CS3211 Tutorial 7.5 & 8

Safety and Concurrency in Rust
Simon – T5

Adapted From Sriram's Slides

Why talk about a third language?

C++ (50% of the mod)

- Threads
- Synchronization
- Atomics
- Memory Ordering
- Debugging
- Lock-Free Programming

Powerful, but needs us to be really careful. Significant mental load?

The promise of performance, but programmer is responsible for safety

Go (25% of the mod)

- Lightweight Co-routines (Goroutines)
- Channels and Message Passing

Useful mental model, but garbage collection, copying in channels, etc, may impact performance.

Rust (25% of the mod)

- Compile-time safety checking
- Safe futures / async
- Safe data parallelism
- Etc

Can we have a fast, modern language that reduces the programmer's mental correctness burden somewhat?

What is the future of prog lang?

Chromium will support third-party Rust libraries

Google has announced that it will allow t
browser project.

Chrome security team member Dana Ja
decision.

Jansens says that Google is now actively

m is to provide a simpler (no IPC) and safer (less complex C+
ndbox...

LINUX / OPEN SOURCE / RUST / SOFTWARE DEVELOPMENT

Rust in the Linux Kernel

Why it's all happening for the Rust programming language, how it made it into the Linux kernel, and where it will go from here.

Oct 5th, 2022 7:00am by Steven J. Vaughan-Nichols

ENGINEERING & DEVELOPERS

WHY DISCORD IS SWITCHING FROM GO TO RUST

The Guarantees of Rust

Safe Rust ⇒ No Undefined Behavior, Yes to Memory Safety

- No **use-after-free**, **double-free**, yes to bounds checks, panics
- References are always valid and variables are initialized before use
- Data races are completely eliminated
- Etc etc

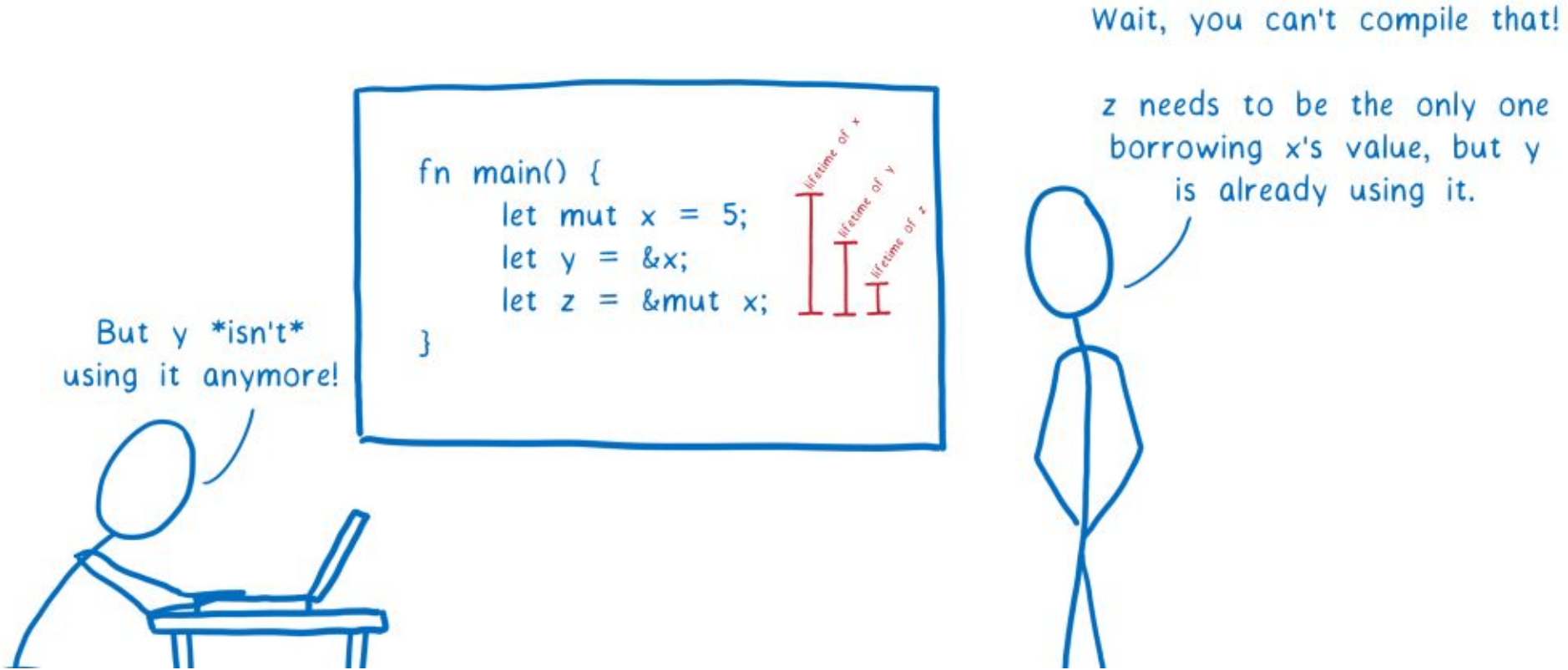
(*not all can be done at compile time, also safe code calls unsafe code..)

