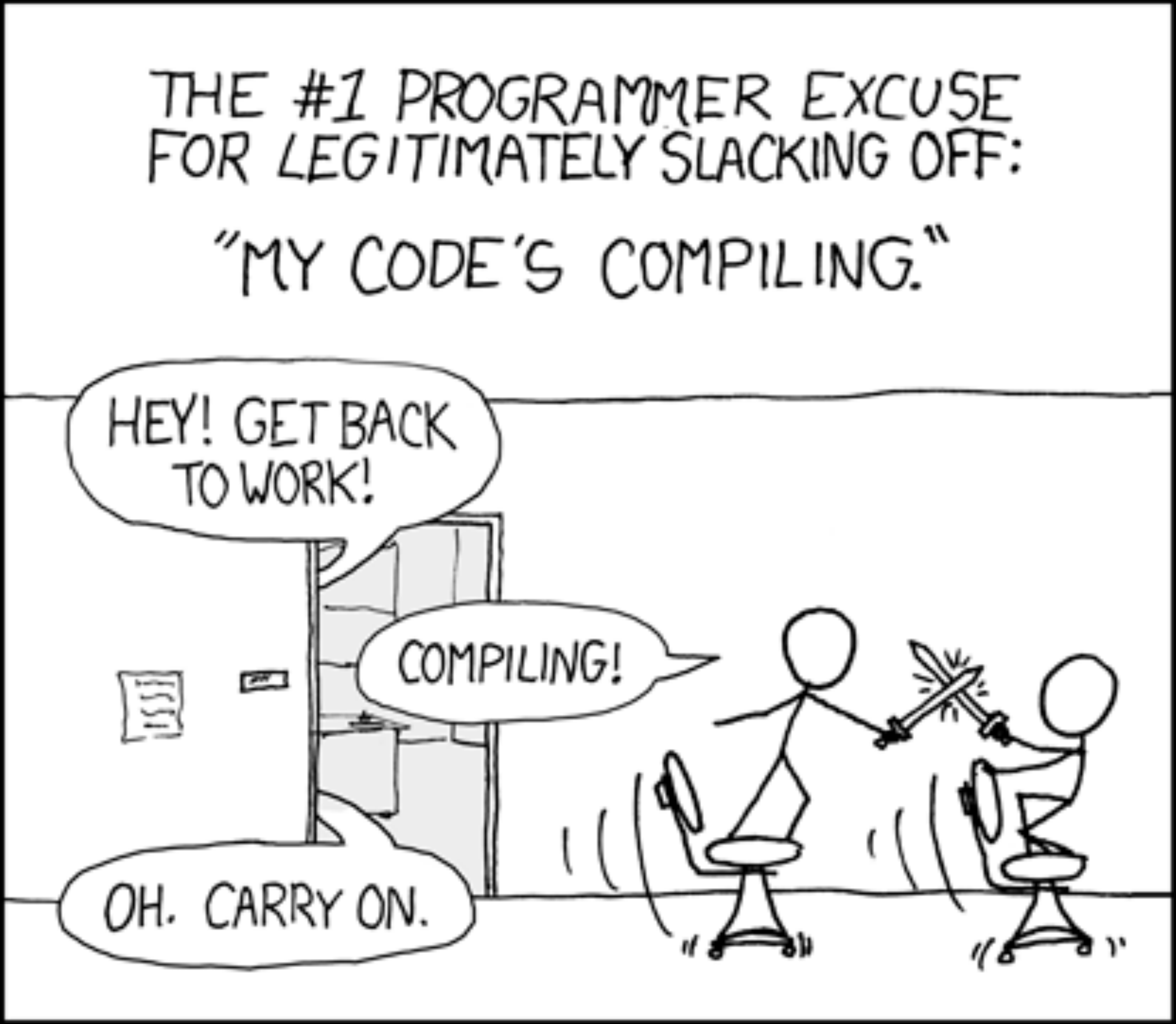


https://xkcd.com/303/

CS 152: *Programming Language Paradigms*

Rust

Prof. Tom Austin
San José State University

What is wrong with C/C++?

