Demystifying unsafe code

Jon Gjengset
@jonhoo

OR: "IT'S JUST C" VS "CERTAIN DOOM"

When we need unsafe.

Why unsafe?

Lets us write code whose safety relies on **invariants the compiler cannot check**.

Use-cases:

- Working with hardware devices.
- Interacting with external code.
- Writing concurrency primitives.
- Overcoming borrow checker limitations.
- *Performance optimizations*.

Working with hardware devices.

Want to show text on screen during boot.

We know things the compiler does not:

- 0xB8000 is mapped to writeable video memory.
- No-one else is writing to 0xB8000.

```
let vga = unsafe {
  std::slice::from_raw_parts_mut(
    0xB8000 as *mut u8, 80 * 24
  )};
```

Interacting with external code.

Want to call into a C library.

Compiler doesn’t know if the C code does something unsafe!

We are **asserting** that the compiler can trust the C code.

```
extern “C” { fn c_abs(input: i32) -> i32; }
fn main() {
  println!(“{}”, unsafe { c_abs(-42)});
}
```

Writing concurrency primitives.

Want to implement Mutex.

Compiler can’t check that only **one** &mut T exists **at a time**.

```
fn lock(&self) -> MutexGuard<T> {
  while !self.held.compare_and_swap(false, true, SeqCst) {}
  MutexGuard::new(self)
}
impl<T> DerefMut for MutexGuard<T> {
  fn deref_mut(&mut self) -> &mut T {
    unsafe { &mut *self.ptr }
```

Overcoming borrow checker limitations.

Want to return reference early.

Compiler will **currently** reject this valid code.

```
fn next(buffer: &mut String) -> &str {
  loop {
    let event = parse(buffer);
    if true { return event; }
  }}
fn parse(buffer: &mut String) -> &str { … }
```

Performance optimizations.

Want to remove any and all overheads.

Always measure first. **Very rarely worth it.**

```
#[repr(C)]
struct SerializedStruct { … }

unsafe fn cast_deserialize(i: &[u8]) -> &SerializedStruct {
  &*(i.as_ptr() as *const SerializedStruct)
}
```

Unsafe gone wrong.

Why not unsafe?

With unsafe, we tell the compiler that the code is ok, because **we checked it manually**.

That is, we checked that the code **never** violates **any** part of the Rust type system. We asserting that the unsafe code we wrote **is safe**.

All comes down to: **are you sure?**

Why not unsafe?

The compiler assumes **all** code follows Rust safety rules.
That is, it assumes no code **ever**:

- Dereferences dangling/unaligned pointers.
- Violates reference aliasing rules.
- Causes an unsynchronized data race with a write.
- Produces an invalid value.
- … and a few others.

That includes unsafe code!

Effects of incorrect unsafe.

Usually, incorrect unsafe == undefined behavior.

And undefined behavior == russian roulette.

Effect ranges from none to crashes to **arbitrary data corruption**. Effect may not appear today, but appear tomorrow. You have **no guarantees**.

You should **always** avoid undefined behavior.

Examples of things gone wrong.

```
fn compute(i: &u32, o: &mut u32) {
  if *i > 10 {
    *o = 1;
  }
  if *i > 5 {
    *o *= 2;
  }
}
```

```
fn compute(i: &u32, o: &mut u32) {
  let cached_i = *i;
  if cached_i > 10 {
    *o = 2;
  } else if cached_i > 5 {
    *o *= 2;
  }
}
```

In Rust, the compiler is allowed to assume that this optimization is okay!

Examples of things gone wrong.

```
impl<T> Vec<T> {
  fn extend_map<U, F>(&mut self, us: &[U], mut f: F)
  where F: FnMut(&U) -> T {
    self.reserve(us.len()); let cur_len = self.len();
    unsafe { self.set_len(cur_len + us.len()) };
    let into = unsafe { self.as_mut_ptr().add(cur_len) };
    for u in us {
      unsafe { std::ptr::write(into, f(u)) }; into += 1;
    }}}
```

What if f() panics?

Examples of things gone wrong.

```
impl<T> Drop for LazyDropVec<T> {
  fn drop(&mut self) {
    for v in self.drain() {
      unsafe { COLLECTOR.drop_later(v) };
    }
  }
}
```

What if T contains &mut TcpStream and writes on drop?

Examples of things gone wrong.

```
macro_rules! offset_of {
  ($t:path, $field:tt) => {
    let uninit = MaybeUninit::<$t>::uninit();
    let ptr = uninit.as_ptr();
    let fptr = unsafe { &(*ptr).$field as *const _ };
    (fptr as usize) - (ptr as usize)
  }
}
```

What if `$t` is `#[repr(packed)]`?

Integrity of unsafe code.

Safe interfaces to unsafe code must behave correctly
no matter what the safe code does.

- Non-deterministic implementation of Eq
- Broken implementation of Ord
- Weird implementations of Deref.

Can only assume **safety** of safe code, not correctness.

Integrity of "internal" unsafe code.

Safe code in the same module can access non-public things!

Encapsulation of unsafe must happen **at visibility boundary**.

Do not assume callers will remember safety invariants.
Never expose unsafe method as safe, even internally!

Is all hope lost?

No!

It is **possible** to write correct unsafe code.

- Be sure you need it.
- Be, like, really sure.
- Read the **nomicon**.
- Be very careful.
- Document **all** unsafe {}
- Run **miri** & ASAN in CI.

An aside on Miri

An interpreter for Rust's mid-level intermediate representation.

Basically, it can run Rust code **in the compiler**.

Can check that the code doesn't do anything "bad".

(works on the playground!)

```
let mut x: Vec<String> = Vec::new();
x.extend_map(&["foo"], |_| panic!());
```

```
error: Miri evaluation error: type validation failed: encountered
uninitialized bytes, but expected something greater or equal to 1
   --> rust/src/liballoc/raw_vec.rs:190:9
      |
  190 |     self.ptr.as_ptr()
      |     ^^^^^^^^^
      = note: inside call to `alloc::raw_vec::RawVec::<u8>::ptr`
      = note: inside call to `Vec::<u8>::as_mut_ptr`
      = note: inside call to `<Vec<u8> as DerefMut>::deref_mut`
      = note: inside call to `<Vec<u8> as IndexMut>::index_mut`
      = note: inside call to `<Vec<u8> as Drop>::drop`
```

No!

It is **possible** to write correct unsafe code.

- Be sure you need it.
- Be, like, really sure.
- Read the **nomicon**.
- Be very careful.
- Document **all** unsafe {}
- Run **miri** & ASAN in CI.