No-Panic Rust: A Nice Technique for Systems Programming

February 3, 2025

Can Rust replace C? This is a question that has been on my mind for many years, as I created and now am tech lead for upb, a C library for Protocol Buffers. There is an understandable push to bring memory safety to all parts of the software stack, and this would suggest a port of upb to Rust.

While I love the premise of Rust, I have long been skeptical that a port of upb to Rust could preserve the performance and code size characteristics that I and others have fought so hard to optimize. In fact, this blog entry was originally going to be an argument for why Rust cannot match C for upb's use case.

But I recently discovered a technique that shifted my thinking a lot. I call it "No-Panic Rust", and while the technique is clearly not new[1], I was not able to find any in-depth discussion of how it works or what problems it solves. This article is my attempt to fill that gap.

I believe that No-Panic Rust is the key to making Rust a compelling option for low-level systems programming. I now am optimistic about the possibility of porting upb to Rust.

What are Panics?

Panics are Rust's mechanism for *unrecoverable errors*. Anytime our program encounters an error, we have three basic options for how to handle it:

1. Handle the error immediately (eg. retry the operation or fall back to plan B).
2. Propagate the error to the caller, who can decide how to handle it.
3. Immediately abort execution.

In Rust, we use `Result` for (2) and `panic!()` for (3). When we use `Result`, it is considered a "recoverable error", because the caller can test for the error and decide how to respond.

With recoverable errors, the potential for error is reflected in the function signature; a function that returns `Result` is fallible from the perspective of the caller. Panics on the other hand present the illusion of infallibility from an API perspective, but then proceed to handle errors by simply aborting.

There is a lot of standard guidance for when to use `panic!()` vs `Result` (for example, here and here), which largely boils down to the idea that panics should only be used for bugs in the code. I especially like the framing given in this Reddit post:

> [If] your **library** is the source of a panic, then one of the following should be true:
>
> - Your library has a bug.
> - Your library documents a precondition of a public API item that, when not met, causes a panic. Therefore, the user of your library has misused your library, and their code has a bug.
>
> If your Rust **application** panics in response to any user input, then the following should be true: your application has a bug, whether it be in a library or in the primary application code.

In this article we are focused on the library case.

Why Are Panics Bad For Systems Libraries?

If we are trying to port a C library to Rust, we really do not want to introduce panics in the code, even for unusual error conditions. They cause many practical problems:

1. **Code Size:** The runtime to handle a panic pulls in about 300Kb of code.[2] We pay this cost if even a single `panic!()` is reachable in the code. From a code size perspective, this is a severe overhead, given that the upb core is only 30Kb.
2. **Unrecoverable exit:** If a panic is triggered, it takes down the entire process.[3] In many applications, this is a severe failure mode that libraries should never invoke. Instead, we should return all errors to the caller using status codes.
3. **Runtime overhead:** A potential panic implies some kind of runtime check. In many cases, the cost of this check will be minimal, but for very small and frequently invoked operations, the cost of this check could be significant.

In the case of upb, I was concerned about all three of these factors. Ideally we could port upb to Rust without users even noticing. To do that, we want to maintain the same performance, code size footprint, and error reporting behavior that the C code has now. Panics get in the way of this ideal.

At some point I realized that it might be possible to ban panics from the library entirely, which would solve all of these problems at once. That is when I started getting much more optimistic about porting upb to Rust.

What is No-Panic Rust?

No-Panic Rust is a subset of Rust for which `panic!()` is unreachable. Programs written in no-panic Rust are guaranteed never to panic under any circumstances.

For a library, this means we should be able to build a `cdylib` that does not have a panic handler linked into it at all.[4]

We can experiment on godbolt.org to see if we have succeeded or not. Using my tool Bloaty, we can see if the `cdylib` binary is >300Kb (suggesting that the panic handler has been linked in) or <10Ki (suggesting it has not).[5]