But, in Real Rust, all of these are still possible!

- Deadlock, livelock, etc
- Memory leaks
- Integer overflow...

Further reading:
https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html
https://tiemoko.com/blog/blue-team-rust/

One more note: Rust is not simple!



Let’s learn (what we can) about Rust

The Challenge: Performance + Guaranteed Safety

- What happens in this C++ program? [p]

```
1   #include <iostream>
2   #include <string>
3
4
5   void f(std::string* s) {
6       free(s);
7   }
8
9
10  int main(int argc, char* argv[]) {
11      std::string* s = new std::string("asdf");
12      f(s);
13      std::cout << *s;
14  }
```

https://godbolt.org/z/GW5hcG3zn

The Challenge: Performance + Guaranteed Safety

- What happens in this C++ program? [p]
  - Undefined behavior: use-after-free (note: you could pick this up with ASan)

```cpp
#include <iostream>
#include <string>


void f(std::string* s) {
    free(s);
}


int main(int argc, char* argv[]) {
  std::string* s = new std::string("asdf");
  f(s);
  std::cout << *s;
}
```

x86-64 gcc 12.2 -fsanitize=address

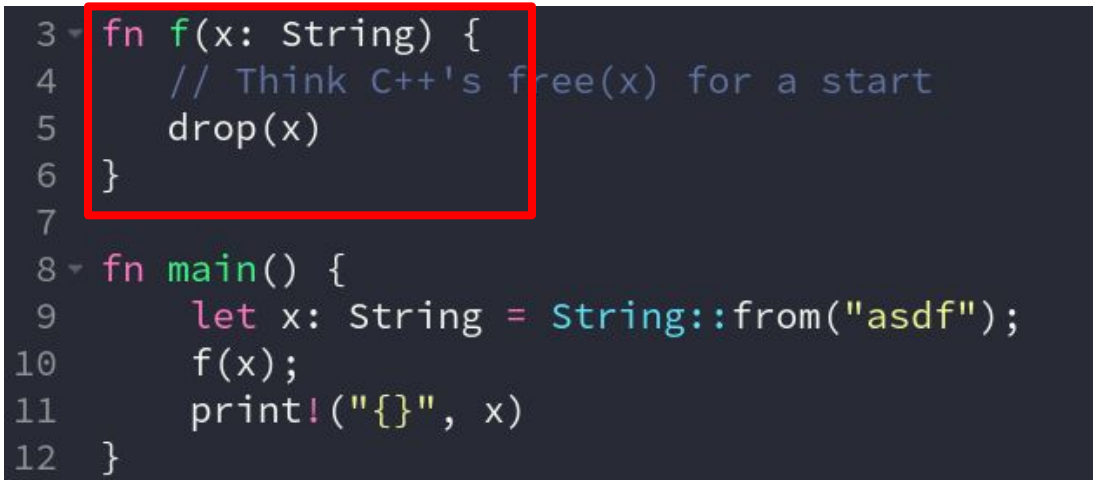
```
Program returned: 1
Program stderr
=================================================================
==1==ERROR: AddressSanitizer: alloc-dealloc-mismatch (operator new vs free) on 0x60300
    #0 0x7efc3e357898  (/opt/compiler-explorer/gcc-12.2.0/lib64/libasan.so.8+0xba898)
    #1 0x40236d in f(std::__cxx11::basic_string<char, std::char_traits<char>, std::all
    #2 0x402453 in main /app/example.cpp:12
    #3 0x7efc3dd37082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082)
    #4 0x40229d in _start (/app/output.s+0x40229d)

0x603000000010 is located 0 bytes inside of 32-byte region [0x603000000010,0x603000000
allocated by thread T0 here:
    #0 0x7efc3e3596b8 in operator new(unsigned long) (/opt/compiler-explorer/gcc-12.2.
    #1 0x4023f9 in main /app/example.cpp:11
    #2 0x7efc3dd37082 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24082)

SUMMARY: AddressSanitizer: alloc-dealloc-mismatch (/opt/compiler-explorer/gcc-12.2.0/l
==1==HINT: if you don't care about these errors you may set ASAN_OPTIONS=alloc_dealloc
==1==ABORTING
```

The Challenge: Performance + Guaranteed Safety

- Translated to Rust: what happens? [np]



The Challenge: Performance + Guaranteed Safety

- **Translated to Rust: what happens? [np]**
  - https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=336f34516c2edf45a31694facfcf955b

