Unsafe for work

November 22, 2024

Rust

Familiar, yet unknown.

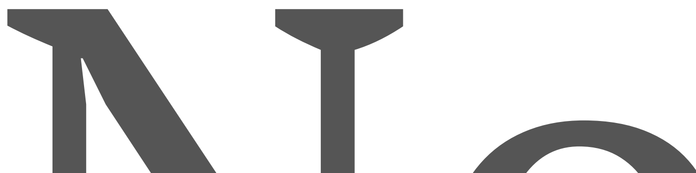
Before we start, have a look at this image. It is from a German book from the early Nineties, and it features some very interesting cover artwork. A knight with a cape and a big bushy mustache rides a friendly green dragon. A castle, a dolphin, and some animals are in the background. Focus on the details; we will return to that image at the end.

What's the deal with `unsafe`?

So, what's the deal with unsafe? It seems like no other feature in Rust has caused more confusion and oh my opinions like unsafe Rust. Every now and then, when you look at online forums or social media, you find people getting riled up about unsafe code. One of my favorites is Should we be worried about the proliferation of unsafe in Rust code on Reddit. This post is fantastic for numerous things. It counts how often the keyword `unsafe` appears in Rust libraries, including the standard library. And I love the word "proliferation". I had to check what proliferation means. It means uncontrolled and rapid growth. In German, we use Proliferation to describe "access and distribution of weapons of mass destruction." Well, that definitely sounds unsafe!

But there's more. People asking if Rust is something like a modernized C or C++ because of `unsafe`. Others, giving keynotes on Rust at conferences, even say that `unsafe` Rust deactivates the borrow checker completely! And I'm pretty sure that lots of people really believe this!

So, before we go into details, let me answer the question: Does `unsafe` Rust deactivate the borrow checker? The answer is a resounding



NO.

`unsafe` Rust does not deactivate the borrow checker. Let’s try it out. This function is called `split` over a generic slice `T`. It takes a point and returns two slices, one from the beginning to the point and one from the point to the end.

```
fn split<T>(slice: &[T], point: usize) -> (&[T], &[T]) {
    (&slice[..point], &slice[point..])
}
```

This function works as intended with all ownership and borrowing rules in place. But what if we want to split a mutable slice? Let’s create a function `split_mut` that takes a mutable slice over `T` and a point and returns two mutable slices, one from the beginning to the point and one from the point to the end.

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut
    (&mut slice[..point], &mut slice[point..])
    //                    ^~~~~~~~~~~~~~~~~~ERROR
}
```

As you would expect, the ownership and borrowing rules won’t allow this function to be compiled.

```
5 | fn split_mut(slice: &mut [T], point: usize) -> (&mut [T], &mu
  |                          let's call the lifetime of this refe
6 |     (&mut slice[..point], &mut slice[point..])
  |     ---------------------------^^^^^----------
  |     |     |                    |
  |     |     |                    second mutable borrow occurs he
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*slice` is borrowed fo
```

We have two mutable borrows, and this is not allowed. So, if **unsafe** really would deactivate the borrow checker, we could just wrap the nasty pieces in **unsafe** and be done with it, right?

```
fn split_mut<T>(slice: &mut [T], point: usize)  > (&mut [T], &mut
    unsafe { (&mut slice[..point], &mut slice[point..]) }
}
```

It turns out that this function still won't be compiled. For the same reasons.

