

A dark, industrial interior, possibly a factory or warehouse, with workers in high-visibility vests and a yellow caution tape in the foreground. The scene is dimly lit, with light coming from windows on the right side. The workers are looking at a laptop or tablet. The overall atmosphere is gritty and technical.

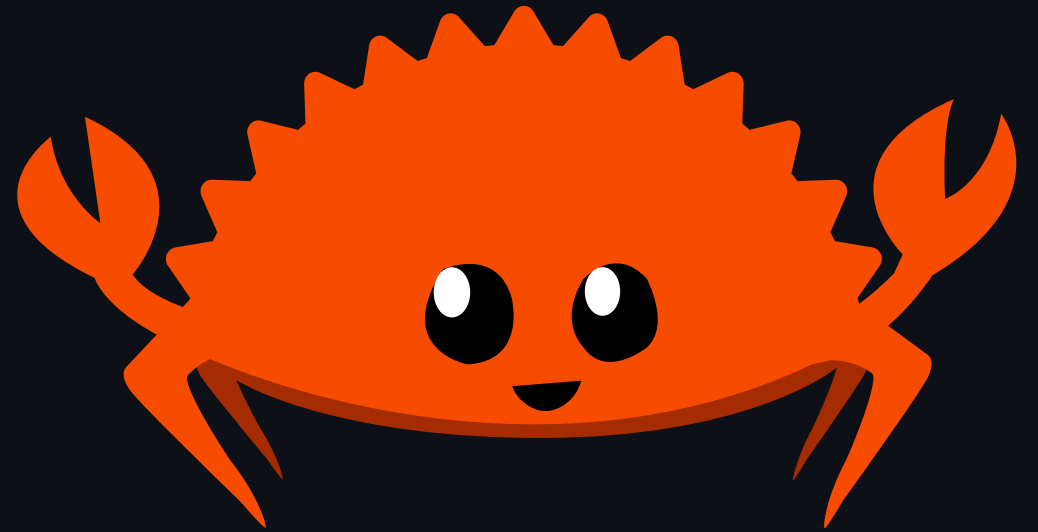
INTRO TO RUST LANG SMART POINTERS AND UNSAFE

Benjamin Owad, David Rudo, and Connor Tsui

Week 10

🦀🦀🦀 We're in week 10! 🦀🦀🦀

Thank you all for sticking with us!



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 9 lectures, you can probably say you are proficient with Rust!
- Now for the *really* interesting stuff...

Finale

Here is the plan for the last ~3.5 lectures:

1. Smart Pointers and `unsafe`
2. Parallelism
3. Concurrency
4. Macros

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*

Final Project

Here are the high-level details about the final project:

- We would like you to spend 6-8 hours developing a project of your choosing
 - *This means a good faith attempt at completing a project*
 - *This bound includes time spent planning and thinking!*
- Your project should incorporate 1 of the 4 advanced topics we will talk about
 - *We can make exceptions if you have a specific idea*
- *If you have less than 400 homework points, you will need to do this*
- More details to come later!

Smart Pointers

- `Rc<T>`
- `RefCell<T>`
- Interior Mutability
- Memory Leaks

Motivation for $Rc \ll T$

Let's Make a List (again)

Let's say we wanted to make recursive-style list with `Box` like before

```
enum List {
  Cons(i32, Box<List>),
  Nil,
}

use crate::List::{Cons, Nil};

fn main() {
  let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
  let b = Cons(3, Box::new(a));
  let c = Cons(4, Box::new(a));
}
```



Cargo's Suggestion

```
Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
9   |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    |     - move occurs because `a` has type `List`, which does not implement the `Copy` trait
10  |     let b = Cons(3, Box::new(a));
    |                               - value moved here
11  |     let c = Cons(4, Box::new(a));
    |                               ^ value used here after move
```

- `Cons` needs to own the data it holds (Recall: `Box`)
- Using `a` again when creating `c`, but `a` has been moved