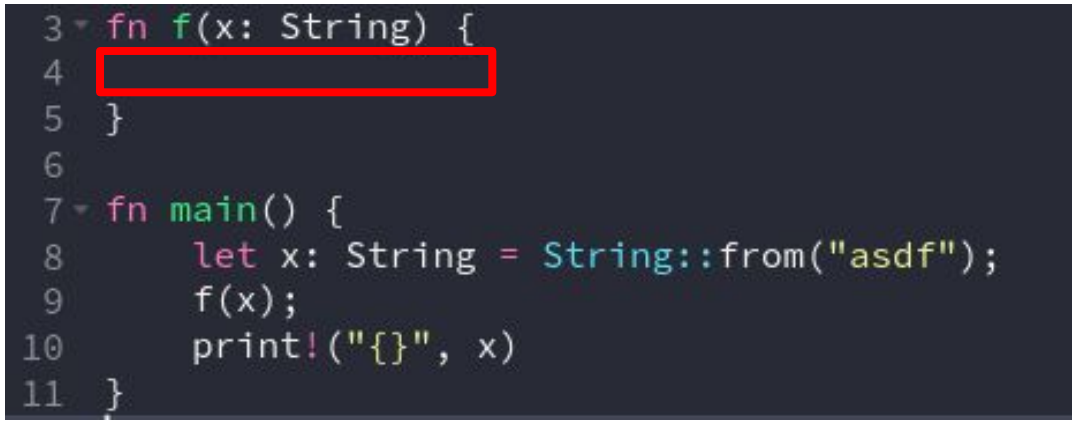
```rust
fn f(x: String) {
    // Think C++'s free(x) for a start
    drop(x)
}

fn main() {
    let x: String = String::from("asdf");
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:11:18
   |
9  |     let x: String = String::from("asdf");
   |         - move occurs because `x` has type `String`, which does not implement the `Copy` tr
10 |     f(x);
   |       - value moved here
11 |     print!("{}", x)
   |                  ^ value borrowed here after move
   |
```

The Challenge: Performance + Guaranteed Safety

- What about this? [p]



The Challenge: Performance + Guaranteed Safety

- What about this? [p]
  - Same issue!
  - But why??

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=6ecc6467b2aafdf94405647a201bf570

```
3  fn f(x: String) {
4
5  }
6
7  fn main() {
8      let x: String = String::from("asdf");
9      f(x);
```

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:11:18
   |
9  |     let x: String = String::from("asdf");
   |         - move occurs because `x` has type `String`, which does not implement the `Copy` tr
10 |     f(x);
   |       - value moved here
11 |     print!("{}", x)
   |                  ^ value borrowed here after move
   |
```

The Challenge: Performance + Guaranteed Safety

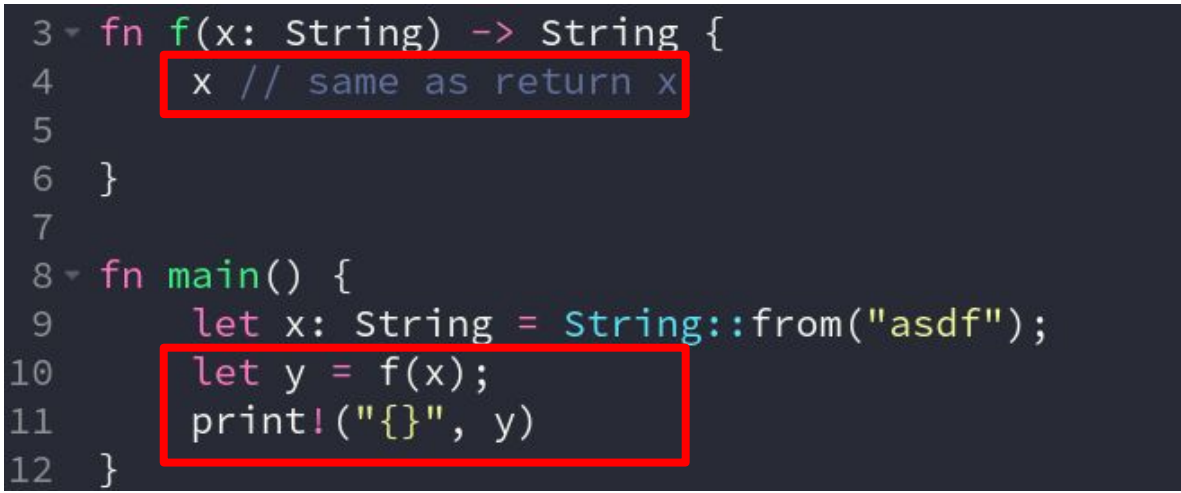
- What about this? [p]
  - Same issue!
  - Too hard to decide validity of ownership on case-by-case basis, the strict rules might lead to some “valid + safe” code to be rejected => no false positives

```
3  fn f(x: String) {
4
5  }
6
7  fn main() {
8      let x: String = String::from("asdf");
9      f(x);
10     print!("{}", x)
11 }
```

```
3  fn f(x: String) {
4      if (based_on_user_input_at_runtime) {
5          drop(x)
6      }
7  }
```

The Challenge: Performance + Guaranteed Safety

- What about this? [p]



The Challenge: Performance + Guaranteed Safety

- What about this? [p]
  - Totally valid!
  - Ownership of string passed to f
  - f owns the string, so ownership can be passed back through return value
    - String is “move”d
  - Still only *one explicit owner*
  - https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=2701de5060e894e20cc960a57a5bb82a

```
3  fn f(x: String) -> String {
4      x // same as return x
5
6  }
7
8  fn main() {
9      let x: String = String::from("asdf");
10     let y = f(x);
11     print!("{}", y)
12 }
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.46s
     Running `target/debug/playground`

asdf
```

What did we just learn?

Concept of “Ownership”

Governs compiler-checked automatic memory management **without GC**

