Unsafe Rust

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time. However, Rust has a second language hiding inside of it that does not enforce these memory safety guarantees: unsafe Rust. This works just like regular Rust, but gives you extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler is trying to determine if code upholds the guarantees or not, it's better for it to reject some programs that are valid than accept some programs that are invalid. That inevitably means there are some times when your code might be okay, but Rust thinks it's not! In these cases, you can use unsafe code to tell the compiler, "trust me, I know what I'm doing." The downside is that you're on your own; if you get unsafe code wrong, problems due to memory unsafety, like null pointer dereferencing, can occur.

There's another reason Rust has an unsafe alter ego: the underlying hardware of computers is inherently not safe. If Rust didn't let you do unsafe operations, there would be some tasks that you simply could not do. Rust needs to allow you to do low-level systems programming like directly interacting with your operating system, or even writing your own operating system! That's one of the goals of the language. Let's see what you can do with unsafe Rust, and how to do it.

Unsafe Superpowers

To switch into unsafe Rust we use the `unsafe` keyword, and then we can start a new block that holds the unsafe code. There are four actions that you can take in unsafe Rust that you can't in safe Rust that we call "unsafe superpowers." Those superpowers are the ability to:

1. Dereference a raw pointer
2. Call an unsafe function or method
3. Access or modify a mutable static variable
4. Implement an unsafe trait

It's important to understand that `unsafe` doesn't turn off the borrow checker or disable any other of Rust's safety checks: if you use a reference in unsafe code, it will still be checked. The `unsafe` keyword only gives you access to these four features that are then not checked by the compiler for memory safety. You still get

some degree of safety inside of an unsafe block!

Furthermore, `unsafe` does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that you as the programmer will ensure the code inside an `unsafe` block will access memory in a valid way.

People are fallible, and mistakes will happen, but by requiring these four unsafe operations to be inside blocks annotated with `unsafe`, you'll know that any errors related to memory safety must be within an `unsafe` block. Keep `unsafe` blocks small and you'll thank yourself later when you go to investigate memory bugs.

To isolate unsafe code as much as possible, it's a good idea to enclose unsafe code within a safe abstraction and provide a safe API, which we'll be discussing once we get into unsafe functions and methods. Parts of the standard library are implemented as safe abstractions over unsafe code that has been audited. This technique prevents uses of `unsafe` from leaking out into all the places that you or your users might want to make use of the functionality implemented with `unsafe` code, because using a safe abstraction is safe.

Let's talk about each of the four unsafe superpowers in turn, and along the way we'll look at some abstractions that provide a safe interface to unsafe code.

Dereferencing a Raw Pointer

Way back in Chapter 4, in the "Dangling References" section, we covered that the compiler ensures references are always valid. Unsafe Rust has two new types similar to references called *raw pointers*. Just like with references, raw pointers can be immutable or mutable, written as `*const T` and `*mut T`, respectively. The asterisk isn't the dereference operator; it's part of the type name. In the context of raw pointers, "immutable" means that the pointer can't be directly assigned to after being dereferenced.

Different from references and smart pointers, keep in mind that raw pointers:

- Are allowed to ignore the borrowing rules and have both immutable and mutable pointers, or multiple mutable pointers to the same location
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic clean-up

By opting out of having Rust enforce these guarantees, you are able to make the

tradeoff of giving up guaranteed safety to gain performance or the ability to interface with another language or hardware where Rust's guarantees don't apply.

Listing 19-1 shows how to create both an immutable and a mutable raw pointer from references.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: Creating raw pointers from references

Notice we don't include the `unsafe` keyword here---you can *create* raw pointers in safe code, you just can't *dereference* raw pointers outside of an unsafe block, as we'll see in a bit.

We've created raw pointers by using `as` to cast an immutable and a mutable reference into their corresponding raw pointer types. Because we created them directly from references that are guaranteed to be valid, we can know that these particular raw pointers are valid, but we can't make that assumption about just any raw pointer.

