Rust

Designed for Safety

Carol (Nichols || Goulding)
@carols10cents

- How does Rust prevent bugs?
- What kinds of bugs are still possible?
- Case studies: Would Rust have prevented these problems?

What is Rust?

- Started in 2009
- Sponsored by Mozilla
- Systems language, aiming to replace C and C++
- Fast, concurrent, and safe

One slide of background

How does Rust prevent bugs?

How does Rust prevent bugs?

- **Immutable by default**
- **Array bounds checks**
- **Ownership**
- Strongly typed
- Warnings
- `match` must handle all cases
- Not allowed to use a variable before initializing
- No nullable pointers
- No undefined behavior

Immutable by default

```
let x = 5;
x += 10;
```

Immutable by default

```
let x = 5;
x += 10;
^ error: re-assignment of
immutable variable `x`
```

Immutable by default

```
let mut x = 5;
x += 10;
```

Explicit about what might change
Errors if code tries to change something you thought shouldn’t change.

Array Bounds Checking

```
let buffer = [10, 20, 30];
println!("{}", buffer[100]);
```

Array Bounds Checking

```
let buffer = [10, 20, 30];
println!("{}", buffer[100]);
thread '<main>' panicked at
'index out of bounds: the len
is 3 but the index is 100'
```

Ownership

Ownership

- Memory has one and only one owning scope
- That scope is responsible for freeing the memory

Shared Mutable State

- Use after free
- Double free
- Data Races

Lexical scoping

```
fn main() {
    let x = Thing::new();
    if x.floopy() {
        let y = OtherThing::new(x.florp());
        println!(“y is {}”, y);
    }
}
```

What about calling functions with arguments

Move by default

```
fn main() {
    let x = Thing::new();
    whatevs(x);
    println!(“x is {}”, x);
}

fn whatevs(t: Thing) {
   println!(“I just want to print {}”, t);
}
```

```
fn main() {
    let x = Thing::new();
    whatevs(x);
    println!(“x is {}”, x);
}                         ^ error: use of moved
                          value: `x`
fn whatevs(t: Thing) {
    println!(“I just want to print {}”, t);
}
```

Borrowing

```
fn main() {
    let x = Thing::new();
    whatevs(&x);
    println!(“x is {}”, x); // cool.
}

fn whatevs(t: &Thing) {
    println!(“I just want to print {}”, t);
}
```

Difference: ampersand at caller and callee indicates borrow
whatevs returns thing, does not clean it up
main is still the owner and does clean up

```
fn whatevs(t: &Thing) {
    t.price += 50;
}
```

Borrows are immutable by default too. If instead of printing, we want to change thing

```
fn whatevs(t: &Thing) {
    t.price += 50;
}   ^ error: cannot assign to
    immutable field `t.price`
```

```
fn whatevs(t: &mut Thing) {
    t.price += 50;
}
```

```
fn main() {
    let mut x = Thing::new();
    whatevs(&mut x);
    println!(“x is {}”, x);
}

fn whatevs(t: &mut Thing) {
    t.price += 50;
}
```

Borrowing Rules

- Many immutable borrows
- Only one mutable borrow
  (and no immutable borrows)

```
let mut list = vec![1, 2, 3];
for i in &list {
    println!(“i is {}”, i);
    list.push(i + 1);
}
```

What is this code doing?

```
let mut list = vec![1, 2, 3];
for i in &list {
    println!("i is {}", i);
    list.push(i + 1);
}   ^ error: cannot borrow `list`
    as mutable because it is also
    borrowed as immutable
```

These rules apply when using multiple threads too, and make concurrency safe.

(example left as an exercise for the reader)

BUT WAIT!
THERE'S MORE!

At COMPILE time!*

*except for array bounds checks

Why is this important?
- what affects the cost of fixing a bug
- what if these checks happened at runtime instead

What bugs are still possible?

Is Rust a silver bullet?

unsafe

unsafe keyword lets you...

- Dereference a raw pointer
- Read or write a mutable static variable
- Call an unsafe function

Lets you opt out of the compiler's checks. A way to say "I, the human, have checked this code and found it to uphold the restrictions of safety, so you, the compiler, should let me run whatever is in here"

Rust does not prevent...

