Lecture 10

Unsafe Rust

Rust's promise to you

If you satisfy the Rust compiler, your program will never exhibit undefined behavior

- Use after free
- Buffer overflow
- Data race
- Invalid reference

🚨Really bad🚨

Rust programs are still free to encounter bad (but safe!) behavior

- Deadlock
- Leak memory
- Overflow integers
- Abort the program
- Accidentally delete the database
- Panics

Bad, but manageable

Sometimes, safety is too restrictive

Example:

- Python is safe, but at the cost of terrible performance. For performance critical code, write it in C and call from python
  - ```
    import numpy as np
    np.matmul(A,B)
    ```
- Java is safe, but base collections like arrays can't be implemented in the language. They're special-cased by the compiler

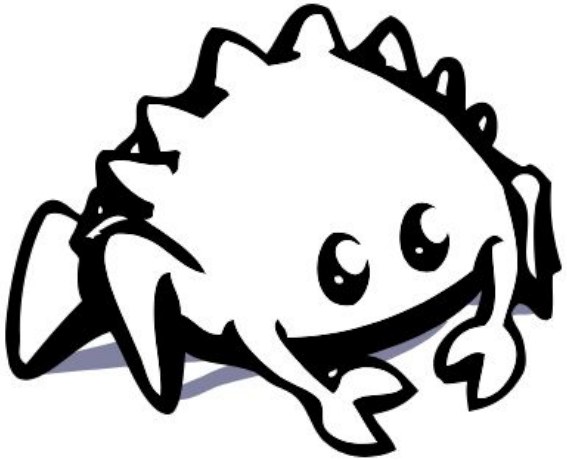
The ergonomics/safety of calling between languages is terrible

- type mismatches
- translating data representation
- etc.

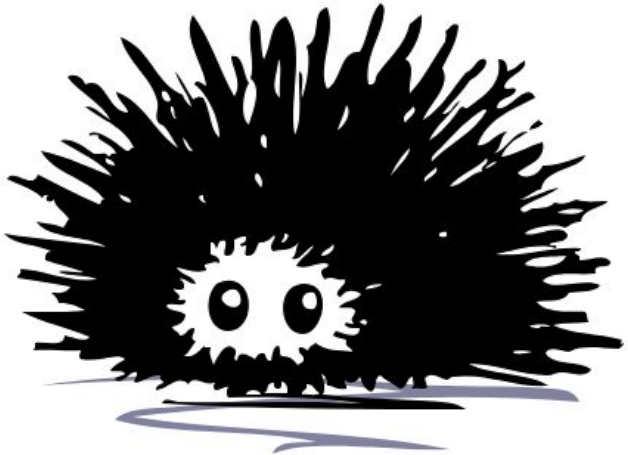
💡 What if we could write safe and unsafe code in the same language?
    ```

Rust contains two languages

Safe Rust



Unsafe Rust



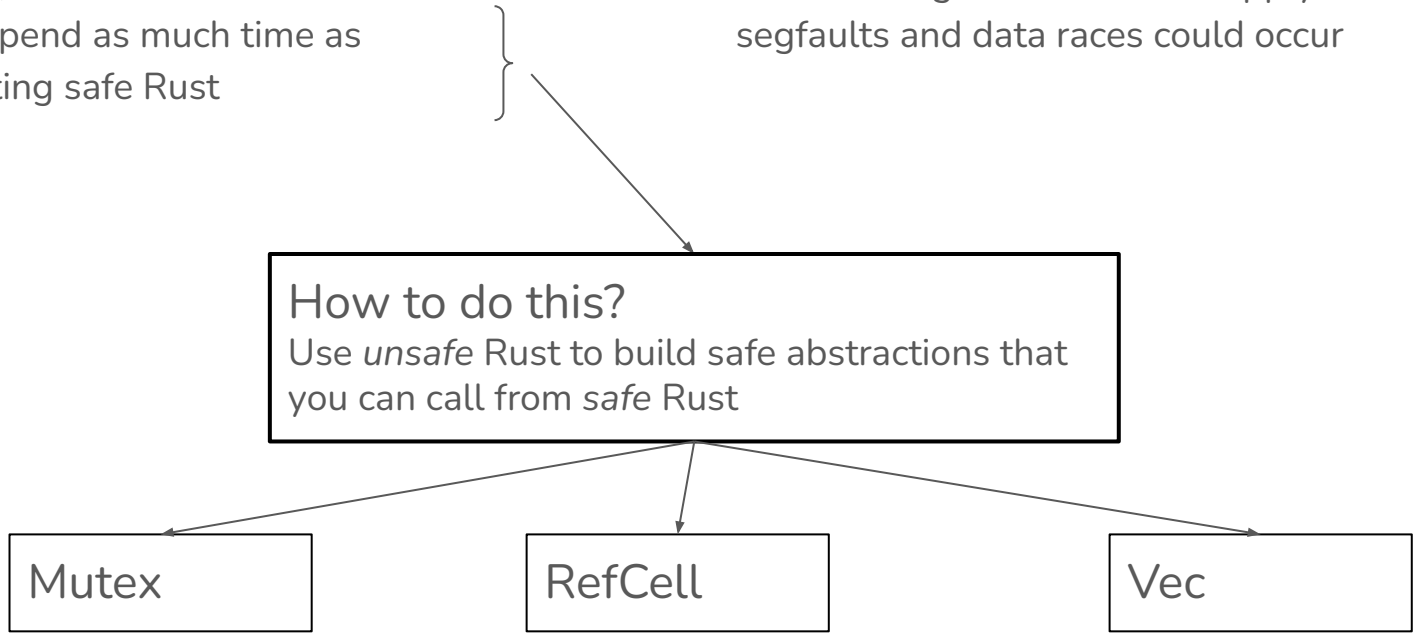
Rust contains two languages

Safe Rust

- Everything we've done up to this point
- If it compiles, it's safe
- You should spend as much time as possible writing safe Rust

Unsafe Rust

- Safe Rust, plus additional powers
- Safe Rust's guarantees don't apply: segfaults and data races could occur



How to build safe abstractions on unsafe?

Two components

- Check to make sure the unsafe is valid
- Perform the unsafe

E.x. get mutable access to separate halves of `Vec`

- Impossible to do in safe Rust; compiler can't verify halves don't overlap