Next we'll create a raw pointer whose validity we can't be so certain of. Listing 19-2 shows how to create a raw pointer to an arbitrary location in memory. Trying to use arbitrary memory is undefined: there may be data at that address or there may not, the compiler might optimize the code so that there is no memory access, or your program might segfault. There's not usually a good reason to be writing code like this, but it is possible:

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: Creating a raw pointer to an arbitrary memory address

Remember that we said you can create raw pointers in safe code, but you can't *dereference* raw pointers and read the data being pointed to. We'll do so now using the dereference operator, `*`, on a raw pointer, which does require an `unsafe` block, as shown in Listing 19-3:

```
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);
}
```

Listing 19-3: Dereferencing raw pointers within an `unsafe` block

Creating a pointer can't do any harm; it's only when accessing the value that it points at that you might end up dealing with an invalid value.

Note also that in Listing 19-1 and 19-3 we created `*const i32` and `*mut i32` raw pointers that both pointed to the same memory location, that of `num`. If instead we'd tried to create an immutable and a mutable reference to `num`, this would not have compiled because Rust's ownership rules don't allow a mutable reference at the same time as any immutable references. With raw pointers, we are able to create a mutable pointer and an immutable pointer to the same location, and change data through the mutable pointer, potentially creating a data race. Be careful!

With all of these dangers, why would we ever use raw pointers? One major use case is when interfacing with C code, as we'll see in the next section on unsafe functions. Another case is when building up safe abstractions that the borrow checker doesn't understand. Let's introduce unsafe functions then look at an example of a safe abstraction that uses unsafe code.

Calling an Unsafe Function or Method

The second type of operation that requires an unsafe block is calls to unsafe functions. Unsafe functions and methods look exactly like regular functions and methods, but they have an extra `unsafe` out front. That `unsafe` indicates the function has requirements we as programmers need to uphold when we call this function, because Rust can't guarantee we've met these requirements. By calling an unsafe function within an `unsafe` block, we are saying that we've read this function's documentations and take responsibility for upholding the function's contracts ourselves.

Here's an unsafe function named `dangerous` that doesn't do anything in its body:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

We must call the `dangerous` function within a separate `unsafe` block. If we try to call `dangerous` without the `unsafe` block, we'll get an error:

```
error[E0133]: call to unsafe function requires unsafe function or block
 -->
  |
4 |     dangerous();
  |     ^^^^^^^^^^^ call to unsafe function
```

By inserting the `unsafe` block around our call to `dangerous`, we're asserting to Rust that we've read the documentation for this function, we understand how to use it properly, and we've verified that everything is correct.

Bodies of unsafe functions are effectively `unsafe` blocks, so to perform other unsafe operations within an unsafe function, we don't need to add another `unsafe` block.

Creating a Safe Abstraction Over Unsafe Code

Just because a function contains unsafe code doesn't mean the whole function needs to be marked as unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction. As an example, let's check out a function from the standard library, `split_at_mut`, that requires some unsafe code and explore how we might implement it. This safe method is defined on mutable slices: it takes one slice and makes it into two by splitting the slice at the index given as an argument. Using `split_at_mut` is demonstrated in Listing 19-4:

```
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]);
```

Listing 19-4: Using the safe `split_at_mut` function

This function can't be implemented using only safe Rust. An attempt might look something like Listing 19-5, which will not compile. For simplicity, we're implementing `split_at_mut` as a function rather than a method, and only for slices of `i32` values rather than for a generic type `T`.

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

Listing 19-5: An attempted implementation of `split_at_mut` using only safe Rust

This function first gets the total length of the slice, then asserts that the index given as a parameter is within the slice by checking that it's less than or equal to the length. The assertion means that if we pass an index that's greater than the index to split the slice at, the function will panic before it attempts to use that index.

Then we return two mutable slices in a tuple: one from the start of the original slice to the `mid` index, and another from `mid` to the end of the slice.