- Integer overflows
- Deadlocks
- Non-data race conditions
- Leaking memory
- Failing to call destructors
- Crashing the program

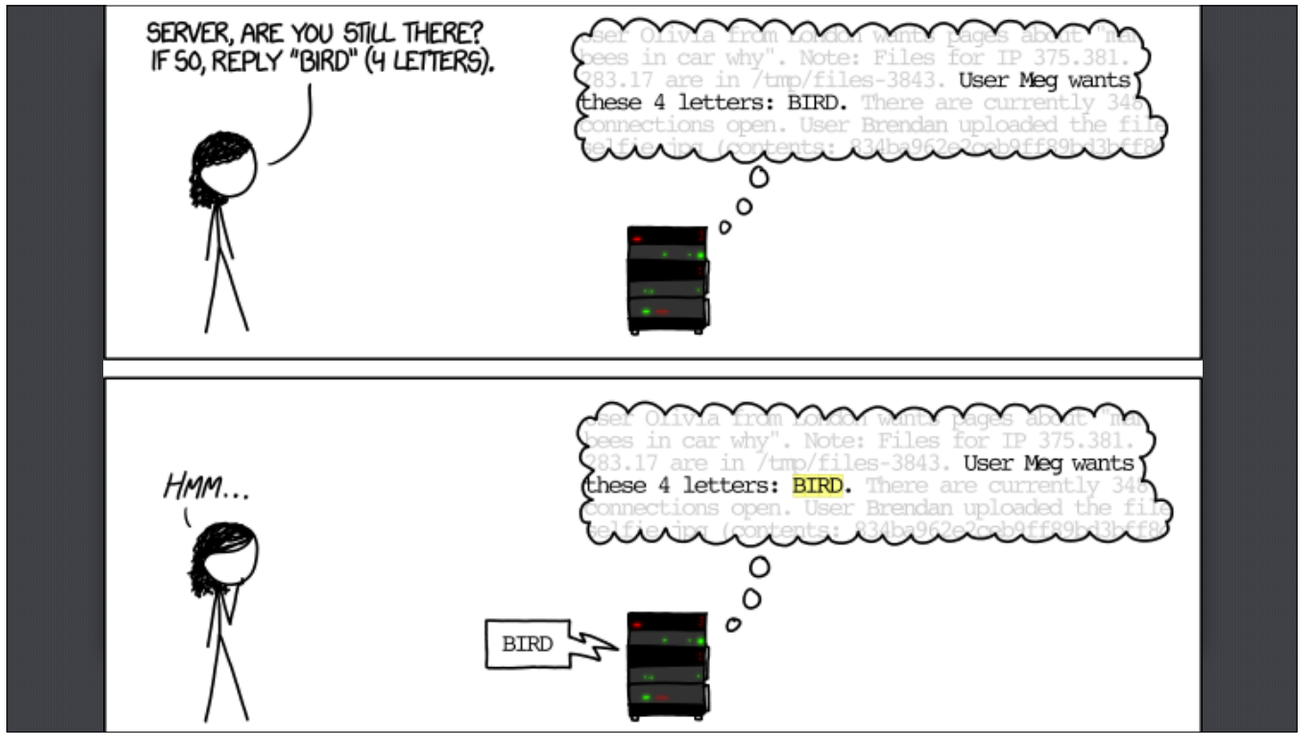
What's a big kind of bug that is still possible in Rust?
Hint: Would you still need to test Rust code?

