}
```

Every program needs a main function

println!() prints a string and a newline to stdout

All of the executable code lives in a function (unlike Python)

Compiling and running

Use rustc to compile (will perform both compiling and linking by default)

-
  ```
  $ rustc helloworld.rs
  ```

rustc produces the executable helloworld

To run a program from the current directory, use ./ as usual:

-
  ```
  $ ./helloworld
  Hello world!
  ```

Jobs of a Compiler

Inputs

- Rust program files and options
- Libraries

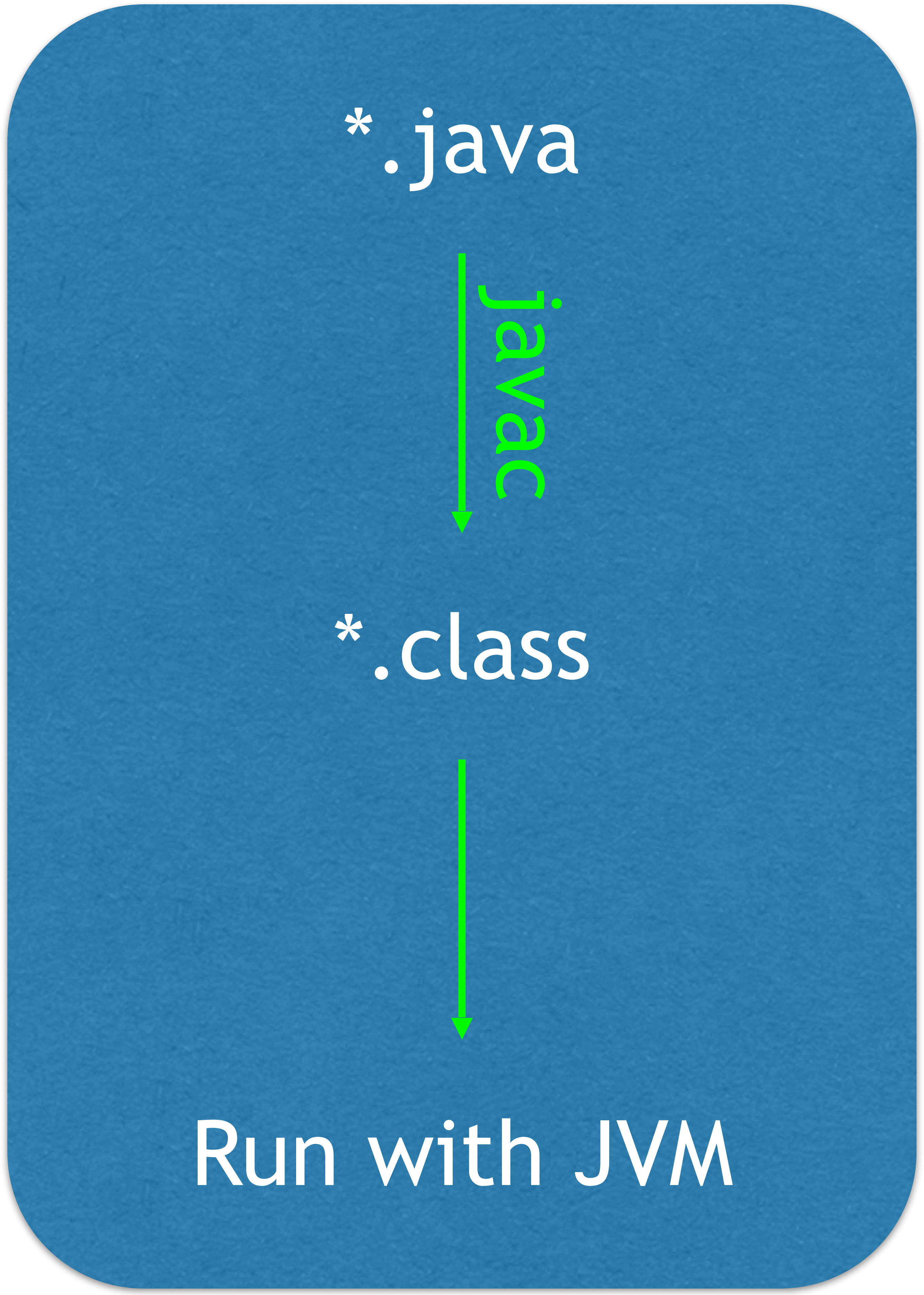
Compilation phases

- Compilation — Turns source files into object files
- Linking — Combines object files into executables

Outputs

- Executable
- Warnings and errors

Compilation



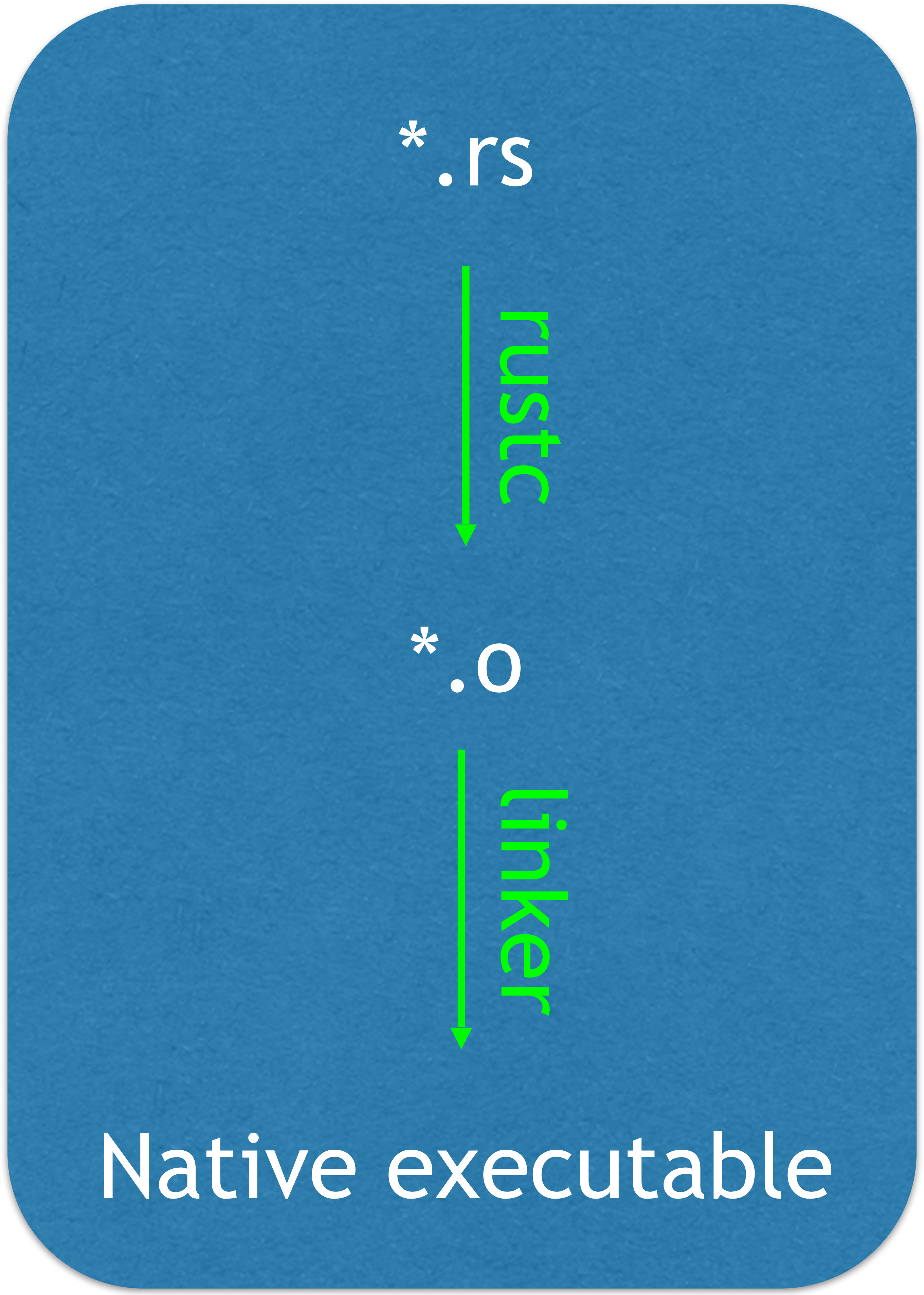
*.java

javac

*.class

Run with JVM

Java Model



*.rs

rustc

*.o

linker

Native executable

Rust Model

Basic types

Integer types

- Signed integer types (can be negative): `i8`, `i16`, **`i32`**, `i64`, `i128`
  - Equivalent to Java's `byte`, `short`, `int`, and `long`
  - `i32` is the default when not specified
- Unsigned integer types (only nonnegative): `u8`, `u16`, `u32`, `u64`, `u128`

Floating point types

- `f32` and `f64`
- Equivalent to Java's `float` and `double`

String types

- `String` and `&str`

More basic types

Boolean type: `bool`

- Values are `true` and `false`

Character type: `char`

- 4-bytes in size, holds one Unicode code point which represents one simple character like B or 한 or 😍 but not complex characters like 🇺🇸

Platform-dependent integer types

- `usize`: 32-bit or 64-bit unsigned integer
  - Used as an index or as a count of items in a collection
- `isize`: signed version of `usize`

Unit type: ()

The unit type `()` has one value: `()`

