Unsafe Rust

Desperate times call for desperate measures

You can find more here

https://doc.rust-lang.org/nomicon/intro.html

“THE KNOWLEDGE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, [...] INCLUDING [...] UNLEASHING INDESCRIBABLE HORRORS”

“Should you wish a long and happy career of writing Rust programs, you should turn back now and forget you ever saw this book.”

Special things you can do in unsafe Rust

Dereference raw pointers

Call unsafe functions (including C functions, compiler intrinsics, …)

Implement unsafe traits

Access or modify mutable statics

Access fields of unions

Say you want to access two parts of a vector

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

Say you want to access two parts of a vector

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

Say you want to access two parts of a vector

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

Forbidden (two mutable references)

But wait, I *know* the two referenced locations are disjoint

Say you want to access two parts of a vector

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

Forbidden (two mutable references)

But wait, I *know* the two referenced locations are disjoint

Say you want to access two parts of a vector

```rust
use std::slice::from_raw_parts_mut;

pub fn main() {
    let mut v = [1, 2, 3];
    let len = v.len();
    let ptr = v.as_mut_ptr();
    let mid = 1; // let's "cut" the vector at positon 1

    // Safety:
    // data is non-null, valid and properly aligned
    // data points to `len` consecutive properly initialized values
    // The memory referenced by the returned slice is not accessed through any other
pointer
    // The total size len * size_of::<T>() of the slice is not larger than isize::MAX
    // mid <= len
    let (part_one, part_two) =  unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    };

    // part_one and part_two are "independant" now
    let a = &mut part_one[0];
    let b = &mut part_two[1];
    println!("{} {}", a, b);
}
```

```rust
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

Say you want to access two parts of a vector

```rust
use std::slice::from_raw_parts_mut;

pub fn main() {
    let mut v = [1, 2, 3];
    let len = v.len();
    let ptr = v.as_mut_ptr();
    let mid = 1; // let's "cut" the vector at positon 1

    // Safety:
    // data is non-null, valid and properly aligned
    // data points to `len` consecutive properly initialized values
    // The memory referenced by the returned slice is not accessed through any other
pointer
    // The total size len * size_of::<T>() of the slice is not larger than isize::MAX
    // mid <= len
    let (part_one, part_two) =  unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    };

    // part_one and part_two are "independant" now
    let a = &mut part_one[0];
    let b = &mut part_two[1];
    println!("{} {}", a, b);
}
```

Has a safe implementation

```rust
pub fn main() {
    let mut v = [1, 2, 3];
    let mid = 1; // let's "cut" the vector at
positon 1

    let (part_one, part_two) =
v.split_at_mut(mid);

    // part_one and part_two are "independant"
now
    let a = &mut part_one[0];
    let b = &mut part_two[1];
    println!("{} {}", a, b);
}
```

But wait, how can it be safe?

- unsafe block are safe under some precondition
- if a function cannot prove all precondition, **you** have to make the function unsafe
- the preconditions left are moved to the function

Example of safe interface

```
fn split_at_mut_unsafe<'a, T>(slice: &'amut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {




}
```

Example of safe interface

```
fn split_at_mut_unsafe<'a, T>(slice: &'amut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    // Safety:
    // ptr is non-null, valid and properly aligned
    // ptr points to `len` consecutive properly initialized values
    // The memory referenced by the returned slice is not accessed through any other pointer
    // The total size len * size_of::<T>() of the slice is not larger than isize::MAX
    let (part_one, part_two) = unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    };
    (part_one, part_two)
}
```

Example of safe interface

```
fn split_at_mut_unsafe<'a, T>(slice: &'amut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    // Safety:
    // ptr is non-null, valid and properly aligned
    // ptr points to `len` consecutive properly initialized values
    // The memory referenced by the returned slice is not accessed through any other pointer
    // The total size len * size_of::<T>() of the slice is not larger than isize::MAX
    let (part_one, part_two) = unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    };
    (part_one, part_two)
}
```

Example of safe interface

```
fn split_at_mut_unsafe<'a, T>(slice: &'amut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    // Safety:
    // ptr is non-null, valid and properly aligned
    // mid <= len
    // The memory referenced by the returned slice is not accessed through any other pointer
    let (part_one, part_two) = unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    };
    (part_one, part_two)
}
```

Example of safe interface

```
// Safety:
// mid <= len
unsafe fn split_at_mut_unsafe<'a, T>(slice: &'amut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {
   let len = slice.len();
   let ptr = slice.as_mut_ptr();

   // Safety:
   // ptr is non-null, valid and properly aligned
   // The memory referenced by the returned slice is not accessed through any other pointer
   let (part_one, part_two) = unsafe {
       (
           from_raw_parts_mut(ptr, mid),
           from_raw_parts_mut(ptr.add(mid), len - mid),
       )
   };
   (part_one, part_two)
}
```

Example of safe interface

```
fn split_at_mut_safe<'a, T>(slice: &'a mut [T], mid: usize) -> (&'a mut [T], &'a mut [T]) {
   assert!(mid <= slice.len());

   // Safety:
   // mid <= len
   let (part_one, part_two) = unsafe { split_at_mut_unsafe(slice, mid) };
   (part_one, part_two)
}
```

Take home message

You should build safe abstractions
(i.e. safe functions calling unsafe ones)