```
3  fn f(x: String) {
4      // Think C++'s free(x) for a start
5      drop(x)
6  }
7
8  fn main() {
9      let x: String = String::from("asdf");
10     f(x);
11     print!("{}", x)
12 }
```

- 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.

The Challenge: Performance + Guaranteed Safety

- Working with owners: **borrowing**

- **&T:** Temporary **immutable** reference to the owned object – **there can be many at a time!**

```
3  fn f(x: &Vec<u8>) {
4      print!("Borrowed version of x in f: {:?}\n", x)
5  }
6
7  fn main() {
8      let x: Vec<u8> = vec![1, 2, 3, 4];
9      print!("x from start of main: {:?}\n", x);
10     f(&x);
11     print!("x from end of main: {:?}\n", x);
12 }
```

```
x from start of main: [1, 2, 3, 4]
Borrowed version of x in f: [1, 2, 3, 4]
x from end of main: [1, 2, 3, 4]
```

The Challenge: Performance + Guaranteed Safety

- Working with owners: immutable **borrowing**

```
3  fn f(x: &Vec<u8>) {
4      print!("Borrowed version of x in f: {:?}\n", x)
5  }
6
7  fn main() {
8      let x: Vec<u8> = vec![1, 2, 3, 4];
9      print!("x from start of main: {:?}\n", x);
10     f(&x);
11     print!("x from end of main: {:?}\n", x);
12 }
```

- **&T:** Temporary **immutable** reference to the owned object - **there can be many at a time!**

```
x from start of main: [1, 2, 3, 4]
Borrowed version of x in f: [1, 2, 3, 4]
x from end of main: [1, 2, 3, 4]
```

Why print! macro doesn't own the arguments: https://stackoverflow.com/questions/30450399/does-println-borrow-or-own-the-variable

The Challenge: Performance + Guaranteed Safety

- **&mut T:** Temporary **mutable** reference to the owned object: **there can only be one at a time!**
  - **If there is a valid &mut T, then no &T allowed simultaneously**

```
3  fn f(x: &mut Vec<u8>) {
4      x[1] = 244;
5      print!("Borrowed version of x in f: {:?}\n", x)
6  }
7
8  fn main() {
9      let mut x: Vec<u8> = vec![1, 2, 3, 4];
10     print!("x from start of main: {:?}\n", x);
11     f(&mut x);
12     print!("x from end of main: {:?}\n", x);
13 }
```

```
x from start of main: [1, 2, 3, 4]
Borrowed version of x in f: [1, 244, 3, 4]
x from end of main: [1, 244, 3, 4]
```

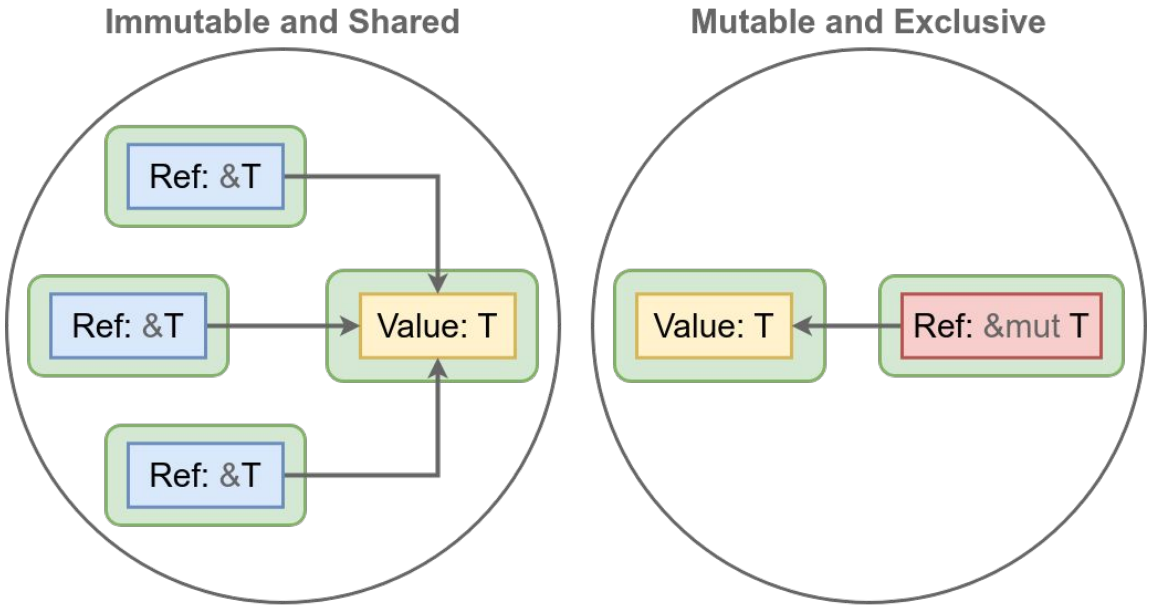
Extra Challenge