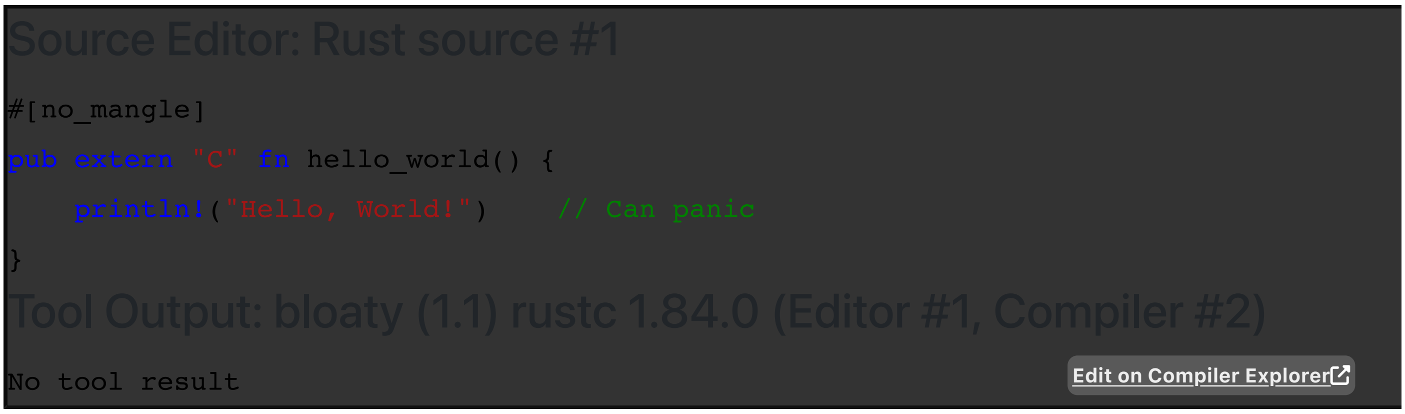
Let's explore this subset a bit. Is "Hello, World" no-panic?

```rust
#[no_mangle]
pub extern "C" fn hello_world() {
    println!("Hello, World!")    // Can panic
}
```

No, per the documentation for `println!()`:

> Panics if writing to `io::stdout` fails.

And indeed, if we try this on Godbolt, we see a big binary:



Source Editor: Rust source #1

```rust
#[no_mangle]
pub extern "C" fn hello_world() {
    println!("Hello, World!")    // Can panic
}
```

Tool Output: bloaty (1.1) rustc 1.84.0 (Editor #1, Compiler #2)

```
No tool result
```

So `println!()` is out. If we want to print to stdout, we'll need to use an API that does not advertise that panic is possible.

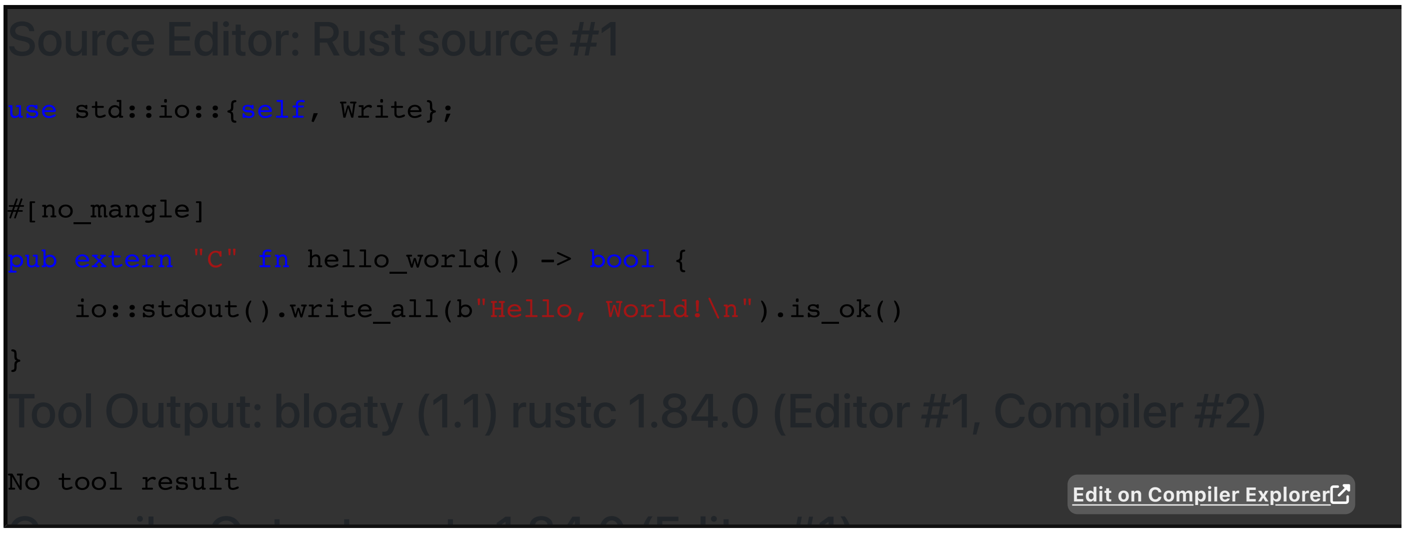
The `stdout` API looks promising, because it has a `write_all()` API that returns a `Result`, which should allow us to handle errors explicitly:

```rust
use std::io::{self, Write};

#[no_mangle]
pub extern "C" fn hello_world() -> bool {
    io::stdout().write_all(b"Hello, World!\n").is_ok()
}
```

This seems like it should be no-panic. We are only calling two APIs, `stdout()` and `write_all()`, neither of which documents a potential panic.

