**Learning Rust: Avoiding Panics**

[[Adrian Macal](https://amacal.medium.com/?source=post_page-----111d2c2cd1bd--------------------------------)](https://amacal.medium.com/?source=post_page-----111d2c2cd1bd--------------------------------)

[[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----111d2c2cd1bd--------------------------------)](https://levelup.gitconnected.com/?source=post_page-----111d2c2cd1bd--------------------------------)

[Adrian Macal](https://amacal.medium.com/?source=post_page-----111d2c2cd1bd--------------------------------)

·

Follow

Published in

[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----111d2c2cd1bd--------------------------------)

·

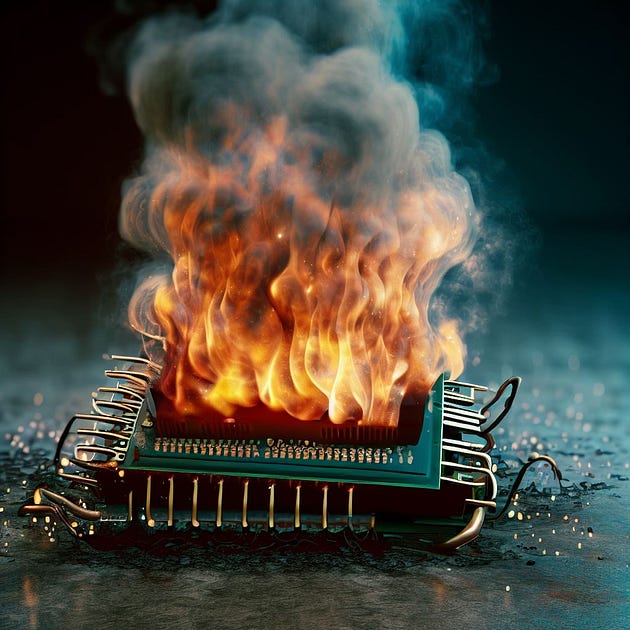
5 min read

·

Sep 18

69

2



Is Rust’s safety net as foolproof as you think or are there instances where the system can still panic?

Rust is well-known primarily for its safety features, largely attributed to its borrowing system. But it’s not just the compiler-enforced borrowing system that contributes to Rust’s safety; explicit error handling also plays a significant role. This is more than just a technical feature — it’s ingrained in the philosophy of Rust’s developer mindset. Rust allows you to specify how different code paths should be managed, making unhandled panics a rare event.

So, why do panics occur at all? Answering that isn’t straightforward. Initially, I thought Rust and its ecosystem were so well-designed that panic paths would be the antithesis of idiomatic Rust code. However, with more experience, I came to understand that panics are sometimes an intended part of the contract between the calling and called functions.

Take this code snippet, taken directly from Rust’s standard library, as an example:

#[cfg(not(test))]  
impl<T> [T] {  
 /// Copies all elements from `src` into `self`, using a memcpy.  
 ///  
 /// The length of `src` must be the same as `self`.  
 ///  
 /// If `T` does not implement `Copy`, use [`clone\_from\_slice`].  
 ///  
 /// # Panics  
 ///  
 /// This function will panic if the two slices have different lengths.  
 /// ...  
 #[doc(alias = "memcpy")]  
 #[stable(feature = "copy\_from\_slice", since = "1.9.0")]  
 #[track\_caller]  
 pub fn copy\_from\_slice(&mut self, src: &[T])  
 where  
 T: Copy,  
 {  
 // The panic code path was put into a cold function to not bloat the  
 // call site.  
 #[inline(never)]  
 #[cold]  
 #[track\_caller]  
 fn len\_mismatch\_fail(dst\_len: usize, src\_len: usize) -> ! {  
 panic!(  
 "source slice length ({}) does not match destination slice length ({})",  
 src\_len, dst\_len,  
 );  
 }  
  
 if self.len() != src.len() {  
 len\_mismatch\_fail(self.len(), src.len());  
 }  
  
 // SAFETY: `self` is valid for `self.len()` elements by definition, and `src` was  
 // checked to have the same length. The slices cannot overlap because  
 // mutable references are exclusive.  
 unsafe {  
 ptr::copy\_nonoverlapping(src.as\_ptr(), self.as\_mut\_ptr(), self.len());  
 }  
 }  
}

The function’s objective is simple: copy the contents from one slice to another using our old friend memcpy. It doesn't return an Err or Option but does perform an internal length check. If the lengths of both slices aren't equal, it panics. This might seem logical but leaves no room for handling the case when lengths differ.