```
pub const fn split_at_mut(v: Vec<T>, mid: usize)
 -> Option<(&mut [T], &mut [T])>
{
```

How to build safe abstractions on unsafe?

Two components

- Check to make sure the unsafe is valid
- Perform the unsafe

E.x. get mutable access to separate halves of `Vec`

- Impossible to do in safe Rust; compiler can't verify halves don't overlap

```
pub const fn split_at_mut(v: Vec<T>, mid: usize)
 -> Option<(&mut [T], &mut [T])>
{
 if mid <= v.len() {
   let len = v.len();
   let ptr = v.ptr();

   unsafe {
     (
       Slice::from_raw(ptr, mid),
       Slice::from_raw(ptr.add(mid), len - mid),
     )
   }
 } else {
   None
 }
}
```

Aside: `unsafe` is poorly named

Unsafe blocks are not for causing undefined behavior

- Dereferencing null pointers is a bad idea regardless

Unsafe blocks are for doing things that the compiler can't verify are safe.

- Better name: `trust_me_this_is_right` blocks

Why should you care about unsafe?

If the goal is to write unsafe Rust as little as possible why should you care?

- You don’t have to! You can write lots of functional, performant Rust code without ever touching unsafe

However, if you want to implement a fast data structure that others depend on, it’s worthwhile to put in extra effort to implement optimizations that require unsafe

- Vec
- Mutex
- Hashmap

Unsafe super powers

In unsafe Rust you can…

- Dereference raw pointers
- Access mutable global variables

- Call unsafe functions
- Implement unsafe traits

That's it! Importantly…

- Ownership still applies
- Reference rules still apply (mutability, validity)

Let's look at examples of using these powers

Dereference raw pointers

References are like pointers, but more restrictive

- Can’t dangle (point to invalid memory)
- Can’t have multiple mutable references

What if we just want plain old pointers?