```
5 | fn split_mut(slice: &mut [T], point: usize) -> (&mut [T], &mu
  |                          let's call the lifetime of this refe
6 |     unsafe { (&mut slice[..point], &mut slice[point..]) }
  |               ---------------------------^^^^^----------
  |               |     |                    |
  |               |     |                    second mutable borro
  |               |     first mutable borrow occurs here
  |               returning this value requires that `*slice` is b
```

So, **unsafe** Rust does not deactivate the borrow checker. But what does **unsafe** Rust do? Let's find out.

For some reason, the image from the beginning comes to mind again.

A language superset

If `unsafe` does not unplug the safety belts, what does it do? The best way `unsafe` Rust can be described is as a language superset. It adds features to a language that allow you to do more than you could do before, like:

- Dereferencing raw pointers.
- Calling `unsafe` functions and methods, including `extern`.
- Implementing `unsafe` traits.
- Mutating static variables.
- Access unions.

If you ask me, the first two points are the most common occurrences of `unsafe`, and I'll focus the rest of this article on them. If you interface with C, you might occasionally use unions in Rust, but I have never seen it in the wild.

Raw pointers

In Rust, you have different pointer types. You know at least three of them: The owned type, a shared reference, and a mutable reference.

Type	Explanation
`T`	Owned.
`&T`	A shared reference.
`&mut T`	A mutable reference.

In Rust, we like to call shared and mutable references also shared and mutable *borrows*. The Rust borrow checker works on those types. They're borrows, and they're being checked by the borrow checker.

But there are also two other types of pointers in Rust. They're called *raw pointers*, and they exist as an immutable `const` and a mutable `mut`

variant.

Type	Explanation
`*const T`	Raw pointer.
`*mut T`	A mutable raw pointer.

Those pointers are unchecked by the borrow checker. The main difference to borrows is that raw pointers don't have a lifetime attached to it, so the borrow checker can't check if values "live long enough".

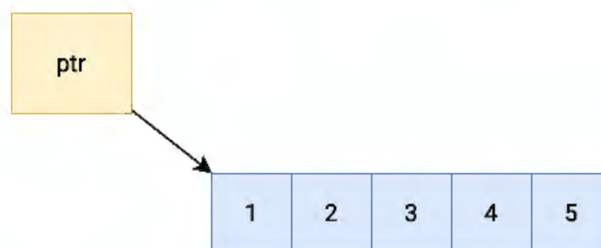
> *Raw pointers don't have a lifetime attached.*

How do you create a raw pointer? There are many ways. The most simple one comes with the most recent Rust 1.80 syntax.

```
let v   [1, 2, 3, 4, 5];
let ptr = &raw const v[0];
```

Here, I create an array `v` and a raw pointer for the first element using `&raw const`. If we hover over `ptr`, we can see the type is `*const i32`. There are other ways of creating raw pointers like `as` casting or using the `.as_ptr()`/`as_mut_ptr()` method, which is implemented for several types.

This pointer is a pointer as you might expect.



It points to an address that might or might not contain a value, and when we print it out using the `Debug` trait, we get a hexadecimal address.

```
let v = [1, 2, 3, 4, 5];
let ptr   &raw const v[0];
println!("{:?}", ptr);
// prints e.g. `0x16f0262a4`
```

So far, so good. We are not interested in the address; we want to access the value behind the pointer. We can do so by dereferencing a pointer. This looks an awful lot like C. However, when we dereference the pointer as it is, we get an error!

```
let mut v   [1, 2, 3, 4, 5];
let ptr = &raw mut v[0];

*ptr   9;
//^~~~~~~ERROR
println!("{:?}", *ptr);
//               ^~~~ERROR
```

Rust will warn us with a descriptive error message that tells us exactly what’s going wrong:

```
error[E0133]: dereference of raw pointer is unsafe and requires
unsafe function or block
    > src/main.rs:33:5
   |
33 |     *ptr = 9;
   |     ^^^^ dereference of raw pointer
   |
   = note: raw pointers may be null, dangling or unaligned;
   they can violate aliasing rules and cause data races:
   all of these are undefined behavior
```

This is where **unsafe** comes into play. If we want to dereference a raw pointer, we have to wrap the operation in an **unsafe** block.

```
let mut v   [1, 2, 3, 4, 5];
let ptr   &raw mut v[0];
```

```
unsafe { *ptr = 9 };
println!("{:?}", unsafe { *ptr });
```

This program compiles and does exactly what we expect it to do. **`unsafe`** is our way of telling the compiler that what's going on here is checked, safe, and okay to do. We're responsible for the safety of the operation. We can mess it up, however. Let's look at a dangling pointer.

```
fn danging_pointer() {
    let ptr: *const i32;

    {
        let value = 42;
        ptr   &raw const value;
    }

    println!("Dereferencing ptr: {}", unsafe { *ptr });
}
```

did you notice that I had a little Freudian slip in the function name? This was totally by accident, but I like it. In this example, we create a pointer outside a block. Inside the block, we create a value and assign the address of the value to the pointer. After the block, the value goes out of scope and will be freed due to the ownership and borrowing rules. The pointer is left dangling. When we try to dereference the pointer, we get a **42** in debug mode and **24565** in release mode. That we got a value in any of these modes is just pure luck. The pointer is dangling, and dereferencing it is *undefined behavior.*

`unsafe` blocks and functions

Let's stick a bit with **`unsafe`** functions and blocks.

```
unsafe fn something_unchecked() {
    // ...
```

```
}
```

An **`unsafe`** *function* highlights that the code inside the function might cause undefined behavior. This does not necessarily mean that you dereference raw pointers or call **`unsafe`** functions. It merely states that if you call this, you need to make sure that some safety precautions are met. You need to check this behavior before you can call the function. We also call this checking if all safety *invariants* are met.