```
let unit: () = ();
```

There isn’t much you can do with it, but we’ll actually be seeing it quite a bit

Introduce variables with let

```
let variable_name: type = value;
```

```
fn compute_area() {
    let width: u64 = 100;
    let height: u64 = 24;
    let area = width * height;

    println!("{width} x {height} = {area}");
}
```

Function arguments/return value

```
fn function_name(arg1: type1, arg2: type2) -> return_type {}
```

```
fn compute_area(width: u64, height: u64) -> u64 {
    let area = width * height;
    return area;
}

fn main() {
    let area = compute_area(20, 40);
    println!("The area is {area}");
}
```

You’re designing a function, neg(), that takes an argument of type i32 and returns an i32 with the opposite sign (i.e., positive values become negative and negative values become positive). Which of the options is the correct way to specify this?

A.
```
i32 neg(i32 val) {
    return -val;
}
```

B.
```
fn i32 neg(val: i32) {
    return -val;
}
```

C.
```
fn neg(val: i32) -> i32 {
    return -val;
}
```

D.
```
fn neg(i32 val) -> i32 {
    return -val;
}
```

Returning a String

```rust
fn rectangle_description(width: u64, height: u64) -> String {
    let desc: String;

    if width == height {
        desc = format!("{width} x {width} square");
    } else {
        desc = format!("{width} x {height} rectangle");
    }
    return desc;
}
```

Blocks have values

```
let val = {
    let x = 10;
    let y = 20;
    x + y
};
```

The value of a block of code in braces is the value of the last expression in the block

Notice the lack of `;` at the end of the block and the `;` after the block

The value of an `if` expression is the value of the last expression of its branches

Variables' scope ends at the end of their containing block

The **scope** of a variable is the region of code where the variable is accessible

```
fn main() {
    let val = {
        let x = 10;
        let y = 20;
        x + y
    };
    println!("{val}"); // OK
    println!("{x} {y}"); // Not OK
}
```

```
error[E0425]: cannot find value `x` in this scope
   --> foo.rs:8:16
    |
143 |     println!("{x} {y}"); // Not OK
    |                 ^ not found in this scope
```

`if` is an expression, it has a value

```rust
fn rectangle_description(width: u64, height: u64) -> String {
    let desc = if width == height {
        format!("{width} x {width} square")
    } else {
        format!("{width} x {height} rectangle")
    };
    return desc;
}
```

The value of an `if` expression is the value of the last expression of its taken branch

Notice the lack of `;` at the end of both blocks of the `if` and the `;` after the `if`

Last expression in a function is returned

```rust
fn rectangle_description(width: u64, height: u64) -> String {
    let desc = if width == height {
        format!("{width} x {width} square")
    } else {
        format!("{width} x {height} rectangle")
    };
    desc
}
```

The `return` is gone as is the semicolon

Idiomatic Rust

```
fn rectangle_description(width: u64, height: u64) -> String {
    if width == height {
        format!("{width} x {width} square")
    } else {
        format!("{width} x {height} rectangle")
    }
}
```

The value returned from the function is the value of the last expression: the `if`

The value of the `if` is the value of the last expression of the taken branch of the `if`

What is the “Rusty” way to write the neg() function? Meaning, which of these is the best practice?