But if we try it, we'll see that panic is indeed reachable in this program somehow.


Source Editor: Rust source #1

```rust
use std::io::{self, Write};

#[no_mangle]
pub extern "C" fn hello_world() -> bool {
    io::stdout().write_all(b"Hello, World!\n").is_ok()
}
```

Tool Output: bloaty (1.1) rustc 1.84.0 (Editor #1, Compiler #2)

```
No tool result
```

From this we have learned that we unfortunately cannot rely on panic annotations in API documentation to determine *a priori* whether some Rust code is no-panic or not. We have to actually try it and observe the results.

How can we diagnose what went wrong? On macOS, the linker has a very handy option called `-why_live`, which will print the chain of symbol references that prevented a symbol from being dead-stripped. We can't access it on Godbolt unfortunately, but on macOS we can run this command:

```
$ RUSTC_LOG=rustc_codegen_ssa::back::link=info \
  RUSTFLAGS="-C link-arg=-Wl,-why_live,_rust_panic" \
  cargo build --release 2>&1 | rustfilt
```

This results in the following output, with extraneous details removed:

```
_core::panicking::panic from [...]
  _core::ops::function::FnOnce::call_once from [...]
    l_anon.56b0c16dbe4596c74313e318a3dfaa78.520 from [...]
      _std::sync::once_lock::OnceLock<T>::initialize from [...]
        _std::io::stdio::stdout from [...]
          _hello_world from [...]
```

The panic reference apparently comes from `_core::ops::function::FnOnce::call_once`, which is called from `_std::io::stdio::stdout`.

This seems to suggest that Rust's standard library does not meet the criteria given above, because it is capable of panicing even in APIs like `std::io::stdout()` that do not document a panic-worthy precondition.

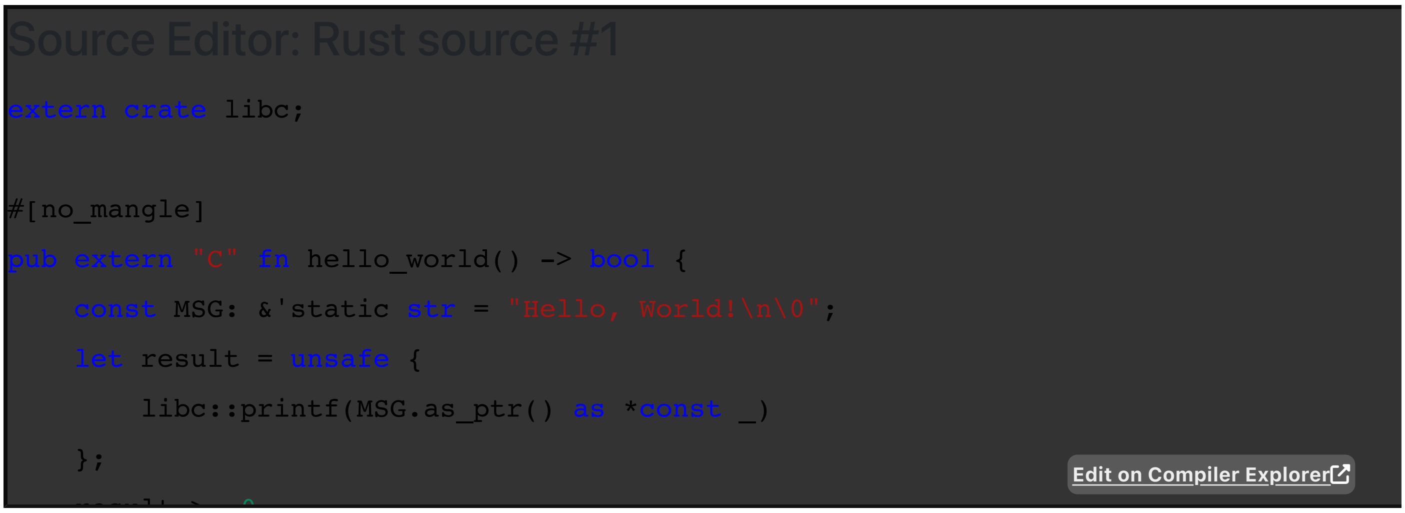
This also implies that we need tests that check for the no-panic property. It's not enough to check once that the code is no-panic, we need to make sure it *stays* no-panic over time, even as our project and our dependendencies evolve.