```
unsafe {
    something_unchecked();
}
```

`unsafe` blocks, on the other hand, are a way of telling the compiler that you've checked all the safety invariants. This is *opt-in*, and you know what you're dealing with when executing **`unsafe`** code inside an **`unsafe`** block.

Those are the contracts you have to fulfill as a developer. Accessing unions, mutating global statics, calling **`unsafe`** functions, and dereferencing raw pointers are all things that can cause undefined behavior. This is the contract you are being presented with. Your use of **`unsafe {}`** tells the compiler that you fulfill your side of the contract.

> ***`unsafe`*** *moves safety checks from the compiler to the developer.*

As with everything in Rust, it makes those decisions *explicit*. For example; You know how Rust's error handling works and that it's here to make impossible states impossible. But I bet when you have been writing Rust, you've used **`unwrap()`** at some point in time, which can cause a panic if you hit one of those "impossible" states. But the difference to other programming languages is that you made the decision by calling **`unwrap()`**. You have made it explicit, and if you hit a panic at one point in time, you know exactly where to look.

The same goes for `unsafe`. `unsafe` blocks are used sparingly and just for a few lines of code. When something happens that violates the safety invariants, you know exactly where to look.

This is fundamentally different from how C or C++ handles unsafe code. Even with all the safety abstractions that exist, it is possible to throw in pointer arithmetics anywhere.

Safe abstractions

We must accept that `unsafe` can't be avoided. There are situations that call for raw pointers; the Rust standard library and popular crates are full of them. Sometimes, this is how certain data structures or   well computers work. But what we can do is to avoid working with `unsafe` code entirely by using safe abstractions.

Here I want to take a look at Bevy, the gaming engine you all know and love. This is just a small example, and if you want to know more, I highly recommend watching Boxy Uwu's talk at Euro Rust 2024.

Bevy has a lot of self made data structures that are optimized for performance. One of those is `ThinArrayPtr`, which is similar to a `Vec<T>` but with capacity and length cut out for performance reasons. It means that you allocate enough memory upfront and work with it. There are no reallocations, just memory.

```
pub struct ThinArrayPtr<T> {
    data: NonNull<T>,
    #[cfg(debug_assertions)]
    capacity: usize,
}
```

I will just focus on the release mode code and leave out the debug assertions. `ThinArrayPtr` uses a `NonNull<T>` pointer to its data. `NonNull<T>` in itself is a safe abstraction of a raw pointer. The

documentation says that *This is often the correct thing to use when building data structures using raw pointers*, so it's exactly what we need.

The **`impl`** block of **`ThinArrayPtr`** starts with a *private* **`empty()`** method that creates a new struct instance with a dangling pointer.

```
impl<T> ThinArrayPtr<T> {
    fn empty()  > Self {
        Self {
            data: NonNull::dangling(),
        }
    }
    //...
}
```

The dangling method itself is a function that calls **`unsafe`** code.

```
pub const fn dangling() -> Self {
    // SAFETY: mem::align_of() returns a non zero
    // usize which is then casted
    // to a *mut T. Therefore, `ptr` is not null
    // and the conditions for
    // calling new_unchecked() are respected.
    unsafe {
        let ptr = crate::ptr::dangling_mut::<T>();
        NonNull::new_unchecked(ptr)
    }
}
```

It creates a dangling, mutable pointer of the size of **`T`** and then calls the **`unsafe`** function **`NonNull::new_unchecked()`**. This function is a safe abstraction of creating a **`NonNull<T>`** from a raw pointer. The documentation says the pointer is dangling, but well aligned, so the pointer's adddress is dividable by the number of bytes that you need to store **`T`**.

Calling **`NonNull::new_unchecked`** requires you – per its documentation – to pass a pointer that is not null. This is the safety invariant you need to check before calling this function. Since we just created a non-null

pointer the line before, this safety condition is met, and we can safely call this function.

That's exactly how you should treat unsafe code. There's a potential safety hazard, but you check the invariant, put everything in an unsafe block, and document what you have checked! Beautiful!

Let's look at the next function in the **`impl`** block, **`with_capacity()`**. This function creates a new instance of **`ThinArrayPtr`** with a given capacity. It is a public function, so it is the interface to you, the user of this library.