```
A. fn neg(val: i32) -> i32 {
       -val
   }
```

```
B. fn neg(val: i32) -> i32 {
       -val;
   }
```

```
C. fn neg(val: i32) -> i32 {
       return -val
   }
```

```
D. fn neg(val: i32) -> i32 {
       return -val;
   }
```

Mutability

Variables are immutable by default (they cannot be changed)

Let’s experiment with the Rust Playground
https://play.rust-lang.org

Cannot assign twice to immutable variable

Error indicates we tried to modify an immutable variable

Error message indicates a solution

- help: consider making this binding mutable: `mut x`

```rust
let mut x = 10;
println!("{x}");
x = 20;
println!("{x}");
```

Group discussion: Why do you think variables are immutable by default in Rust when most languages make them mutable by default?

A. Select this answer

Strings

A `String` holds an *owned* collection of characters

- Owned means the collection of characters belongs to the String value

A `&str` is an *immutable reference* to a string

- References are a way to share values

Text in double quotes is a &str, a reference to an immutable string

We can create a `String` from a `&str` using `String::from()`

```
let s1: &str = "Điếc không sợ súng.";
let s2: String = String::from("Ignorance is bliss.");
```

Omitting the type

```
let s1 = "Điếc không sợ súng.";
let s2 = String::from("Ignorance is bliss.");
```

The type of a variable is often omitted when it is clear from context

- Strings in double quotes are always `&str` so the type is omitted
- When the type name appears on the right-hand side of the `=`, the type is omitted

Converting between &str and String

`String::from(s)` creates a `String` from a `&str` by making a copy of the string

`"foo".to_string()` also creates a `String` from a `&str` by making a copy of the string

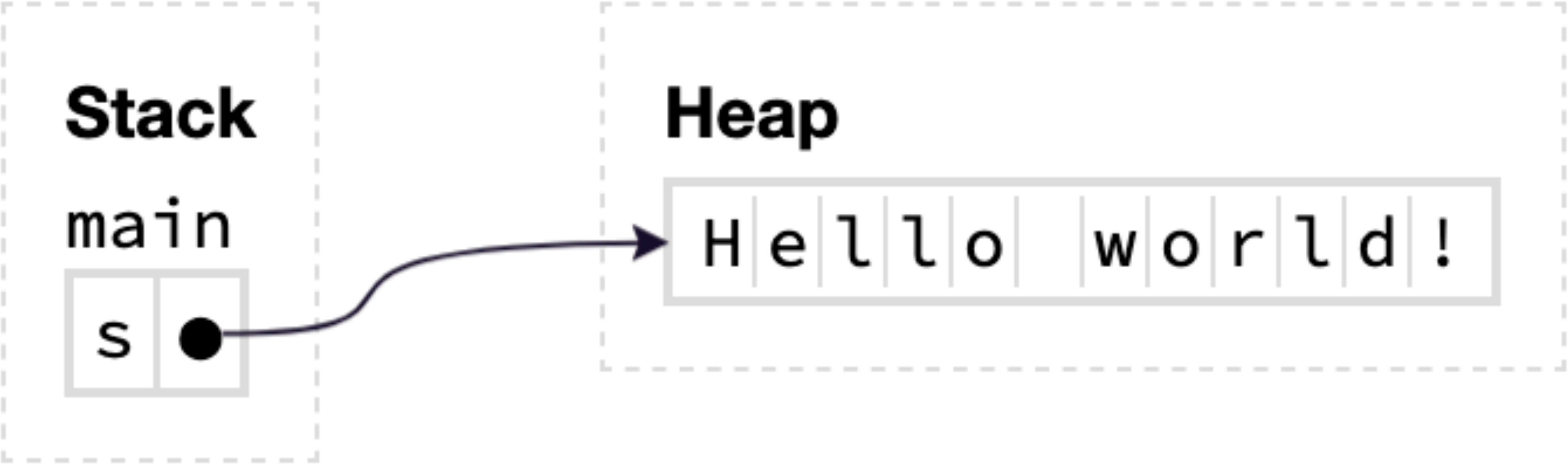
A `String`'s `as_str()` method returns a `&str` reference to itself, no copy is made

