Unsafe Rust is a necessary ”escape hatch”

- Unsafe means that the **compiler cannot ensure safety** of the code.
  - performs operations that are not allowed by the safety rules
- Consequently, the responsibility for safety falls on the programmer who creates the unsafe code.
- 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 Unsafe?

- 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 (FFI)

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

https://stackoverflow.com/questions/50519147/double-mutable-borrow-error-in-a-loop-happens-even-with-nll-on

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)
}
```

Three primary things the unsafe is used for:

- Dereferencing raw pointers
- Calling functions or methods which are unsafe (including calling a function over FFI)
- Accessing or modifying static mutable variables

Dereferencing Raw Pointers

```
let mut i = 42;
let raw_p: *mut u32 = &mut i; // implicit coercion
// let raw_p = &mut i as *mut u32;

unsafe {
    *raw_p *= 10;
}
println!("i = {i}");
```

Calling unsafe Functions

- A function is marked unsafe to indicate that it is possible to violate memory safety by calling it.

```
unsafe fn from_raw_parts<'a, T>(data: *const T, len: usize) -> &'a [T] {…}
```

```
use std::slice;

let some_vector = vec![1, 2, 3, 4];

let pointer: *const u32 = some_vector.as_ptr();
let length: usize = some_vector.len();

unsafe {
      let my_slice: &[u32] = slice::from_raw_parts(pointer, length);

      assert_eq!(some_vector.as_slice(), my_slice);
}
```

Unsafe Block

```
let node = unsafe {
    Box::from_raw(self.head) // call unsafe function
};
```

- An unsafe function can only be called within an unsafe block.
- The body of an unsafe function is automatically an unsafe block.
- When a raw pointer is being dereferenced, it must be enclosed in an unsafe block.

```
unsafe { (*self.tail).next = new_tail };
```

It is a good programming practice to have unsafe code encapsulated inside safe functions.

Accessing or modifying static mutable variables

- Care should be taken to ensure that modifications to a mutable static are safe with respect to other threads running in the same process.

```
static mut LEVELS: u32 = 0;

// This doesn't internally protect against races,
// so this function is `unsafe`
unsafe fn bump_levels_unsafe1() -> u32 {
    let ret = LEVELS;
    LEVELS += 1;
    ret
}

// Assuming that we have an `atomic_add` function which returns the old value,
// this function is "safe" but the meaning of the return value may not be what
// callers expect, so it's still marked as `unsafe`
unsafe fn bump_levels_unsafe2() -> u32 {
    atomic_add(&mut LEVELS, 1)
}
```

Unsafe Trait

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

unsafe impl<T: Sync + ?Sized> Send for &T {}
unsafe impl<T: Sync> Sync for LinkedList<T> {}
```

- An unsafe trait has the `unsafe` keyword preceding the trait keyword.
- All implementations of an unsafe trait must be marked as `unsafe`.
- Unsafe traits can be a source of confusion and occurs rarely in practice.
  - The methods of the implementation are not automatically unsafe and therefore do not have to be called inside an unsafe block.
  - Also, it is not very clear when a trait needs to be declared as unsafe.

Effects of incorrect unsafe

- It is important to point out that *just because a piece of code is marked as unsafe does not necessarily mean that it is 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.

Astrauskas et al. suggested three principles

1. Unsafe code should be used sparingly.
2. Unsafe code should be straightforward and self-contained.
   - Most of the unsafe blocks are quite small.
   - When an unsafe block calls an unsafe function, the target is mostly in the same crate.
3. Unsafe code should be well encapsulated and documented.

Exercise

- Complete the signature and implementation for the following versions of the ‘**swap**’ function without using any other dependencies. Some signatures aren’t entirely complete.

```
1. fn swap(x: & i32, y: & i32)
2. fn swap<T>(x: & T, y: & T)
```

- Each should swap the values of the two parameters such that, after the function returns, we have:

```
*y = old(*x) && *x = old(*y)
```