```
let mut x = vec![1,2,3];
let first = x[0];
let mut second = x[1];
second += 1;
```

Is this allowed?

What did we just learn?

Concept of “Borrowing”



(Exclusivity guaranteed at compile time)

Lifetimes

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=bcf4d2ddd32727d47f9a0171eadd184e S

- This rust code doesn't compile!

- Problem: **what is the appropriate lifetime of the "borrowed" return value ? Will it "live as long" as our inputs?**

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

https://github.com/dtolnay/rust-faq#why-arent-function-signatures-inferred

Lifetimes

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=f204b41edfcc053a809d5e27175dcb0f

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- We use **lifetimes** to tell the compiler here: **the output str will live as long as the inputs (which have the same lifetime)**
- **Now the compiler can go and verify this (as part of its type system)!**

Lifetimes

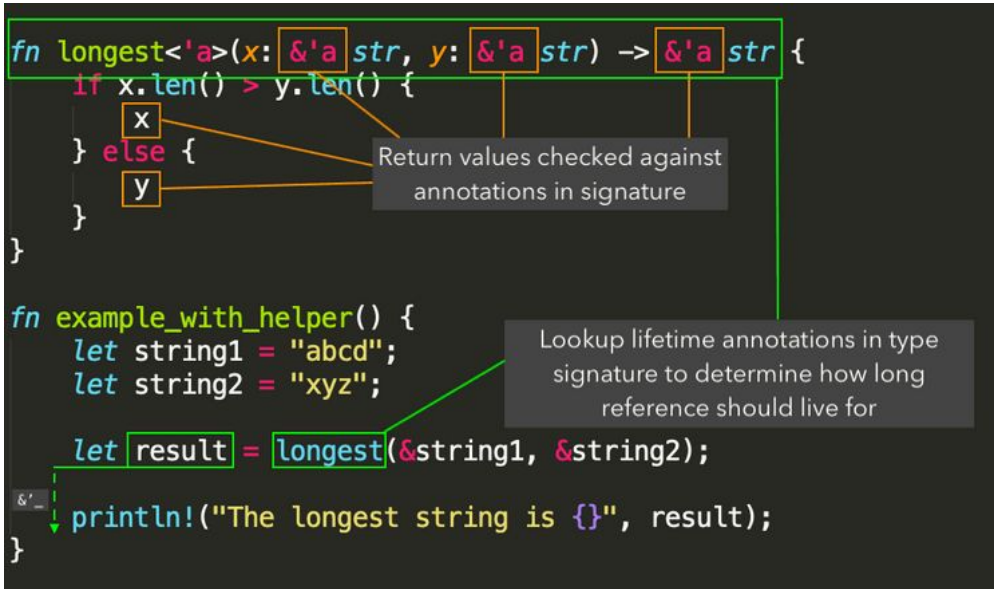
https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0cb691feed7b15c04e1e5a6e90d41cc7

```rust
fn print_longest_return_first_arg<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if x.len() > y.len() {
        print!("X is longer!\n")
    } else {
        print!("Y is longer!\n")
    }
    x
}
```

- An example of **multiple lifetimes** that the compiler can verify!
- We only return the first argument, so we can specify that the **return value follows the lifetime of the first argument**

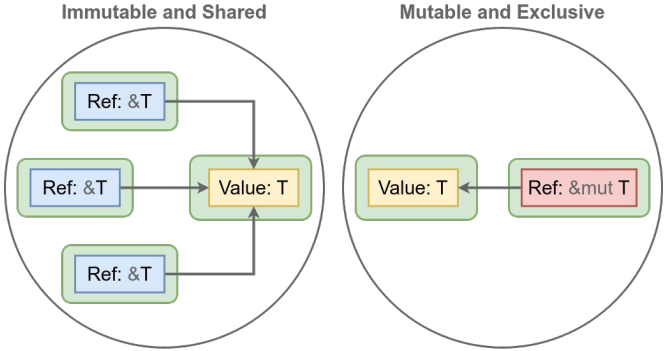
What did we just learn?

- **Lifetimes:** proving to the compiler that we always have valid references in our program!



TLDR – First pass of interesting parts of Rust

- Ownership
- Borrowing (Mutable & Immutable)
- Lifetimes




```
3  fn f(x: String) {
4      // Think C++'s free(x) for a start
5      drop(x)
6  }
7
8  fn main() {
9      let x: String = String::from("asdf");
10     f(x);
11     print!("{}", x)
12 }
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn example_with_helper() {
    let string1 = "abcd";
    let string2 = "xyz";

    let result = longest(&string1, &string2);

    println!("The longest string is {}", result);
}
```

Return values checked against annotations in signature

Lookup lifetime annotations in type signature to determine how long reference should live for

Tutorial: (1) Concurrent Counter

1. Concurrent Counter

- **Recall:** Our data-race / undefined behavior C++ code to add to a counter from multiple threads

```
int counter;

int main() {
  std::thread t0{[]() { ++counter; }};
  std::thread t1{[]() { ++counter; }};

  t0.join();
  t1.join();

  std::cout << counter << std::endl;

  return 0;
}
```

1. Concurrent Counter

- Direct translation to Rust has compile errors! Let's try to fix...