```
let s1 = String::from("blah");
let s2 = s1.as_str();
```

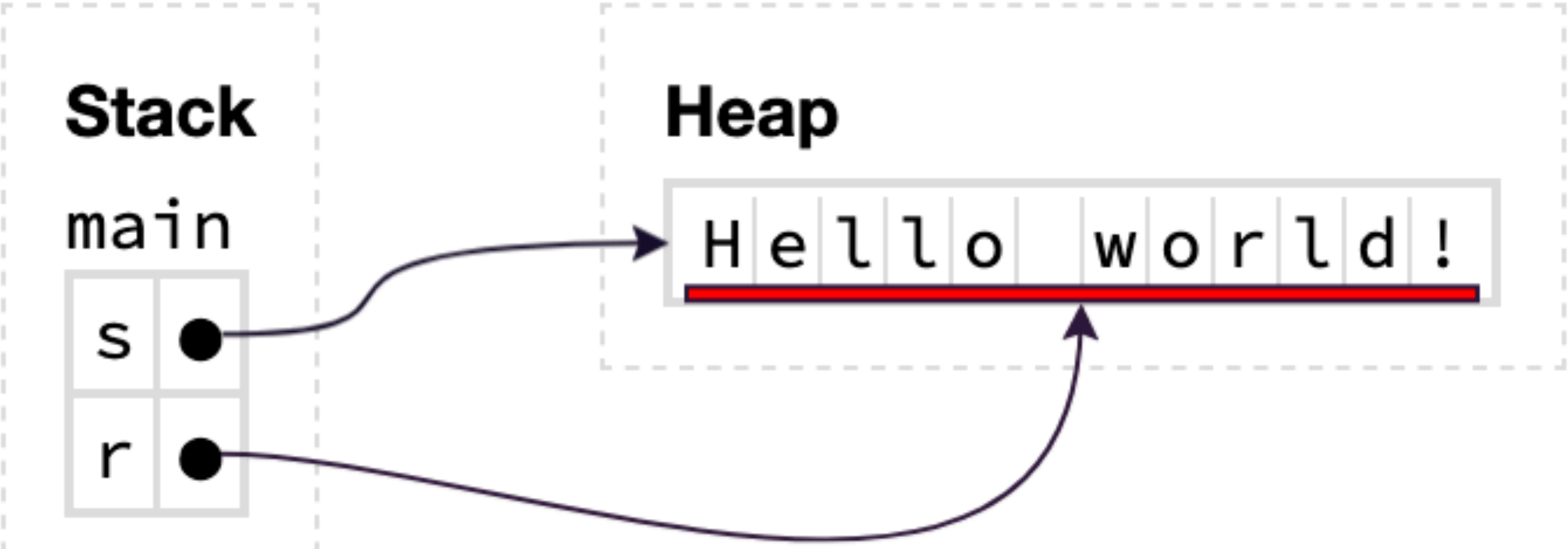
Example