References?

```
enum List<'a> {  
    Cons(i32, &'a List<'a>),  
    Nil,  
}  
  
use crate::List::{Cons, Nil};  
  
fn main() {  
    let nil = Nil;  
    let a = Cons(10, &nil);  
    let b = Cons(5, &a);  
    let c = Cons(3, &a);  
    let d = Cons(4, &a);  
}
```

- While it can be done, it's a little messy
- Now we have to deal with lifetimes... gross

Introducing `Rc<T>`

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

- Short for reference counter
- keeps track of the number of references to a value to determine dropping

When to use `Rc<T>`

- Allocate data on the heap for *multiple* parts of our program to *read*
 - Can't determine at compile time which part will use the data last
- Only used for single threaded scenarios (We'll talk about `Arc<T>` next week)
- Use `Rc::new(T)` to create a new `Rc<T>`
 - `Rc::clone()` isn't a deep clone, it increments the ref counter

Cloning Demonstrated

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));

    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));

    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
// Rc::strong_count(&a) is now 0, cleaned up and dropped
```

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

Rc<T> Recap

- Great for sharing **immutable** references without lifetimes
- Should be used when last variable to use the data is unknown
 - Otherwise, make that variable the owner and have everything borrow
- Provides almost no overhead
 - O(1) increment of counter
 - Potential allocation/de-allocation on heap

`RefCell<T>` and Interior Mutability

A safe abstraction over unsafe code™

First, `Cell<T>`

```
use std::cell::Cell;

let c1 = Cell::new(5i32);
c1.set(15i32);

let c2 = Cell::new(10i32);
c1.swap(&c2);

assert_eq!(10, c1.into_inner()); // consumes cell
assert_eq!(15, c2.get()); // returns copy of value
```

- Shareable, mutable container
- Move values in and out of cell
- Is used for `Copy` types
 - where copying or moving values isn't too resource intensive
- If an option, should always be used for low overhead

RefCell<T>

- Hold's sole ownership like `Box<T>`
- Allows borrow checker rules to be enforced at **runtime**
 - Interface with `.borrow()` or `borrow_mut()`
 - If borrowing rules are violated, `panic!`
- Typically used when Rust's conservative checking "gets in the way"
- It is **not** thread safe!
 - Use `Mutex<T>` instead

Interior Mutability

```
fn main() {  
    let x = 5;  
    let y = &mut x; // cannot borrow immutable x as mutable  
}
```

- It would be useful for a value to mutate itself in its methods but appear immutable to other code
- Code outside the value's methods wouldn't be able to mutate it
- This can be achieved with `RefCell<T>`

Interior Mutability with Mock Objects

```
pub trait Messenger {  
    fn send(&self, msg: &str); // Note how this takes an &self NOT &mut self  
}  
  
pub struct LimitTracker<'a, T: Messenger> {  
    messenger: &'a T,  
    value: usize,  
    max: usize,  
}
```

- `LimitTracker` tracks a value against a maximum value and sends messages based on how close to the maximum value the current value is
- We want to mock a messenger for our limit tracker to keep track of messages for testing

Limit Tracker

```
impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    // --- snip ---
    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("Warning: You've used up over 75% of your quota!");
        }
    }
}
```

Our Mock Messenger

```
struct MockMessenger {
    sent_messages: Vec<String>,
}

impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger { sent_messages: vec![] }
    }
}

impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        self.sent_messages.push(String::from(message));
    }
}
```

- This code won't compile! `self.sent_messages.push` requires `&mut self`

Let's Use Interior Mutability

```
use std::cell::RefCell;

struct MockMessenger {
    sent_messages: RefCell<Vec<String>>,
}

impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger {
            sent_messages: RefCell::new(vec![]),
        }
    }
}

impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        self.sent_messages.borrow_mut().push(String::from(message));
    }
}
```

Managing Borrows