If we try to compile this, we'll get an error:

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
 -->
  |
6 |     (&mut slice[..mid],
  |           ----- first mutable borrow occurs here
7 |      &mut slice[mid..])
  |           ^^^^^ second mutable borrow occurs here
8 | }
  | - first borrow ends here
```

Rust's borrow checker can't understand that we're borrowing different parts of the slice; it only knows that we're borrowing from the same slice twice. Borrowing different parts of a slice is fundamentally okay because our two slices aren't overlapping, but Rust isn't smart enough to know this. When we know something is okay, but Rust doesn't, it's time to reach for unsafe code.

Listing 19-6 shows how to use an `unsafe` block, a raw pointer, and some calls to

unsafe functions to make the implementation of `split_at_mut` work:

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut
[i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Listing 19-6: Using unsafe code in the implementation of the `split_at_mut` function

Recall from the “Slices” section in Chapter 4 that slices are a pointer to some data and the length of the slice. We use the `len` method to get the length of a slice, and the `as_mut_ptr` method to access the raw pointer of a slice. In this case, because we have a mutable slice to `i32` values, `as_mut_ptr` returns a raw pointer with the type `*mut i32`, which we’ve stored in the variable `ptr`.

We keep the assertion that the `mid` index is within the slice. Then we get to the unsafe code: the `slice::from_raw_parts_mut` function takes a raw pointer and a length and creates a slice. We use this function to create a slice that starts from `ptr` and is `mid` items long. Then we call the `offset` method on `ptr` with `mid` as an argument to get a raw pointer that starts at `mid`, and we create a slice using that pointer and the remaining number of items after `mid` as the length.

The function `slice::from_raw_parts_mut` is unsafe because it takes a raw pointer and must trust that this pointer is valid. The `offset` method on raw pointers is also unsafe, because it must trust that the offset location is also a valid pointer. We therefore had to put an `unsafe` block around our calls to `slice::from_raw_parts_mut` and `offset` to be allowed to call them. We can tell, by looking at the code and by adding the assertion that `mid` must be less than or equal to `len`, that all the raw pointers used within the `unsafe` block will be valid pointers to data within the slice. This is an acceptable and appropriate use of `unsafe`.

Note that we don't need to mark the resulting `split_at_mut` function as `unsafe`, and we can call this function from safe Rust. We've created a safe abstraction to the unsafe code with an implementation of the function that uses `unsafe` code in a safe way because it creates only valid pointers from the data this function has access to.

In contrast, the use of `slice::from_raw_parts_mut` in Listing 19-7 would likely crash when the slice is used. This code takes an arbitrary memory location and creates a slice ten thousand items long:

```
use std::slice;

let address = 0x012345usize;
let r = address as *mut i32;

let slice = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Listing 19-7: Creating a slice from an arbitrary memory location

We don't own the memory at this arbitrary location, and there's no guarantee that the slice this code creates contains valid `i32` values. Attempting to use `slice` as if it was a valid slice would result in undefined behavior.

Using `extern` Functions to Call External Code

Sometimes, your Rust code may need to interact with code written in another language. For this, Rust has a keyword, `extern`, that facilitates the creation and use of a *Foreign Function Interface* (FFI). A Foreign Function Interface is a way for a programming language to define functions and enable a different (foreign) programming language to call those functions.

Listing 19-8 demonstrates how to set up an integration with the `abs` function from the C standard library. Functions declared within `extern` blocks are always unsafe to call from Rust code, because other languages don`t enforce Rust's rules and guarantees and Rust can't check them, so responsibility falls on the programmer to ensure safety:

Filename: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Listing 19-8: Declaring and calling an `extern` function defined in another language

Within the `extern "C"` block, we list the names and signatures of external functions from another language we want to be able to call. The `"C"` part defines which *application binary interface* (ABI) the external function uses---the ABI defines how to call the function at the assembly level. The `"C"` ABI is the most common, and follows the C programming language's ABI.