```
fn main() {
  let s = String::from("Hello world!"); // L2
  let r = s.as_str(); // L3
  let s2 = s.to_string(); // L4
}
```

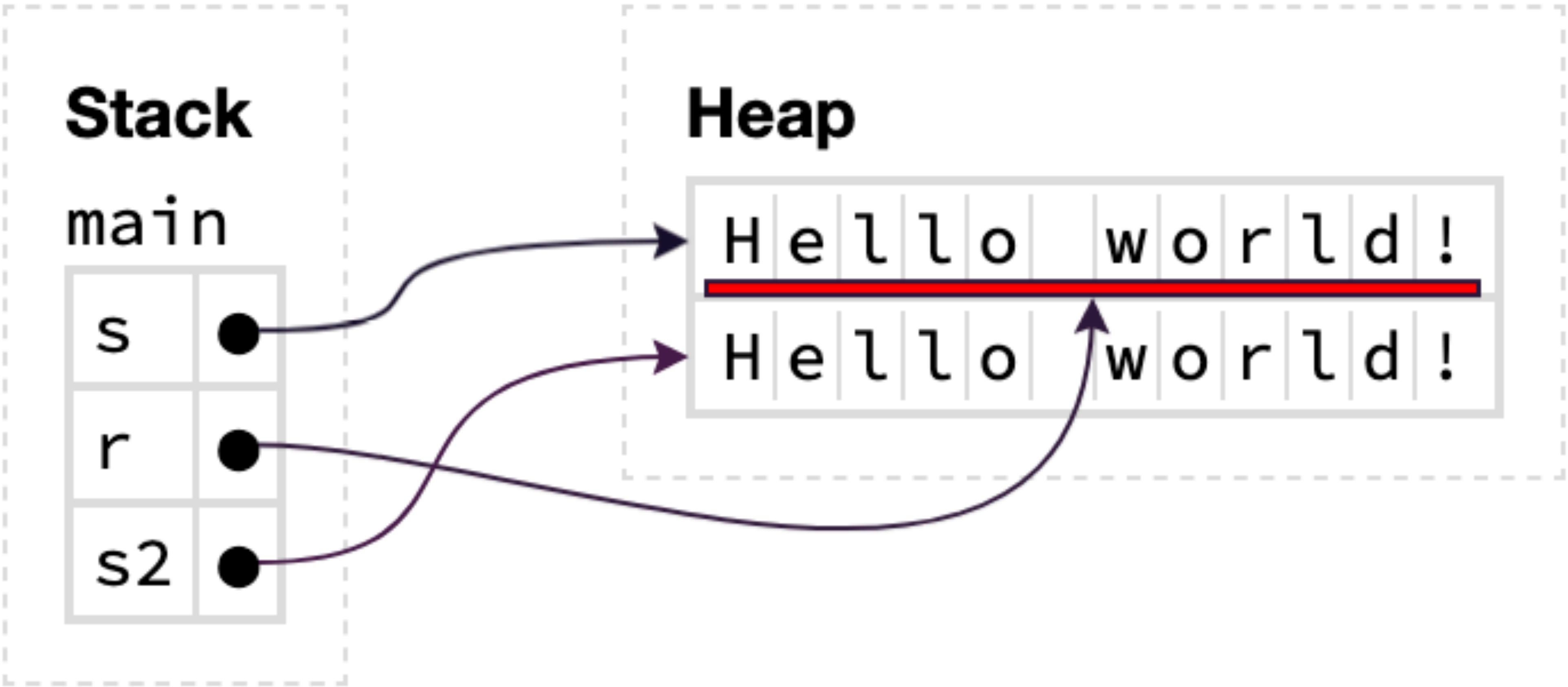
L2



L3



L4



Passing strings to functions

```rust
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);    // Valid, passes a reference to t to bar
    bar("xyz"); // Valid
}
```

Returning &str is hard

There are two problems with this function:

```rust
fn foo(num: i32) -> &str {
    let s = format!("num = {num}");
    return &s;
}
```

1. Rustc gives an error, “expected named lifetime parameter” (we’ll talk about lifetimes later)

2. More importantly, s goes away when the function ends so the reference to it would be invalid; Rust prevents this.

Variables’ scope ends at the end of their containing block

The **scope** of a variable is the region of code where the variable is accessible

```
fn main() {
    let val = {
        let x = 10;
        let y = 20;
        x + y
    };
    println!("{val}"); // OK
    println!("{x} {y}"); // Not OK
}
```

```
error[E0425]: cannot find value `x` in this scope
   --> foo.rs:8:16
    |
143 |     println!("{x} {y}"); // Not OK
    |                ^ not found in this scope
```

Aside, C doesn’t prevent this!

```c
#include <stdio.h>

char *foo(int num) {
    char s[100];
    snprintf(s, sizeof(s), "num = %d", num);
    return s;
}

int main() {
    char *s = foo(123);
    puts(s);
    return 0;
}
```

What happens when we run this?

```
$ ./example
`\M

$ ./example
`?

$ ./example
`??
```

General rule of strings

When passing a string to a function, use a `&str` reference

When returning a string from a function, return a `String`

These rules don't always hold, later we'll see how to return a `&str` in some cases