```
let address: usize = 0x00012345;
let ptr: *const i32 = address as *const i32;
unsafe {
  println!("Value at address: {}", *ptr);
}
```

```
let address: usize = 0x00000000;
let ptr: *mut i32 = address as *mut i32;
unsafe {
   *ptr = 42;
}
```

Dereference raw pointers

References are like pointers, but more restrictive

- Can't dangle (point to invalid memory)
- Can't have multiple mutable references

What if we just want plain old pointers?

Pointers...

- can point to invalid memory
- can be null
- are either mutable or const

```
let address: usize = 0x00012345;
let ptr: *const i32 = address as *const i32;
unsafe {
  println!("Value at address: {}", *ptr);
}
```

```
let address: usize = 0x00000000;
let ptr: *mut i32 = address as *mut i32;
unsafe {
   *ptr = 42;
}
```

Mutable globals

Mutating global variables are unsafe. Why?

```
static mut counter: u32 = 0;

fn main() {
    counter = counter + 1;
}
```

```
error[E0133]: use of mutable static is unsafe and
requires unsafe function or block
--> globals.rs:4:15
  |
4 |     counter = counter + 1;
  |               ^^^^^^^ use of mutable static
```

Mutable globals

Mutating global variables are unsafe. Why?

- data races

```
static mut counter: u32 = 0;

fn main() {
    counter = counter + 1;
}
```

```
error[E0133]: use of mutable static is unsafe and
requires unsafe function or block
--> globals.rs:4:15
  |
4 |     counter = counter + 1;
  |               ^^^^^^^ use of mutable static
 = note: mutable statics can be mutated by multiple
threads: aliasing violations or data races will cause
undefined behavior
```

Mutable globals

How to fix with unsafe?

By convention: provide a `SAFETY` comment that explains how you've manually verified that the code in the `unsafe` block is safe.

```
static mut counter: u32 = 0;

fn main() {
    // SAFETY: this application
    // is single-threaded
    unsafe {
        counter = counter + 1;
    }
}
```

free()

The **free**() function frees the memory space pointed to by ptr, which must have been returned by a previous call to **malloc**() or related functions. Otherwise, or if ptr has already been freed, undefined behavior occurs. If ptr is NULL, no operation is performed.

Unsafe functions

Based on the signature, what do you think is the difference between these functions? Why would you call one or the other?

```
impl Vec<T> {
    pub fn get(&self, index: usize) -> Option<&T>
    pub unsafe fn get_unchecked(&self, index: usize) -> &T
}
```

Unsafe functions

Based on the signature, what do you think is the difference between these functions? Why would you call one or the other?

Out-of-bounds checks are required to maintain safety. `get_unchecked` skips the bounds check.

```
impl Vec<T> {
    pub fn get(&self, index: usize) -> Option<&T>
    pub unsafe fn get_unchecked(&self, index: usize) -> &T
}
```

Unsafe functions

Functions marked `unsafe` don't necessarily use unsafe internally.

`unsafe` on a function means: "calling this function correctly requires upholding properties that the compiler cannot check for you"

Unsafe functions come with a comment saying what properties you need to uphold

```
[–] pub unsafe fn get_unchecked<I>(
```

...

Safety

Calling this method with an out-of-bounds index is *undefined behavior* even if the resulting reference is not used.

Unsafe functions

Is there an inefficiency here?

```
fn sum(v: &Vec<usize>) -> usize {
    let mut counter = 0;
    for i in 0..v.len() {
        counter += v[i];
    }
    counter
}
```

Unsafe functions

Is there an inefficiency here?

```
fn sum(v: &Vec<usize>) -> usize {
    let mut counter = 0;
    for i in 0..v.len() {
        counter += v[i];
    }
    counter
}
```

What's an even better way to write this?

```
fn sum_unsafe(v: &Vec<usize>) -> usize {
    let mut counter = 0;
    for i in 0..v.len() {
        unsafe {
            counter +=
v.get_unchecked(i);
        }
    }
    counter
}
```

Unsafe functions

```rust
fn sum_iter(v: &Vec<usize>) -> usize {
    let mut counter = 0;
    for n in v.iter() {
        counter += n;
    }
    counter
}
```

No bounds check, no unsafe ✅

Unsafe traits

Unsafe function -> need to uphold special properties to **call this function**

Unsafe trait -> need to uphold special properties to **implement this trait**

Only two examples in the standard library

```
pub unsafe trait Send { }
pub unsafe trait Sync { }
```

Improper implementation of Send/Sync will cause a data race, which is undefined behavior

Comparison: `PartialEq`

```
pub trait PartialEq<Rhs> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

Implementations must ensure that `eq` and `ne` are consistent with each other:

- `a != b` if and only if `!(a == b)`.