```rust
use std::thread;

fn main() {
    let mut counter = 0;

    let t0 = thread::spawn(|| { counter += 1; });
    let t1 = thread::spawn(|| { counter += 1; });

    t0.join();
    t1.join();

    println!("{}", counter);
}
```

```
error[E0373]: closure may outlive the current function, but it borrows
 --> src/main.rs:7:26
  |
7 |     let t1 = thread::spawn(|| { counter += 1; });
  |                            ^^   ------- `counter` is borrowed here
  |                            |
  |                            may outlive borrowed value `counter`
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:7:12
  |
7 |     let t1 = thread::spawn(|| { counter += 1; });
  |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `counter` (and any oth
  |
7 |     let t1 = thread::spawn(move || { counter += 1; });
  |                            ++++

error[E0502]: cannot borrow `counter` as immutable because it is also
 --> src/main.rs:12:18
  |
6 |     let t0 = thread::spawn(|| { counter += 1; });
  |              ----------------------------------
  |              |             |    |
  |              |             |    first borrow occurs due to use of
  |              |             mutable borrow occurs here
  |              argument requires that `counter` is borrowed for `'st
```

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=19d09528931f6431500585c66cd49661

1. Concurrent Counter

- Let's listen to the compiler
- Output of this program? [p]

```
6 |   let t0 = thread::spawn(|| { counter += 1; });
  |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `counter` (and any other referenced variables),
  |
6 |   let t0 = thread::spawn(move || { counter += 1; });
  |                          ++++
```

```rust
fn main() {
    let mut counter = 0;

    let t0 = thread::spawn(move || { counter += 1; });
    let t1 = thread::spawn(move || { counter += 1; });

    t0.join();
    t1.join();

    println!("{}", counter);
}
```

1. Concurrent Counter

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=738271fb046792be74c3aaf10a0c8008

- Let's listen to the compiler (but not the right warning :))
- Output of this program? [p] (Outputs 0!)
- Each thread tries to move counter into its scope (to own it)
  - Counter is *Copyable* so each thread just makes a private copy of counter!!

```
3  fn main() {
4      let mut counter = 0;
5
6      let t0 = thread::spawn(move || { counter += 1; });
7      let t1 = thread::spawn(move || { counter += 1; });
8
9      t0.join();
10     t1.join();
11
12     println!("{}", counter);
13 }
```

Each thread now has a *local* variable called counter that they increment

1. Concurrent Counter

- **Problem 1:** counter is **shared and mutable** – not allowed!
- **Problem 2:** compiler thinks threads **might outlive counter!**
  - **Possible dangling reference to counter?**

```rust
use std::thread;

fn main() {
    let mut counter = 0;

    let t0 = thread::spawn(|| { counter += 1; });
    let t1 = thread::spawn(|| { counter += 1; });

    t0.join();
    t1.join();

    println!("{}", counter);
}
```

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=19d09528931f6431500585c66cd49661

1. Concurrent Counter

- Fixing problem 1: put our counter in a Mutex!
- **Problem?**

```
1  use std::thread;
2  use std::sync::{Mutex};
3
4  fn main() {
5      let counter: Mutex<i32> = Mutex::new(0);
6      let t0 = thread::spawn(|| *counter.lock().unwrap() += 1);
7      let t1 = thread::spawn(|| *counter.lock().unwrap() += 1);
8
9      t0.join();
10     t1.join();
11     println!("{}", *counter.lock().unwrap());
12 }
```

1. Concurrent Counter

```
error[E0373]: closure may outlive the current function, but it bo
 --> src/main.rs:6:26
  |
6 |   let t0 = thread::spawn(|| *counter.lock().unwrap() += 1);
  |                          ^^  ------- `counter` is borrowed he
  |                          |
  |                          may outlive borrowed value `counter`
  |
note: function requires argument type to outlive `'static`
```

- Problem 2 appears: compiler is not convinced that our threads finish before the end of main(), when counter is dropped

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=b22193c1e9298500fd432a055ace0497

```rust
use std::thread;
use std::sync::{Mutex};

fn main() {
    let counter: Mutex<i32> = Mutex::new(0);
    let t0 = thread::spawn(|| *counter.lock().unwrap() += 1);
    let t1 = thread::spawn(|| *counter.lock().unwrap() += 1);

    t0.join();
    t1.join();
    println!("{}", *counter.lock().unwrap());
}
```

1. Concurrent Counter

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=fe97826e618ddbdea7f4c8c63b33aaa3

- Fixing problem 2: wrap the mutex in a **shared-pointer equivalent ("Atomically Reference Counted") type**
- The counter will be dropped when both references die - **so the compiler is satisfied!**

```
4  fn main() {
5      let counter = Arc::new(Mutex::new(0));
6      let counter1 = counter.clone();
7      let counter2 = counter.clone();
8      let t0 = thread::spawn(move || { *counter1.lock().unwrap() += 1 });
9      let t1 = thread::spawn(move || { *counter2.lock().unwrap() += 1 });
10
11     t0.join();
12     t1.join();
13     println!("{}", *counter.lock().unwrap());
```

Why does this question matter?

- Rust will literally **prevent you** from writing non-thread-safe code!
- The compiler is not some genius oracle
- But knows that if you follow a set of (more restrictive that necessary) rules, your program will be safe

Safely writing code that isn't thread-safe
An under-appreciated Rust feature

http://cliffle.com/blog/not-thread-safe/

Tutorial: (2) Scoped Threads

Or: do we really need Arc?

2. Scoped Threads

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=79d8080575170cb2972a25bae9b69ef5

- Compiler not smart enough! (w.r.t scope of counter vs threads)
- We can help it by **using an automatically scoped thread (must join before the end of the scope)** – think std::jthread from C++

```
4  fn main() {
5      let counter = Mutex::new(0);
6      thread::scope(|s| {
7          s.spawn(|| *counter.lock().unwrap() += 1);
8          s.spawn(|| *counter.lock().unwrap() += 1);
9      });
10
11     println!("{}", *counter.lock().unwrap());
12 }
```

Tutorial: (3) Interior Mutability

3. Interior Mutability

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=79d8080575170cb2972a25bae9b69ef5

- Doesn't it seem crazy that we can do this?
- Two areas of code can **mutably** change a **non-mutable** reference!