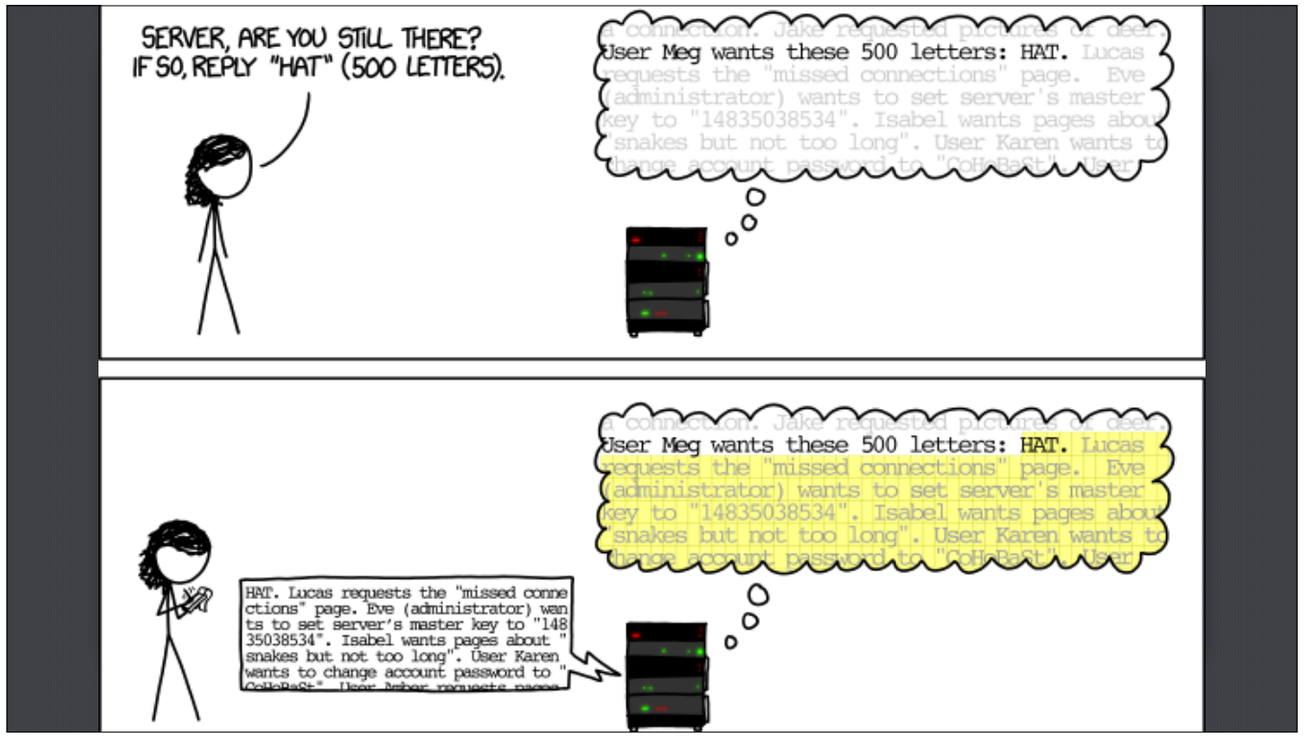
Logic errors

A compiler is never going to be able to check that you're doing the thing that you want to be doing.

Case Studies

Heartbleed
April 2014


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
these 4 letters: BIRD.
HMM...
these 4 letters: BIRD.
BIRD


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.
User Meg wants these 500 letters: HAT.
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

Root causes

Root causes

- Buffer overread
- Did not validate user input
- Reusing a buffer
- Wrote their own memory allocator

- Not idiomatic C

Rust would help prevent this because...

- No out of bound memory reads
- Less convenient to share memory

Would Rust have prevented Heartbleed?

You can write crappy code in any language.

Would Rust have prevented Heartbleed?

Bad Rust < Bad C?

Does Rust make it less likely that you'll write bad code?

Apple `goto fail`
Feb 2014

Verifying a signature of a handshake for a secure connection

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) !=
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) !=
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) !=
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

If something isn't right, we need to set `err` and THEN goto fail.
Skipping some of these checks means invalid things might get through WITHOUT getting the value in `err` that they should have, and will look like success.

- bad alignment
- no curly braces
- `goto` makes it hard to follow code
- dead code

Rust would help prevent this because...

- Required braces around `if` bodies
- Dead code warnings on by default
- No `goto` statement

```
fn verify(certificate: &mut Certificate) {
    if certificate.valid_until() > Date::today() {
        certificate.valid = false;
        return;
    }
        return;
    if certificate.ready_hash(&SSL_HASH_SHA) {
        certificate.valid = false;
        return;
    }
}
```

```
fn verify(certificate: &mut Certificate) {
    if certificate.valid_until() > Date::today() {
        certificate.valid = false;
        return;
    }
        return;
    if certificate.ready_hash(&SSL_HASH_SHA) {
    ^ warning: unreachable statement
        certificate.valid = false;
        return;
    }
}
```

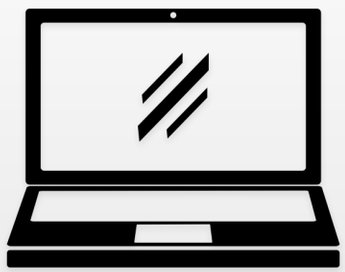
Would Rust have prevented `goto fail`?

Would Rust have prevented `goto fail`?

Probably.

the point?

Computers are good at tedium.



⚠️Beep, boop. You forgot a semicolon in 23,982 places

Created by Dillon Arloff
from Noun Project

Rust has the computer
handle more tedium,
so you can concentrate
on your real problem.

- Rust is not the only tool to do this - we NEED many different ways to make programming correctly easier
- I want to keep seeing innovation like this
- We don’t have to keep creating the same bugs over and over

Questions?

HOMEWORK

HOMEWORK

(just kidding)

is.gd/rust_safety

Want more Rust??

Rust coffee

- Every Wednesday 6:30pm-8:30pm
- Coffee Tree Roasters at Bakery Square
- Work on Rust projects, talk with other people working on Rust projects

Rust Belt Rust Conference

- Thurs Oct 27-Fri Oct 28, 2016
- rust-belt-rust.com
- @rustbeltrust