- Painfully slow build times
- Not memory safe
- No good concurrency story

"When the three of us [Ken Thompson, Rob Pike, and Robert Griesemer] got started, it was pure research. The three of us got together and decided that we hated C++."

--Ken Thompson on the motivation for Go

"C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it blows your whole leg off."

--Bjarne Stroustrup

C++ is a horrible language.

--Linus Torvalds

Tony Hoare's billion dollar mistake

"But I couldn't resist the temptation to put in a `null` reference, simply because it was so easy to implement. This has led to **innumerable errors, vulnerabilities, and system crashes**, which have probably caused a billion dollars of pain and damage in the last forty years."

Challenges with C

A buggy C function

```c
int* zero_negs(int a[], int len){
  int res[len];
  for (int i=0; i<len; i++) {
    if (a[i] < 0) res[i] = 0;
    else res[i] = a[i];
  }
  return res;
}
```

Fixed?

```
int* zero_negs(int a[], int len){
  int *res=malloc(sizeof(int)*len);
  for (int i=0; i<len; i++) {
    if (a[i] < 0) res[i] = 0;
    else res[i] = a[i];
  }
  return res;
}
```

A consumer of data, which frees the data.

```
void print_arr(int a[], int len){
  for (int i=0; i<len; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
  free(a);
}
```

But what if the consumer is called twice?

```
int main(int argc, char** argv) {
  int nums[] = {0,12,5,-42,9,7,-18,0};
  int n = 8;
  int *no_negs = zero_out_negs(nums,n);
  print_arr(no_negs, n);
  // ... Sometime later in the code.
  // Freeing memory twice.
  print_arr(no_negs, n);
}
```

Memory Management

- C/C++ force the programmer to manage memory, which can cause:
  - Memory leaks
  - Dangling pointers
- Java uses a *garbage collector*
  - Stop-the-world gc.
  - Applications stops while gc runs.

Rust history

- Developed by Graydon Hoare of Mozilla
- Used in
  - Project Servo: layout engine for Firefox
  - The Rust compiler
- Emphasis:
  - Safety
  - Control of memory layout
  - Concurrency

hello_world.rs

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

Denotes that `println` is a macro

```
$ rustc hello_world.rs
$ ./hello_world
Hello, world!
```

Primitive Types

- signed integers: `i8`, `i16`, `i32`, `i64`
- unsigned integers: `u8`, `u16`, `u32`, `u64`
- pointer sizes: `isize` (signed),
  `usize` (unsigned)
- floating point: `f32`, `f64`
- `char`, `bool`
- arrays `[1,2,3]` and tuples `(1,true)`
- the unit type `()`

Functions in Rust

```
fn foo(x: i32) -> i32 {
  x + 3
}

fn main() {
  println!("{}", foo(4));
}
```