Comparison: `PartialEq`

```
pub trait PartialEq<Rhs> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

Implementations must ensure that `eq` and `ne` are consistent with each other:

- `a != b` if and only if `!(a == b)`.

```
struct Id(String)
impl std::cmp::PartialEq for Id {
    fn eq(&self, other: &Self) -> bool {
        self.0 == other.0
    }
    fn neq(&self, other: &Self) -> bool {
        self.0 == other.0
    }
}
```

Comparison: `PartialEq`

```
pub trait PartialEq<Rhs> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

Implementations must ensure that `eq` and `ne` are consistent with each other:

- `a != b` if and only if `!(a == b)`.

```
struct Id(String)
impl std::cmp::PartialEq for Id {
    fn eq(&self, other: &Self) -> bool {
        self.0 == other.0
    }
    fn neq
        se
    }
}
```

```
// Requires PartialEq
let mut names = BTreeSet::new();
names.insert(Id("John"));
names.contains(Id("Tim"));
```

Because it's safe to implement `PartialEq`, `BTreeSet` can't rely on the `eq` and `neq` being implemented properly

Unsafe traits

Rust could have `UnsafePartialEq` that is unsafe to implement but allows code to rely on the fact that `eq`/`neq` are implemented correctly

Why would having `UnsafePartialEq` as the default equality trait be better or worse than the current default?

Unsafe traits

Rust could have `UnsafePartialEq` that is unsafe to implement but allows code to rely on the fact that `eq`/`neq` are implemented correctly

Why would having `UnsafePartialEq` as the default equality trait be better or worse than the current default?

Pro:

PartialEq is implemented more often than it's consumed

- Not marking the trait `unsafe` makes things easier for the implementer, but harder for the consumer

Con:

Types like BTreeMap might be leaving performance on the table

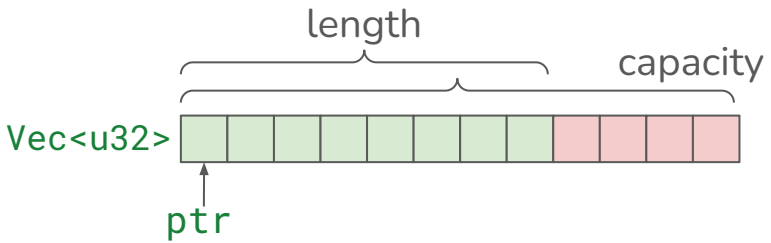
Case Study: implementing Vec

`Vec` is implemented with unsafe code

For full tutorial, see the Rustnomicon

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }
    unsafe {
      ptr::write(self.ptr.add(self.len), elem);
    }
    self.len += 1;
}
```



How is this better than C?

The standard library has lots of unsafe code, so are Rust's guarantees all a lie and we should go back to using C?

No! Separating `unsafe` code greatly reduces the set of code that needs to be manually checked.

- When a segfault occurs, only need to check for bugs in your `unsafe` code

How safe and unsafe interact

Safe code has to trust unsafe code implicitly

- e.g. all safe code **can assume** that `Send` and `Sync` types are implemented properly

Unsafe code can't trust safe code at all

- Unsafe code in BTreeMap **can't assume** that `PartialEq` is implemented properly

Writing `unsafe` code is hard
- You must be defensive

Zooming out

Undefined behavior: what happens when code executes that violates certain rules, like reading from uninitialized memory

Safe Rust: if the program compiles, no undefined behavior will occur

`unsafe` block: allows writing code that compiles, but might cause undefined behavior.

`unsafe` function: function that could cause undefined behavior if you don't check the documentation to see what contracts you must uphold

`unsafe` trait: trait that could cause undefined behavior if you don't check the documentation to see what contracts your type must uphold

Unsafe is hard: panic safety

New trick: panics can be caught

Works similar to try/catch in other languages, but shouldn't be used that way

```
let result = panic::catch_unwind(|| {
    panic!("oh no!");
});
assert!(result.is_err());
```

Panic safety: if your code panics but the panic is caught, are data structures left in a valid state?

Unsafe is hard: panic safety

```
pub fn push(&mut self, elem: T) {
    self.len += 1;
    if self.len == self.cap { self.grow(); }
    unsafe {
      ptr::write(self.ptr.add(self.len), elem);
    }