```
impl Messenger for MockMessenger {  
    fn send(&self, message: &str) {  
        let mut one_borrow = self.sent_messages.borrow_mut();  
        let mut two_borrow = self.sent_messages.borrow_mut();  
  
        one_borrow.push(String::from(message));  
        two_borrow.push(String::from(message));  
    }  
}
```

- We still use the `&` and `mut` syntax for `RefCell`
- `borrow` returns either a `Ref` or `RefMut` which implement `Deref`
 - This means coercion applies: treat them like normal references



What Makes Each Smart Pointer Unique

- `Rc<T>` - Enables multiple owners of the same data
- `Box<T>` - Allows immutable or mutable borrows that are checked at compile time
- `RefCell<T>` - Allows immutable/mutable borrows that are checked at runtime

Combining Smart Pointers: `Rc<RefCell<T>>`

```
#[derive(Debug)]  
enum List {  
    Cons(Rc<RefCell<i32>>, Rc<List>),  
    Nil,  
}
```

- Common type seen in Rust
- Enables multiple owners of mutable data (with runtime checks)
- Extremely powerful, but comes with some overhead

Rc<RefCell<T>> List

```
let value = Rc::new(RefCell::new(5));  
  
let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));  
  
let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));  
let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));  
  
*value.borrow_mut() += 10;  
  
println!("a after = {:?}", a);  
println!("b after = {:?}", b);  
println!("c after = {:?}", c);
```

```
a after = Cons(RefCell { value: 15 }, Nil)  
b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))  
c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

Let's Try Another List

```
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}
```

- This implementation allows modifying the list structure instead of list values
- Now we have a function `tail` that gets the rest of our list

What Happens?

```
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

println!("a initial rc count = {}", Rc::strong_count(&a));
println!("a next item = {:?}", a.tail());

let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

println!("a rc count after b creation = {}", Rc::strong_count(&a));
println!("b initial rc count = {}", Rc::strong_count(&b));
println!("b next item = {:?}", b.tail());

if let Some(link) = a.tail() {
    *link.borrow_mut() = Rc::clone(&b);
}

println!("b rc count after changing a = {}", Rc::strong_count(&b));
println!("a rc count after changing a = {}", Rc::strong_count(&a));

println!("a next item = {:?}", a.tail());
```

Answer

```
Exited with signal 6 (SIGABRT): abort program
```

```
a initial rc count = 1  
a next item = Some(RefCell { value: Nil })  
a rc count after b creation = 2  
b initial rc count = 1  
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })  
b rc count after changing a = 2  
a rc count after changing a = 2  
a next item = Some(RefCell { value: Cons(10, RefCell { value: Cons(5, RefCell...
```

- We see that at the end we have a reference cycle!

Let's Look Closer

```
let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));  
// a is Cons(5, Nil)  
  
let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));  
// b is Cons(10, a) = Cons(10, Cons(5, Nil))  
  
if let Some(link) = a.tail() {  
    // link is Nil (pointed to by a)  
    *link.borrow_mut() = Rc::clone(&b);  
    // link is now b = Cons(10, a)  
}  
// a = Cons(5, link) = Cons(5, b) = Cons(5, Cons(10, a))  
// ^^ reference cycle of a made!
```

- This can cause a memory leak!
 - `Rc` only frees when the `strong_count` is 0

Avoiding Reference Cycles

- We know `Rc::clone` increases the `strong_count`
- You can create a `Weak<T>` reference to a value with `Rc::downgrade`
 - This increases the `weak_count` and can be nonzero when the `Rc` is freed
- To ensure valid references, `Weak<T>` must be upgraded before any use
 - Returns an `Option<Rc<T>>`

Weak<T> Trees

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

Weak<T> Trees In Action

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}" , leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}" , leaf.parent.borrow().upgrade());
} // Tree is effectively dropped even with parent references!
```

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 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`.

```
impl<T> SomeType<T> {  
    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 decre(&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 if `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



- *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 directly be reassigned 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.

```
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

Calling `unsafe` functions is similar, we must call them in an `unsafe` block.

```
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 will not have Rust's safety guarantees, we will 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.

```
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 as the programmer to ensure safety

extern "C"

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

```
#[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`.

- We may or may not cover `unsafe` traits next week
 - (specifically `Send` and `Sync`)
- `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 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`,
///   and adding that size to `data` must not "wrap around" the address space.
///   See the safety documentation of [pointer::offset].
```

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
- If you do get something wrong though...
- 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 be 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**

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!

