Intro to Rust Lang

Unsafe

The Story So Far...

- We have covered all of the basic features of Rust, as well as many of the intermediate concepts
- If you are confident you understand the past 11 lectures, you can probably say you are proficient with Rust!

Epilogue

As much as we'd love to dive deep into each of these topics in depth, we simply do not have time.

However...

- The goal of this course was never to feed you information
- The goal was to teach you the *core ideas* of Rust and how to think about it
- We hope that you will take the knowledge from this class and use it to explore more about this programming language *yourself*

Unsafe Rust

Into the Woods

So far, we've only seen code where memory safety is guaranteed at compile time.

- Rust has a second language hidden inside called *unsafe Rust*
- `unsafe` Rust does not enforce memory safety guarantees

Why `unsafe`?

- Static analysis is *conservative*
- By definition, it enforces *soundness* rather than *completeness*
- We need a way to tell the compiler: "Trust me, I know what I'm doing"
- Additionally, computer hardware is inherently unsafe

`unsafe` in 2024

- Rust's precise requirements for `unsafe` code are still being determined
- There's an entire book dedicated to `unsafe` Rust called the Rustonomicon

What is `unsafe`, really?

If you take anything away from today, it should be this:

Unsafe code is the mechanism Rust gives developers for taking advantage of invariants that, for whatever reason, the compiler cannot check.

- *Jon Gjengset, Rust for Rustaceans*

What `unsafe` is not

It's important to understand that `unsafe` is *not* just a way to skirt the rules of Rust.

- Ownership
- Borrow Checking
- Lifetimes
- `unsafe` is a way to *enforce* these rules using reasoning beyond the compiler
- The onus is on *you* to ensure the code is **safe**

The `unsafe` Keyword

There are 2 ways to use the `unsafe` keyword in Rust. The first is marking a *function* as `unsafe`.

```rust
impl<T> SomeType<T> {
    //  vvvvvv
    pub unsafe fn decr(&self) {
        self.some_usize -= 1;
    }
}
```

- Here, the `unsafe` keyword serves as a warning to the caller
- There may be additional invariants that must be upheld *before* calling `decr`

The `unsafe` Keyword

The second way is marking an *expression* as `unsafe`

```
impl<T> SomeType<T> {
    pub fn as_ref(&self) -> &T {
        unsafe { &*self.ptr }
    }
}
```

The `unsafe` Contracts

```
impl<T> SomeType<T> {
    pub unsafe fn decr(&self) {
        self.some_usize -= 1;
    }

    pub fn as_ref(&self) -> &T {
        unsafe { &*self.ptr }
    }
}
```

- The first requires the caller to be careful
- The second assumes the caller *was* careful when invoking `decr`

The `unsafe` Contracts

Imagine is `SomeType<T>` was really `Rc<T>`:

```
impl<T> Rc<T> {
    pub unsafe fn decr(&self) {
        self.count -= 1;
    }

    pub fn as_ref(&self) -> &T {
        unsafe { &*self.ptr }
    }
}
```

- When `self.count` hits 0, `T` is dropped
- What if someone else constructed `&T` without incrementing `count`?
- As long as nobody corrupts the reference count, this code is safe

Unsafe Superpowers

So what can we do with `unsafe`?

With `unsafe`, we get 5 superpowers! We can:

1. Call an `unsafe` function or method
2. Access or modify a mutable static variable
3. Implement an `unsafe` trait
4. Access fields of `union`s

Unsafe Superpowers

1. Call an `unsafe` function or method
2. Access or modify a mutable static variable
3. Implement an `unsafe` trait
4. Access fields of `union`s

These 4 things aren't all that interesting, so why the big fuss?

THE UNSAFE SUPERPOWER

The **biggest** superpower of all is superpower 5!