}
```

Panic safety: if your code panics but the panic is caught, are data structures left in a valid state?

Is this code panic safe?

Unsafe is hard: panic safety

```
pub fn push(&mut self, elem: T) {
    self.len += 1;
    if self.len == self.cap { self.grow(); }
    unsafe {
      ptr::write(self.ptr.add(self.len), elem);
    }

}
```

Panic safety: if your code panics but the panic is caught, are data structures left in a valid state?

Is this code panic safe?

No! If `grow()` panics, the length has been incremented but no new element has been added

Unsafe is hard: panic safety

```
pub fn push(&mut self, elem: T) {
    self.len += 1;
    if self.len == self.cap { self.grow(); }
    unsafe {
      ptr::write(self.ptr.add(self.len), elem);
    }

}
```

Panic safety: if your code panics but the panic is caught, are data structures left in a valid state?

How would we implement `Vec` without unsafe? Do we need to worry about panic safety?

Is this code panic safe?

No! If `grow()` panics, the length has been incremented but no new element has been added

Unsafe is hard: pointer vs. reference

Raw pointers can *dangle*: point to invalid memory

Even in unsafe, references can't dangle

```
impl Vec<T> {
   pub unsafe fn get_unchecked(&self, index:
usize) -> &T
}

fn main() {
   let v: Vec<u32> = Vec::new();
   let first: *const u32 = unsafe {
       v.get_unchecked(0) as *const u32
   };
}
```

Is this code allowed?

Unsafe is hard: pointer vs. reference

Raw pointers can *dangle*: point to invalid memory

Even in unsafe, references can't dangle

No! `get_unchecked` produces a dangling reference, even if it's immediately converted to a pointer

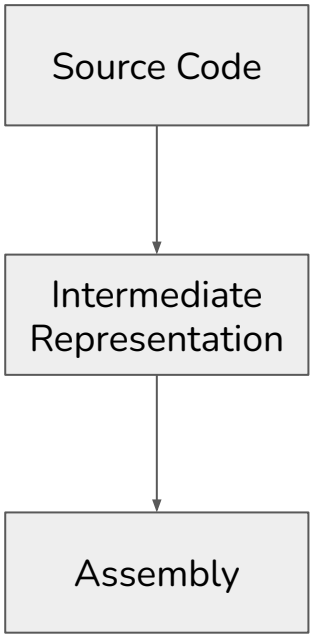
```
impl Vec<T> {
   pub unsafe fn get_unchecked(&self, index:
usize) -> &T
}

fn main() {
   let v: Vec<u32> = Vec::new();
   let first: *const u32 = unsafe {
       v.get_unchecked(0) as *const u32
   };
}
```

Is this code allowed?

Bottom line: writing unsafe is hard, what can we do about it?

Normal Rust compilation



Very easy to analyze
- Type information
- Lifetimes

Hard to analyze, fast to run
- Just reasons about bytes
- No lifetimes or types

Idea:

When testing, instead of running assembly, run the intermediate representation and check for bugs

MIRI

A tool that detects undefined behavior while Rust code runs

Slow to run, so only run your code with MIRI during development

How MIRI works:

Run Rust code in a sandbox

- Track allocation sizes, lifetimes, etc.

If unsafe code causes undefined behavior, print an error.

Real bugs in standard library found with MIRI

- Debug for vec_deque::Iter accessing uninitialized memory
- Vec::into_iter doing an unaligned ZST read
- From<&[T]> for Rc creating a not sufficiently aligned reference

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=b4ca5b55b0c881a3cad3488be9054387

Final Project Ideas

Graphical applications

- Chat app (project 3 as a starting point)
- Music player
- Code editor
- Video game (project 1 as a starting point)

Challenging projects from other domains

- Ray tracer, garbage collector, compiler, database, load balancer, consensus algorithm, file system, scheduler

Misc.

- Write a smart contract in Rust on the Solana blockchain
- Run Rust in the browser using WebAssembly
- Run Rust on Arduino using no_std

Open source contributions

- Build a data structure using unsafe and publish it to crates.io
  - Vec, HashMap, Channel, Mutex
- Implement data science functions in Rust, then allow them to be called from Python and publish the library
- A blazing fast command line tool

Anything else!

- (just check first)