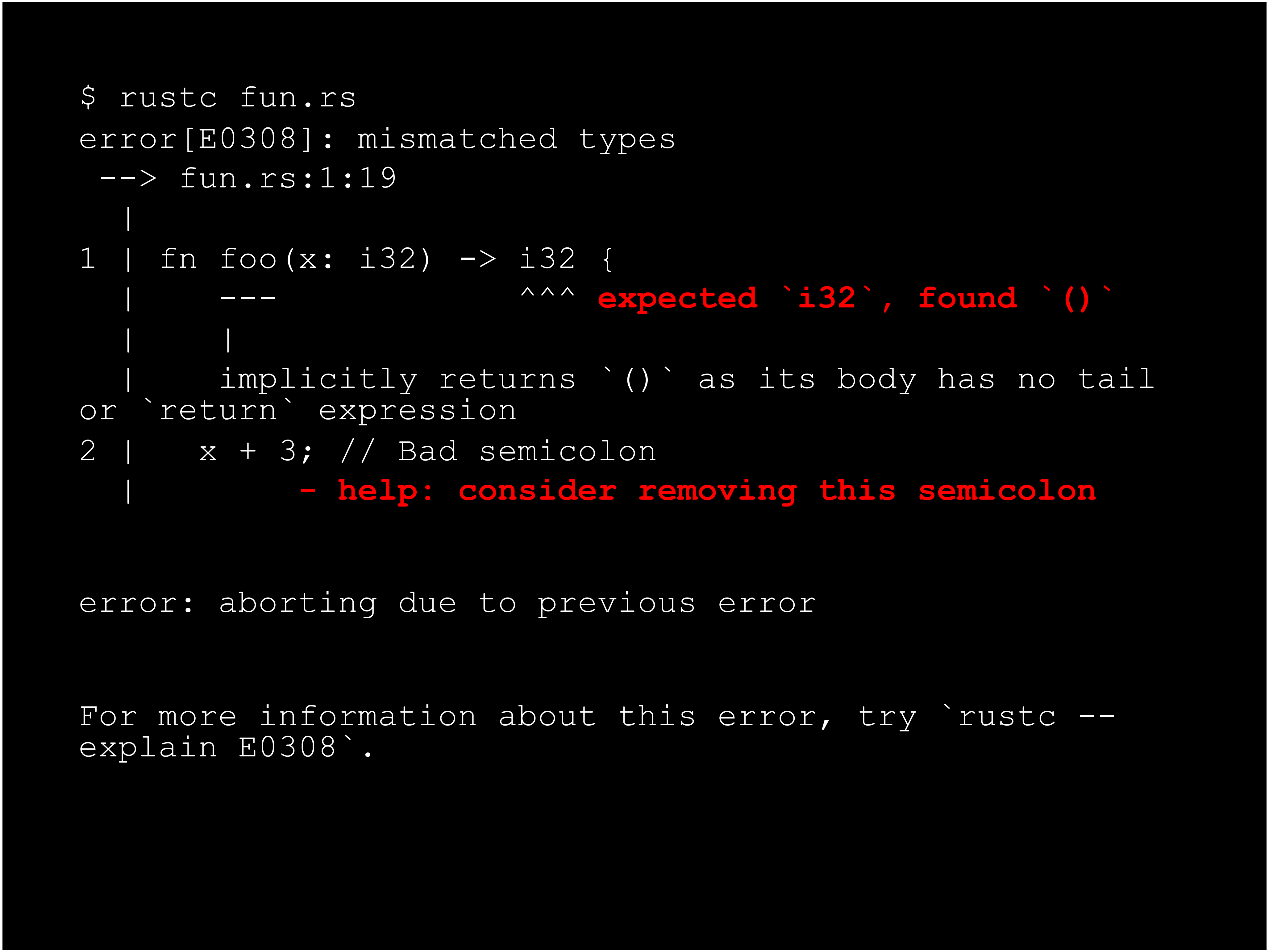
Compiling and Running Rust Program

```
$ rustc fun.rs
$ ./fun
7
$
```

Broken Rust Program

```
fn foo(x: i32) -> i32 {
  x + 3 ; // Semicolon error
}

fn main() {
  println!("{}", foo(4));
}
```



```
$ rustc fun.rs
error[E0308]: mismatched types
 --> fun.rs:1:19
  |
1 | fn foo(x: i32) -> i32 {
  |    ---            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail
or `return` expression
2 |   x + 3; // Bad semicolon
  |        - help: consider removing this semicolon


error: aborting due to previous error


For more information about this error, try `rustc --
explain E0308`.
```

Variables in Rust

```
fn main() {
  // Type annotations are not needed
  let a = 1;
  let b = 2;

  // But you can specify them if you want
  let c: isize = 3;

  // '{}' is a placeholder for arguments
  println!("a:{} b:{} c:{}", a, b, c);
}
```

More sophisticated printing

```
fn main() {
  // Numbers can specify argument
  println!("<{0}>{1}</{0}>", "h1", "Hi!");

  // Named arguments can also be useful
  println!("<{tag}>{body}</{tag}>",
    tag="strong",
    body="Welcome to Rust");
}
```

Structs

- Rust can create more sophisticated data structures through *structs*
- We will illustrate with a complex number example

```
struct Complex { real: i32, imaginary: i32 }

fn add_complex(c1: Complex, c2: Complex) -> Complex {
  let r = c1.real + c2.real;
  let i = c1.imaginary + c2.imaginary;
  Complex { real: r, imaginary: i }
}

fn main() {
  let cmplx1 = Complex { real: 7, imaginary: 2 };
  let cmplx2 = Complex { real: 3, imaginary: 1 };
  let ans = add_complex(cmplx1, cmplx2);

  println!("The answer is {}+{}i",
    ans.real,
    ans.imaginary);
}
```