```
4  fn main() {
5      let counter = Mutex::new(0);
6      thread::scope(|s| {
7          s.spawn(|| *counter.lock().unwrap() += 1);
8          s.spawn(|| *counter.lock().unwrap() += 1);
9      });
10
11     println!("{}", *counter.lock().unwrap());
12 }
```

What is a Mutex type in Rust?

1. An actual mutex to protect in data
2. And an **UnsafeCell** that contains our data

```rust
pub struct Mutex<T: ?Sized> {
    inner: sys::Mutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

```rust
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

What is a Mutex type in Rust?

- When we deference the mutex after locking...
- The Mutex library code uses **unsafe** Rust to get a **mutable reference** from the **UnsafeCell**!
  - It's OK: we know that a locked mutex allows only 1 thread to access the data
  - **But the compiler isn't a human and we need to disable some checks with unsafe**

```
pub struct Mutex<T: ?Sized> {
    inner: sys::Mutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

```
impl<T: ?Sized> Deref for MutexGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &T {
        unsafe { &*self.lock.data.get() }
    }
}
```

```
pub const fn get(&self) -> *mut T {
    // We can just cast the pointer from `UnsafeCell<T>` to `T` because of
    // #[repr(transparent)]. This exploits std's special status, there is
    // no guarantee for user code that this will work in future versions of the compiler!
    self as *const UnsafeCell<T> as *const T as *mut T
}
```

Unsafe...

- As long as the library is written correctly, we can still have guarantees in our program

- **Task: can we use atomics to avoid Mutexes?**

3.1 Atomics version

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=c478db74d562120e42f8e5b04138ccff

- Almost identical to C++!
- Quick aside: why is Relaxed OK here? [np]

```rust
use std::sync::atomic::{AtomicI32, Ordering};
use std::thread;

fn main() {
    let counter = AtomicI32::new(0);

    thread::scope(|s| {
        s.spawn(|| {
            counter.fetch_add(1, Ordering::Relaxed);
        });
        s.spawn(|| {
            counter.fetch_add(1, Ordering::Relaxed);
        });
    });

    println!("{}", counter.load(Ordering::Relaxed));
}
```

Atomics

Rust pretty blatantly just inherits the memory model for atomics from C++20.

Tutorial: (4) Static Items

4 Static version

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=5ed58ec8a4b5c11cbc6b3d7e5de65fcb

- No need scoped threads now
- Compiler can **prove that COUNTER will last for the whole program!**

```rust
use std::thread;
use std::sync::atomic::{AtomicI32, Ordering};

static COUNTER: AtomicI32 = AtomicI32::new(0);

fn main() {
  let t0 = thread::spawn(|| { COUNTER.fetch_add(1, Ordering::Relaxed); });
  let t1 = thread::spawn(|| { COUNTER.fetch_add(1, Ordering::Relaxed); });

  t0.join().unwrap();
  t1.join().unwrap();

  println!("{}", COUNTER.load(Ordering::Relaxed));
}
```

Tutorial: (5) Rayon

5. Rayon

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=82d3cb263bd0ba2e59f5db8608bac675

- Imagine this trivially parallelizable program

```rust
fn magic_sum(from: u128, to: u128) -> u128 {
    (from..to).filter(|i| i % 7 == i % 5).sum()
}

fn main() {
    let (from, to) = {
        // Comment out the line below if you are using the Rust Playground
        let mut args = std::env::args();
        // Use the line below instead if you are using the Rust Playground
        // let mut args = ["", "0", "100000000"].iter();
        args.next(); // skip argv[0]
        (args.next().unwrap(), args.next().unwrap())
    };
    println!("{}", magic_sum(from.parse().unwrap(), to.parse().unwrap()));
}
```

5. Rayon

https://play.rust-lang.org/?version=stable&mode=release&edition=2021&gist=54a7f51d6e474074db215fef69f44b60

- Safe parallelism in Rust is as simple as this!

```rust
use rayon::prelude::*;

fn magic_sum(from: u128, to: u128) -> u128 {
    (from..to).into_par_iter().filter(|i| i % 7 == i % 5).sum()
}

fn main() {
    let (from, to) = {
        // Comment out the line below if you are using the Rust Playground
        // let mut args = std::env::args();
        // Use the line below instead if you are using the Rust Playground
        let mut args = ["", "0", "100000000"].iter();
        args.next(); // skip argv[0]
        (args.next().unwrap(), args.next().unwrap())
    };
    println!("{}", magic_sum(from.parse().unwrap(), to.parse().unwrap()));
}
```