```
pub fn with_capacity(capacity: usize)  > Self {
    let mut arr   Self::empty();
    if capacity > 0 {
        // SAFETY:
        //   The `current_capacity` is 0 because it was just crea
        unsafe { arr.alloc(NonZeroUsize::new_unchecked(capacity))
    }
    arr
}
```

The function **`with_capacity`** creates a new empty instance of **`ThinArrayPtr`** and then checks if the capacity is greater than zero. If it is, it will allocate new memory for the data. The capacity is passed as a **`NonZeroUsize`**, calling **`new_unchecked`** is an **`unsafe`** operation. The documentation of **`NonZeroUsize::new_unchecked`** states that *The value must not be zero.* Which is, well, what we checked in the if statement before.

As you can see, we check the safety invariant just a line before actually calling the **`unsafe`** function. The **`unsafe`** function has just one little condition we need to meet to call it safely, which can be checked easily.

You see this a lot when writing **`unsafe`** code in Rust. Instead of having one big **`unsafe`** block and throwing pointers around, you work with very narrow, tiny checks on safety invariants.

This code from Bevy, even if you've never worked with the library or checked its code, tells you exactly where `unsafe` behavior might happen and how to avoid it.

And this makes the use of `unsafe` … safe.

I'm pretty sure you used some safe abstractions in your Rust code. You might have used `Vec<T>`, `Rc<T>`, `Arc<T>`, `Mutex<T>`, or maybe `String`? Yes, even `String` is just a vector over characters under the hood, and since `Vec<T>` is a safe abstraction over a raw pointer, `String` also contains unsafe code deep down in the call graph.

And this is fine. This still doesn't mean that Rust is a memory-unsafe language.

Do you know memory-safe languages like Python or JavaScript? They do the same. You have arrays and objects as a safe abstraction in their language, but underneath runs a VM written in C and C++ that interprets this safe abstraction and allocates the necessary memory. And I figure the VM developers checked all the invariants that could happen.

The only difference is that the `unsafe` parts are much easier to access because they happen in the same programming language. But I'd argue that makes it even better. You might think that there's a proliferation of `unsafe` in Rust, whereas it's quite astonishing that there is, in fact, so little of `unsafe` code.

Unsafe for work

This article is called "Unsafe for Work," and I must admit that the name alone motivated me to write it. But what does all of this mean for you when writing Rust applications?

First of all: Don't worry. You're fine, really. You'll probably work entirely with safe abstractions and won't see a single `unsafe` block in your code.

But when we look at what Rust allows that is **`unsafe`**, we can see some use cases that might arise.

- Unsafe Rust allows you to dereference raw pointers. Like when accessing a register on your embedded device.
- Unsafe Rust allows calling **`unsafe`** functions and methods, including **`extern`**, which you need for FFI.
- Unsafe Rust allows you to access unions, which are also sometimes necessary when you want to interface with C.
- Or you need all of them, implement unsafe traits, and mutate static variables when you implement your own performance-optimized data structures.

So, hardware access, FFI, and performance optimizations. Those are just a number of use cases, but frankly, also those use cases that people currently want to use Rust for. They want to work with embedded devices. They want to ditch C++ for obvious reasons but must carry along years of legacy code. Or they have a performance-critical application and want to use Rust for it.

But let's look at a real-world project. I work on a JavaScript runtime that is written in Rust. We built on Google's V8 engine, which is written in C++; thus, we need to interface with C++ code a lot.

The Rust part of the project has 50,000 lines of code, and we have a single occurrence of **`unsafe`**.

```
// SAFETY: Assumes that V8 passes a valid string pointer
let details_c_str = unsafe { CStr::from_ptr(details.detail) };
details_c_str.to_string_lossy()
```

We have so few because we build on Deno's excellent **`rusty_v8`** crate, which is a safe abstraction over V8's C++ API. The **`unsafe`** block is there because when V8 crashes, it produces an error message we want to send over to monitoring. We serialized it from a pointer we got and checked beforehand that this pointer holds data.

Using safe abstractions is key. There's a wonderful comment in another Reddit post about the confusion of `unsafe` in Rust:

> *I think people think you just write all low level rust code in an unsafe block. I am working on a microcontroller project in Rust for work, and I am not yet using unsafe for anything, even when accessing memory directly (through the esp_idf_svc crate).*

And this is key! We need (first-party) safe abstractions.

Unsafe guidelines

When writing `unsafe`, you should follow some guidelines. It's the obvious stuff:

- Limit your `unsafe` code. You've seen that `unsafe` blocks only span a few lines of code. It should be possible to grasp the safety invariants you've checked and the `unsafe` operation following.
- `unsafe` keywords are part of your API design. If you write functions that might cause undefined behavior somewhere, mark them as such. Keep those functions small and easy to understand.
- Document your `unsafe` code. Use a `# Safety` headline in your comment to explain the safety precautions users must take when using your `unsafe` code. When using `unsafe` blocks, comment them with `// SAFETY` and the checks you've made.

In short: Everything you've seen from the Bevy example above.

Miri

When writing `unsafe` code, it's also a good idea to check your code with Miri. Miri is a nightly module that checks Rust's mid level intermediate

representation (MIR). It can detect some kinds of undefined behavior.

Let’s go to the *danging pointer* example from above.