Lab, part 1: Modify Complex.rs

Currently, the code prints:

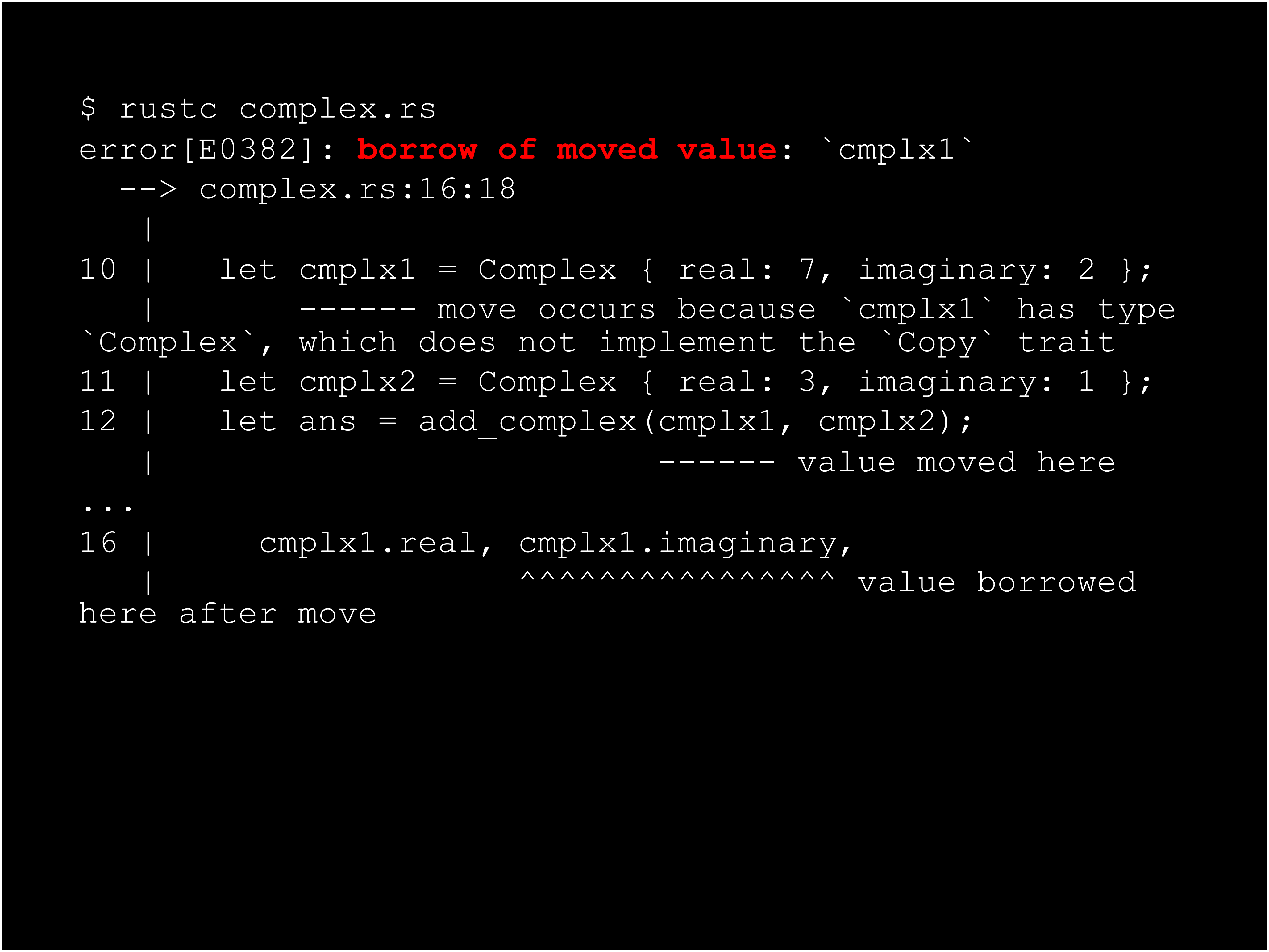
```
The answer is 10+3i
```

Modify the println to refer to `cmplx1` and `cmplx2`. It should print:

```
7+2i + 3+1i = 10+3i
```