- **DEREFERENCE A RAW POINTER**
  - That's it!
  - *But honestly, it's enough to wreak all sorts of havoc...*

Raw Pointers

Unsafe Rust has 2 types of Raw Pointers:

- `*const T` is an immutable raw pointer
- `*mut T` is a mutable raw pointer
- *Note that the asterisk* `*` *is part of the type name*
- *Immutable* here means that the pointer can't be reassigned directly after being dereferenced

Pointers vs References

Raw Pointers themselves are allowed to do some special things:

- They can ignore borrowing rules by have multiple immutable and mutable pointers to the same location
- They are not guaranteed to point to valid memory
- They don't implement any automatic cleanup
- They can be `NULL` 💀

Raw Pointers Example

Here's an example of creating raw pointers.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

- We have both an immutable and mutable pointer pointing to the same place
- Notice how there is no `unsafe` keyword here
- We can *create* raw pointers safely, we just cannot *dereference* them

Raw Pointers Example

Here is another example of creating a raw pointer.

```
let address: usize = 0xDEADBEEF;
let r = address as *const i32;
```

- We construct a pointer to (likely) invalid memory
- Again, no `unsafe` keyword necessary here!

Raw Pointers and `unsafe`

Let's actually try and dereference these pointers.

```rust
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

- There's no undefined behavior here? Right?
- *Right?*
- Right! 🦀

Calling `unsafe` Functions

We must also call `unsafe` functions in an `unsafe` block.

```rust
unsafe fn dangerous() {}

fn main() {
    unsafe {
        dangerous();
    }
}
```

- We would get an error if we called `dangerous` without the `unsafe` block!

Using `extern` Functions

Sometimes, we might need to interact with code from another language.

- Rust has the keyword `extern` that facilitates the use of a *Foreign Function Interface (FFI)*
- Since other languages do not have Rust's safety guarantees, we have no idea if they are safe to call or not!

`extern "C"`

Let's see how we would set up integration with the `abs` function from the C standard library.

```rust
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

- The `"C"` defines the *Application Binary Interface (ABI)* that the external function uses
- We have no idea if `abs` is doing what it is supposed to be doing, so it is on us

`extern "C"`

We can also use `extern` to allow other languages to call Rust code!

```rust
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

- Note how the usage of `extern` does not require `unsafe`

Mutable Static Variables

We can mutate global static variables with `unsafe`.

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Last 2 Superpowers

The last 2 superpowers are implementing an `unsafe` trait and accessing fields of a `union`.

- `Send` and `Sync` are both `unsafe` traits
  - The developer must provide their own proof of thread safety
- `union`s are primarily used to interface with unions in C code

How to use `unsafe` code

- Just because a function contains `unsafe` code doesn't mean you need to mark the entire function as `unsafe`
- Often, we want to write `unsafe` code that we *know* is actually safe
- A common abstraction is to wrap `unsafe` code in a safe function

split_at_mut

Let's take a look at `split_at_mut` from the standard library.

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

`split_at_mut`

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]);
```

- Unfortunately, we cannot write this function using only safe Rust
- How would we attempt it?

`split_at_mut` Implementation

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```

- What is the issue with this?
- Can you figure out what the compiler will tell us *just by looking at the function signature*?

`split_at_mut` Compiler Error

If we try to compile, we get this error:

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
 --> src/main.rs:6:31
  |
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
  |                         - let's call the lifetime of this reference `'1`
...
6 |     (&mut values[..mid], &mut values[mid..])
  |     --------------------------^^^^^^--------
  |     |     |                   |
  |     |     |                   second mutable borrow occurs here
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*values` is borrowed for `'1`

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` due to previous error
```

`split_at_mut` Implementation

Let's try again with `unsafe`.

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

`split_at_mut` Implementation

```
unsafe {
    (
        slice::from_raw_parts_mut(ptr, mid),
        slice::from_raw_parts_mut(ptr.add(mid), len - mid),
    )
}
```

- `from_raw_parts_mut` is `unsafe` because it takes a raw pointer and must trust it is valid
- Since the `ptr` came from a valid slice, we know it is valid!

`from_raw_parts_mut` Safety Contract

Here is the actual safety contract for `from_raw_parts_mut`:

```
/// # Safety
///
/// Behavior is undefined if any of the following conditions are violated:
///
/// * `data` must be [valid] for both reads and writes for `len * mem::size_of::<T>()` many bytes,
///   and it must be properly aligned. This means in particular:
///
///     * The entire memory range of this slice must be contained within a single allocated object!
///       Slices can never span across multiple allocated objects.
///     * `data` must be non-null and aligned even for zero-length slices. One
///       reason for this is that enum layout optimizations may rely on references
///       (including slices of any length) being aligned and non-null to distinguish
///       them from other data. You can obtain a pointer that is usable as `data`
///       for zero-length slices using [`NonNull::dangling()`].
///
/// * `data` must point to `len` consecutive properly initialized values of type `T`.
///
/// * The memory referenced by the returned slice must not be accessed through any other pointer
///   (not derived from the return value) for the duration of lifetime `'a`.
///   Both read and write accesses are forbidden.
///
/// * The total size `len * mem::size_of::<T>()` of the slice must be no larger than `isize::MAX`
```

`from_raw_parts_mut` Misuse

We could very easily misuse `from_raw_parts_mut` if we wanted to...