```
fn danging_pointer() {
    let ptr: *const i32;

    {
        let value   42;
        ptr = &raw const value;
    }

    unsafe {
        println!("Dereferencing ptr: {}", *ptr);
    }
}
```

If you run this code with Miri, you get the following output:

```
$ cargo +nightly miri run

error: Undefined Behavior: out-of-bounds pointer use:
alloc1905 has been freed, so this pointer is dangling
    > src/main.rs:15:43
   |
15 |  println!("Dereferencing ptr: {}", *ptr);
   |                                    ^^^^
   |       out-of-bounds pointer use: alloc1905
   |       has been freed, so this pointer is
   |       dangling
   |
   = help: this indicates a bug in the program: it performed an
   invalid operation, and caused Undefined Behavior
```

Miri will tell me exactly what’s wrong: This pointer is dangling, and this indicates a bug in the program that causes undefined behavior.

Here’s a **`split_mut`** function that works by using **`unsafe`** code:

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut
    let ptr   slice.as_mut_ptr();
    let len   slice.len();

    unsafe {
        (
            from_raw_parts_mut(ptr, point),
            from_raw_parts_mut(ptr.add(point), len - point),
        )
    }
}
```

If I check it with Miri, it will compile. Thank you, Miri

Let’s cause some havoc and move the index by one element so that the last element from the first slice is the first element from the last slice. It’s fun, but it can also cause undefined behavior.

```
fn split_mut<T>(slice: &mut [T], point: usize) -> (&mut [T], &mut
    let ptr = slice.as_mut_ptr();
    let len   slice.len();

    unsafe {
        (
            from_raw_parts_mut(ptr, point),
            from_raw_parts_mut(ptr.add(point - 1), len - point + 
        )
    }
}
```

Miri won’t let this one pass.

```
error: Undefined Behavior: trying to retag from <4464>
for Unique permission at alloc1963[0x4], but that tag does
not exist in the borrow stack for this location
    > src/main.rs:17:9
   |
17 | / (
18 | |  std::slice::from_raw_parts_mut(ptr, point),
19 | |  std::slice::from_raw_parts_mut(ptr.add(point   1), len   
```

```
20 | | )
   | | ^
   | | |
   | | trying to retag from <4464> for Unique
   permission at alloc1963[0x4], but that…
```

Again, this prevents me from introducing code that can cause undefined behavior. Miri won't be able to check everything, but not running it would be a mistake. Get what you can.

Unsafe is a safety feature

I think Florian Gilcher coined the phrase "Unsafe is a safety feature." What you've seen in this article is that **`unsafe`** allows you to go a bit further than regular Rust. It's a language superset that has its own rules, and it gives more responsibility to the developer.

But the way **`unsafe`** has been designed, its usage is limited, targeted, narrow, and explicit. It's a way to highlight potentially problematic pieces of your code. It helps you figure out what's wrong and reduces the complexity to a few lines of code. With that, I'm 100% convinced that **`unsafe`** is a safety feature.

Oh, right. The image from the beginning. Have you figured it out? Maybe it will become clearer when I put it next to its Japanese original and remove the black bars.

The shocking revelation.

It's the cover of a German strategy guide for Super Mario World. Honestly, I think it's beautiful! Not even GenAI can mess it up that badly, and I'm all here for it. For the nerds: There is a dolphin in Super Mario World that you can ride. The albatross appears in post-game, replacing bullet bill. I have yet to figure out what the hamster in the lifebuoy is supposed to be.

The point is that the artist clearly had a different view of the game than most of us did. They heard about the stories and maybe saw some images from a game. Not knowing that Mario is a plumber but knowing that Mario must save a princess led them to a depiction of the game like that.

They saw what was familiar to them, missing the point entirely, creating a false, misguiding representation. And I think this is what happens when we talk about `unsafe` Rust. We see asterisks, pointers, and addresses and think that it's just C/C++ all over again, when in reality, it's not.

Related Articles

November 20, 2024 | Rust, Tokio

Tokio: Macros

November 19, 2024 | Rust, Tokio

Tokio: Channels

November 18, 2024 | Rust, Tokio

Tokio: Getting Started

oida.dev © 2012 - 2024