Imagine calling this method without a prior length check. A panic ensues, causing a catastrophic failure, which should ideally never happen:

pub fn copy\_raw(src: &[u8], dst: &mut [u8]) {  
 dst.copy\_from\_slice(src)  
}

Upon compilation, the generated assembly code strictly follows your instructions — it checks the slice lengths and triggers a panic if they don’t match.

example::copy\_raw:  
 push rax  
 cmp rcx, rsi  
 jne .LBB0\_1  
 mov rax, rdi  
 mov rdi, rdx  
 mov rsi, rax  
 mov rdx, rcx  
 pop rax  
 jmp qword ptr [rip + memcpy@GOTPCREL]  
.LBB0\_1:  
 lea rdx, [rip + .L\_\_unnamed\_1]  
 mov rdi, rcx  
 call qword ptr [rip + core::slice::<impl [T]>::copy\_from\_slice::len\_mismatch\_fail@GOTPCREL]  
 ud2

Now, consider a different scenario where you want to copy only a segment of the slices:

pub fn copy\_sliced(src: &[u8], dst: &mut [u8]) {  
 dst[0..10].copy\_from\_slice(&src[0..10])  
}

As someone reading the code, I notice that while the slice lengths for the specified range are equal (0..10), there’s no inherent guarantee that either slice has a minimum length of 10. This is also what the compiler notices, leading it to add those safety checks:

example::copy\_sliced:  
 push rax  
 cmp rcx, 9  
 jbe .LBB1\_3  
 cmp rsi, 9  
 jbe .LBB1\_4  
 movzx eax, word ptr [rdi + 8]  
 mov word ptr [rdx + 8], ax  
 mov rax, qword ptr [rdi]  
 mov qword ptr [rdx], rax  
 pop rax  
 ret  
.LBB1\_3:  
 lea rdx, [rip + .L\_\_unnamed\_2]  
 mov edi, 10  
 mov rsi, rcx  
 call qword ptr [rip + core::slice::index::slice\_end\_index\_len\_fail@GOTPCREL]  
 ud2  
.LBB1\_4:  
 lea rdx, [rip + .L\_\_unnamed\_3]  
 mov edi, 10  
 call qword ptr [rip + core::slice::index::slice\_end\_index\_len\_fail@GOTPCREL]  
 ud2

The compiler took the exact precautions that I, as a developer, would take if I were programming defensively, assuming the worst-case execution path. Notably, the code is now designed to handle two potential failure points — both related to slice addressing being out of range. Interestingly, the original check for memcpy length has been completely sidestepped.

Suppose I specify to the compiler that both arrays have a fixed size — 16 bytes for src and 20 bytes for dst. In that case, I could write the function like this:

pub fn copy\_fixed(src: &[u8; 16], dst: &mut [u8; 20]) {  
 dst[0..10].copy\_from\_slice(&src[0..10])  
}

For anyone reading the code, it’s transparent that both arrays can safely accommodate the 0..10 range. But can the compiler make the same deduction? Examining the compiled assembly code:

example::copy\_fixed:  
 movzx eax, word ptr [rdi + 8]  
 mov word ptr [rsi + 8], ax  
 mov rax, qword ptr [rdi]  
 mov qword ptr [rsi], rax  
 ret

The answer is a resounding ‘Yes.’ The compiler picks up on the fixed array sizes, optimizing out any safety checks that would otherwise bog down the runtime.

Returning to the original problem, what if we introduced an explicit length check to our function? We can do this by returning an Option<usize> instead of allowing a panic to occur. Here's how it might look:

pub fn copy\_safe(src: &[u8], dst: &mut [u8]) -> Option<usize> {  
 if src.len() == dst.len() {  
 dst.copy\_from\_slice(src);  
 None  
 } else {  
 Some(src.len() - dst.len())  
 }  
}

This change not only prevents the runtime from panicking but also lets us quantify the length mismatch with a precise numerical value. From an assembly standpoint, there’s no panic-related code, demonstrating that our approach is indeed safer and more robust.

example::copy\_safe:  
 mov r8, rdx  
 mov rax, rsi  
 mov rdx, rsi  
 sub rdx, rcx  
 jne .LBB3\_1  
 push rax  
 mov rsi, rdi  
 mov rdi, r8  
 mov rdx, rax  
 call qword ptr [rip + memcpy@GOTPCREL]  
 xor eax, eax  
 add rsp, 8  
 ret  
.LBB3\_1:  
 mov eax, 1  
 ret

We can pat ourselves on the back; this is undeniably a leap in the right direction without sacrificing the performance.

To panic or not to panic? The question is challenging to answer, but there are a few key points to consider:

* Defensive programming is implicitly applied by the Rust compiler; if you don’t check it, Rust will do it
* If we add an additional defensive layer ourselves, the compiler might eliminate its own defensive code
* If we invoke code that could potentially panic, but no actual execution path leads to it, the unused panic path may be completely eliminated