```
use std::slice;

let address: usize = 0xDEADBEEF;
let r = address as *mut i32;

let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

- This might seem ridiculous, but when you always assume your code is safe...

With Great Power...

What could go wrong?

- Probably not much, *if* you're careful
  - By careful, we mean writing a proof for every use of `unsafe`
- If you do get something wrong...
- With `unsafe`, you hold great responsibility

Undefined Behavior

If you get something wrong, your program now has *undefined behavior*.

- It should go without saying that undefined behavior is bad
- The best scenario is you get a visible error:
  - Segfaults
  - Unexpected deadlocks
  - Garbled output
  - Panics that *don't* exit the program
- The worst case...

Undefined Behavior

The worst case scenario is that your program state is invisibly corrupted.

- Data races
- Transactions aren't atomic
- Backups are corrupted
- Security leaks
- Schrödinger’s Bug

Interacting with Safe Rust

Unsafe code is not defined.

- The compiler could eliminate the entire `unsafe` block if it wanted to
- It could also miscompile surrounding, safe code!
- In a lot of ways, `unsafe` Rust is far worse than C/C++ because it assumes *all* of Rust's safety guarantees

Safe `unsafe`: Valid References

You may recall that all references must be valid. A valid reference:

- must never dangle
- must always be aligned
- must always point to a valid value for their target type
- must either be immutably shared or mutably exclusive
- Plus more guarantees relating to lifetimes

Other Validity Requirements

Some primitive types have other guarantees:

- `bool` is 1 byte, but can only hold `0x00` or `0x01`
- `char` cannot hold a value above `char::MAX`
- Most Rust types cannot be constructed from uninitialized memory
- If Rust didn't enforce this, it wouldn't be able to make niche optimizations
  - `Option<&T>` is a good example
  - What if `Option<Option<bool>>` used `0x00` through `0x03`?
- It doesn't matter if Rust does make the optimization, all that matters is that it is *allowed* to whenever it wants

Even More Validity Requirements

Here are some even more requirements:

- Owned Pointer Types (like `Box` and `Vec`) are subject to optimizations assuming the pointer to memory is not shared or aliased anywhere
- You can never assume the layout of a type when casting
- All code must prepared to handle `panic!`s and *stack unwinding*
- Stack unwinding drops everything in the current scope, returns from that scope, drops everything in that scope, returns, etc…
- All variables are subject to something called the *Drop Check*, and if you drop something incorrectly, you might cause undefined behavior

Fighting with `unsafe`

That was a lot, right?

- Remember that it is very possible to write safe `unsafe` code
- A lot of the time, it isn't actually that difficult
- Being careful is half the battle
- Being absolutely sure you actually need `unsafe` is the other half

Working with `unsafe`

It is tempting to reason about unsafety *locally*.

- Consider whether the code in the `unsafe` block is safe in the context of both the rest of the codebase, and in the context of other people using your library
- Encapsulate the unsafety as best you can
- Read and write documentation!
- Use tools like `Miri` to verify your code!
- **Make sure to formally reason about your program**

Miri

Miri is an undefined behavior detection tool for Rust.

- An interpreter for Rust's mid-level intermediate representation
- Can detect out-of-bounds memory accesses and use-after-free
- Invalid use of uninitialized data
- Not sufficiently aligned memory accesses and references

Recap: `unsafe`

- With `unsafe`, we have great powers
- But we must accept the responsibility of leveraging those powers
- There are consequences to writing unsafe `unsafe` code
- `unsafe` is a way to *promise* to the compiler that the indicated code is safe

Next Lecture: Parallelism

Thanks for coming!

Slides created by:

Connor Tsui, Benjamin Owad, David Rudo, Jessica Ruan, Fiona Fisher, Terrance Chen