Calling Rust Functions from Other Languages

You can also use `extern` to create an interface that allows other languages to call Rust functions. Instead of an `extern` block, we add the `extern` keyword and specify the ABI to use just before the `fn` keyword. We also need to add a `#[no_mangle]` annotation to tell the Rust compiler not to mangle the name of this function. Mangling is when a compiler changes the name we've given a function to a different name that contains more information for other parts of the compilation process to consume but is less human readable. Every programming language compiler mangles names slightly differently, so for a Rust function to be nameable from other languages, we have to disable the Rust compiler's name mangling.

In this example we make the `call_from_c` function accessible from C code, once it's compiled to a shared library and linked from C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

This usage of `extern` does not require `unsafe`.

Accessing or Modifying a Mutable Static Variable

We've managed to go this entire book without talking about *global variables*, which Rust does support, but which can be problematic with Rust's ownership rules. If you have two threads accessing the same mutable global variable, it can cause a data race.

Global variables are called *static* variables in Rust. Listing 19-9 shows an example declaration and use of a static variable with a string slice as a value:

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Listing 19-9: Defining and using an immutable static variable

`static` variables are similar to constants, which we discussed in the "Differences Between Variables and Constants" section in Chapter 3. The names of static variables are in `SCREAMING_SNAKE_CASE` by convention, and we *must* annotate the variable's type, which is `&'static str` in this case. Static variables may only store references with the `'static` lifetime, which means the Rust compiler can figure out the lifetime by itself and we don't need to annotate it explicitly. Accessing an immutable static variable is safe.

Constants and immutable static variables may seem similar, but a subtle difference is that values in a static variable have a fixed address in memory. Using the value will always access the same data. Constants, on the other hand, are allowed to duplicate their data whenever they are used.

Another difference between constants and static variables is that static variables can be mutable. Both accessing and modifying mutable static variables is *unsafe*. Listing 19-10 shows how to declare, access, and modify a mutable static variable named `COUNTER`:

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

Listing 19-10: Reading from or writing to a mutable static variable is unsafe

Just like with regular variables, we specify mutability using the `mut` keyword. Any code that reads or writes from `COUNTER` must be within an `unsafe` block. This code compiles and prints `COUNTER: 3` as we would expect because it's single threaded. Having multiple threads access `COUNTER` would likely result in data races.

With mutable data that's globally accessible, it's difficult to ensure there are no data races, which is why Rust considers mutable static variables to be unsafe. Where possible, it's preferable to use the concurrency techniques and thread-safe smart pointers we discussed in Chapter 16, so the compiler checks that data accessed from different threads is done safely.

Implementing an Unsafe Trait

Finally, the last action that only works with `unsafe` is implementing an unsafe trait. A trait is unsafe when at least one of its methods has some invariant that the compiler can't verify. We can declare that a trait is `unsafe` by adding the `unsafe` keyword before `trait`, and then implementation of the trait must be marked as `unsafe` too, as shown in Listing 19-11:

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}
```

Listing 19-11: Defining and implementing an unsafe trait

By using `unsafe impl`, we're promising that we'll uphold the invariants that the compiler can't verify.

As an example, recall the `Sync` and `Send` marker traits from the "Extensible Concurrency with the `Sync` and `Send` Traits" section of Chapter 16, and that the compiler implements these automatically if our types are composed entirely of `Send` and `Sync` types. If we implement a type that contains something that's not `Send` or `Sync`, such as raw pointers, and we want to mark that type as `Send` or `Sync`, we must use `unsafe`. Rust can't verify that our type upholds the guarantees that it can be safely sent across threads or accessed from multiple threads, so we need to do those checks ourselves and indicate as such with `unsafe`.

When to Use Unsafe Code

Using `unsafe` to take one of these four actions isn't wrong or even frowned upon, but it is trickier to get `unsafe` code correct because the compiler isn't able to help uphold memory safety. When you have a reason to use `unsafe` code, it is possible to do so, and having the explicit `unsafe` annotation makes it easier to track down the source of problems if they occur.