Summary

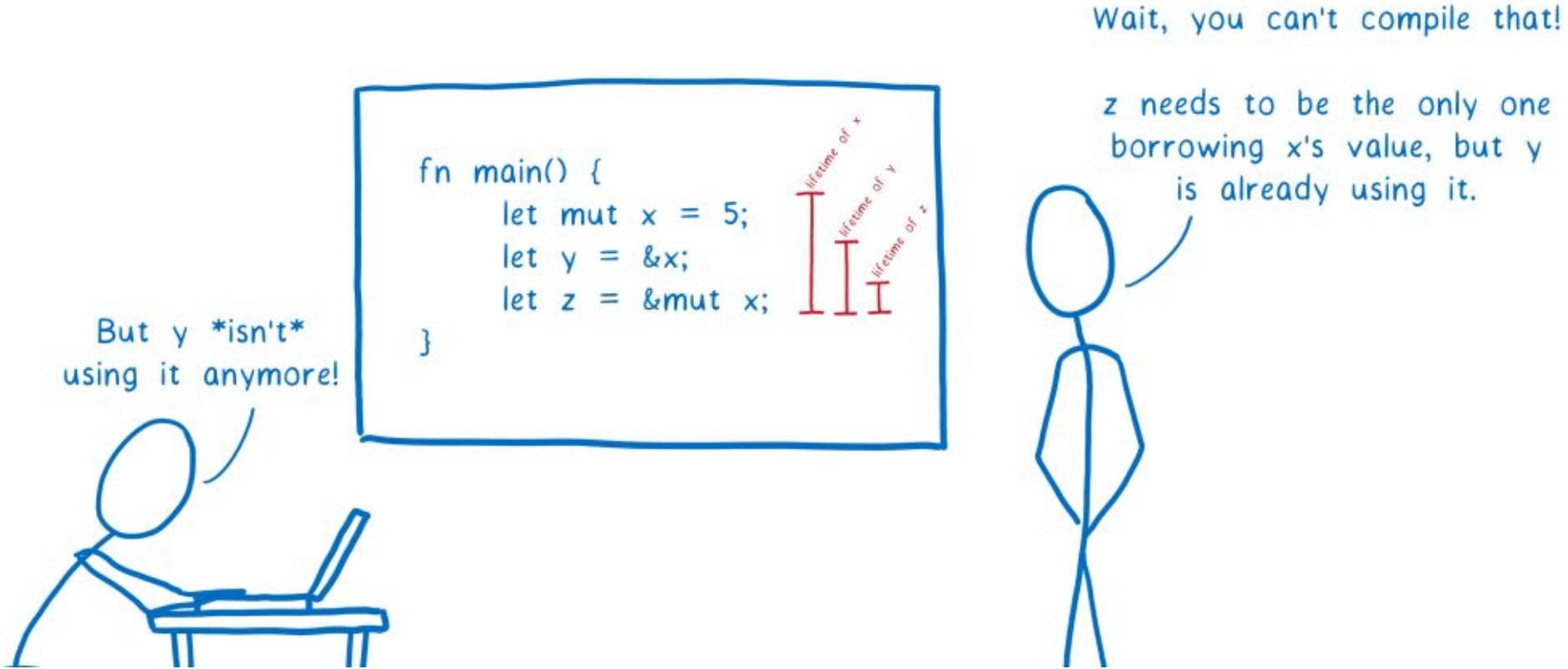
- **Rust** is a language with many **useful guarantees**
  - **No UB, memory safety, type safety…**

- **More rules to be followed** if you want this reward
  - Ownership
  - Borrowing
  - Lifetimes, etc

- Really see the use when doing **concurrent work!**

Extra Quiz

Spot the bug here !!!



Interesting complexities with scopes

- Problem?

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=fe901c4e48353a6f2ca450b9fe10427a

```
4  fn main() {
5      let arr = vec![String::new(); 10];
6      thread::scope(|s| {
7          for i in 0..10 {
8              s.spawn(|| println!("{}", &arr[i]));
9          }
10     });
11 }
```

Interesting complexities with scopes

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=fe901c4e48353a6f2ca450b9fe10427a

- Problem?
  - The thread needs the value i but it might be dropped before the thread(s) start! (end of for loop)
    - Cannot borrow via reference! Must *move* into thread!

```
4  fn main() {
5      let arr = vec![String::new(); 10];
6      thread::scope(|s| {
7          for i in 0..10 {
8              s.spawn(|| println!("{}", &arr[i]));
9          }
10     });
11 }
```

Interesting complexities with scopes

- We can **borrow** one thing (**arr**)
- And **move** the other (i)!
- move statement is very coarse, so we have to handle it ourselves..

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0057fccdfb504a00f183afbe2b91c435

```rust
fn main() {
    let  arr = vec![String::new(); 10];
    thread::scope(|s| {
        // Note that &T is Copy - so a move copies this reference
        let borrowed_arr = &arr; // We can borrow this, and move i
        for i in 0..10 {
            // We can do this too!
            // let borrowed_arr = &arr;
            s.spawn(move || println!("{}", borrowed_arr[i]));
        }
    })
}
```