Possible attempt:

```
println!("{}+{}i + {}+{}i = {}+{}i",
    cmplx1.real, cmplx1.imaginary,
    cmplx2.real, cmplx2.imaginary,
    ans.real, ans.imaginary);
```



```
$ rustc complex.rs
error[E0382]: borrow of moved value: `cmplx1`
  --> complex.rs:16:18
   |
10 |   let cmplx1 = Complex { real: 7, imaginary: 2 };
   |       ------ move occurs because `cmplx1` has type
`Complex`, which does not implement the `Copy` trait
11 |   let cmplx2 = Complex { real: 3, imaginary: 1 };
12 |   let ans = add_complex(cmplx1, cmplx2);
   |                         ------ value moved here
...
16 |     cmplx1.real, cmplx1.imaginary,
   |                  ^^^^^^^^^^^^^^^^ value borrowed
here after move
```

Memory management approaches revisited

- C/C++
  - manually managed
  - let the programmer beware
- Java
  - Virtual machine with garbage collector
  - Run-time enforcement of key properties
  - Performance overhead

Rust memory management

- No run-time or garbage collection
- Compiler statically enforces memory safety
- Uses RAII strategy
  - **Resource Acquisition Is Initialization**
  - resource allocation done at initialization
  - resource deallocation done when the object goes out of scope

Ownership Transfer Example

```
fn f(x: Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(a);
}
```

Error

```
fn f(x: Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(a);
    println!("{}", a);
}
```

Fixed: `f` Modified to Borrow

```
fn f(x: &Box<i32>) {
    println!("{}", x);
}
fn main() {
    let a = Box::new(42_i32);
    println!("{}", a);
    f(&a);
    println!("{}", a);
}
```

Lab, part1: continued

Work on lab part1 to fix complex.rs.

Mutability in Rust

- Like Racket, Rust discourages mutable data.
- If you want to make a value mutable, you must use the `mut` modifier.

Array example

```
fn main() {
  let mut a: [i32; 10] = [0;10];
  let mut i = 0;
  while i <= 10 {
    println!("Accessing {}", i);
    a[i] = i as i32;
    i = i + 1;
  }
}
```

Function with mutable borrow

```
fn square_cplx(c: &mut Complex) {
  let r = c.real * c.real -
      c.imaginary * c.imaginary;
  let i = c.real * c.imaginary +
      c.imaginary * c.real;
  c.real = r;
  c.imaginary = i;
}
```

Calling function with mutable borrow

```
// ans is mutable
let mut ans = ...;

// Loans ans to function
square_cplx(&mut ans);
```

Typechecking in Rust

- Is Rust statically or dynamically typed?
- Sample code:

```
fn main() {
    let x = 42;
    println!("{}", x);
}
```

This Code Won't Compile

```
fn main() {
    let s = "hello";
    let x = s + 42;
    println!("{}", x);
}
```

Compilation Error

```
$ rustc typing.rs
error[E0369]: binary operation `+`
cannot be applied to type `&str`
 --> typing.rs:3:15
  |
3 |     let x = s + 42;
  |             - ^ -- {integer}
  |             |
  |             &str
  |
```

Type Inference

- Rust *can* have type annotations:

```
let x: i32 = 99;
```

- For local variables, types are optional.
- For functions, types are mandatory.

Function Type Annotations

```
fn double(n: i32) -> i32 {
  // No semicolon on the next line.
  // (Explicit return w/ ';' also works).
  n*2
}

// No arguments, no return value.
fn main() {
  let x = 45;
  let y = double(3);
  println!("{}", x+y);
}
```

Rust documentation

Rust programming language "book"

https://doc.rust-lang.org/nightly/book/

Rust by Example

http://rustbyexample.com/

Lab, part 2: Implement Quicksort

- Use sort0.rs, sort1.rs, and sort2.rs for reference (available online)

Optional due to COVID-19