To get a fully no-panic version of “Hello, World”, we have to reach for the C library `libc`. This makes sense, since the C library is generally written to return all errors as status codes or `errno`. Unfortunately this means turning to `unsafe`:

```
extern crate libc;

#[no_mangle]
pub extern "C" fn hello_world() -> bool {
    const MSG: &'static str = "Hello, World!\n\0";
    let result = unsafe {
        libc::printf(MSG.as_ptr() as *const _)
    };
    result >= 0
}
```

And checking on Godbolt, we see the small binary that confirms that this library is indeed no-panic:



```
extern crate libc;

#[no_mangle]
pub extern "C" fn hello_world() -> bool {
    const MSG: &'static str = "Hello, World!\n\0";
    let result = unsafe {
        libc::printf(MSG.as_ptr() as *const _)
    };
```

Opt No-Panic

What about adding two numbers? Is this no-panic?

```
#[no_mangle]
pub extern "C" fn hello_world(a: i32, b: i32) -> i32 {
    a + b
}
```

This is a trick question: this is no-panic in opt mode only. For numeric operations like addition, Rust introduces overflow checks (which panic on failure) in debug mode, but leaves them out of opt builds.

We can observe this on Godbolt if we add separate panes for opt and non-opt builds:[6]

Source Editor: Rust source #1

```
#[no_mangle]
pub extern "C" fn hello_world(a: i32, b: i32) -> i32 {
    a + b
}
```

Tool Output: Bloaty (opt build)

```
No tool result
```

Compiler Output: rustc 1.84.0 (Editor #1)

Flags: `--crate-type=cdylib -O`

```
.plt:
 push    QWORD PTR [rip+0x2faa]          # 3fd0 <_GLOBAL_OFFSET_TABLE_+0x8>
 jmp     QWORD PTR [rip+0x2fac]          # 3fd8 <_GLOBAL_OFFSET_TABLE_+0x10>
 nop     DWORD PTR [rax+0x0]
hello_world:
```

Edit on Compiler Explorer

This essentially creates a new class of code, which is "no-panic in opt, but can panic in dbg".

For the case of upb, this seems like a great option, because it gives us extra consistency checks in debug mode without suffering the problems of panic in release builds. It is essentially the Rust equivalent of `assert()` in C. Overflow by itself does not represent a safety issue, so we are not giving up safety by leaving the panics out of opt builds.

Rust's Standard Library

What about using standard containers like `Vec`?

```rust
use std::hint::black_box;

#[no_mangle]
pub extern "C" fn hello_world() {
    let vec: Vec<u32> = Vec::new();
    black_box(vec);
}
```

It turns out this is also "opt no-panic" code (perhaps `Vec` is internally performing some arithmetic which can overflow):

Source Editor: Rust source #1

```
use std::hint::black_box;

#[no_mangle]
pub extern "C" fn hello_world() {
    let vec: Vec<u32> = Vec::new();
    black_box(vec);
}
```

Tool Output: Bloaty (opt build)

```
No tool result
```

Compiler Output: rustc 1.84.0 (Editor #1)

Flags: `--crate-type=cdylib -O`

```
.plt:
 push    QWORD PTR [rip+0x2f9a]          # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
```

Edit on Compiler Explorer

But once we try to actually push elements into the `Vec`, we’re squarely out of no-panic Rust:

Source Editor: Rust source #1

```
use std::hint::black_box;

#[no_mangle]
pub extern "C" fn hello_world() {
    let mut vec: Vec<u32> = Vec::new();
    vec.push(1);
    black_box(vec);
}
```

Tool Output: Bloaty (opt build)

```
No tool result
```

Compiler Output: rustc 1.84.0 (Editor #1)

Flags: `--crate-type=cdylib -O`

```
core::str::pattern::simd_contains::{{closure}}:
```

Edit on Compiler Explorer

`Vec` does have a few APIs that will surface allocation errors instead of panicking. Theoretically, this code should be no-panic:

```
#![feature(vec_push_within_capacity)]

use std::hint::black_box;

```

```rust
#[no_mangle]
pub extern "C" fn hello_world() -> bool {
    let mut vec: Vec<u32> = Vec::new();
    if !vec.try_reserve(1).is_ok() {
        return false;
    }
    if !vec.push_within_capacity(1).is_ok() {
        return false;
    }
    black_box(vec);
    true
}
```

This requires the nightly compiler, but I was able to make this work as no-panic on macOS. For some reason, it did not work with the nightly compiler on Godbolt, which appears to always include the panic runtime no matter what I do, even for a trivial library. I was not able to figure out why.

The Rust standard library was not really designed to be no-panic. For example, memory allocation failure will panic in most cases. If we want to be no-panic, we will probably have to avoid most of the standard library. Realisticaly we will probably want to go fully `#![no_std]`.

A Dance With The Optimizer

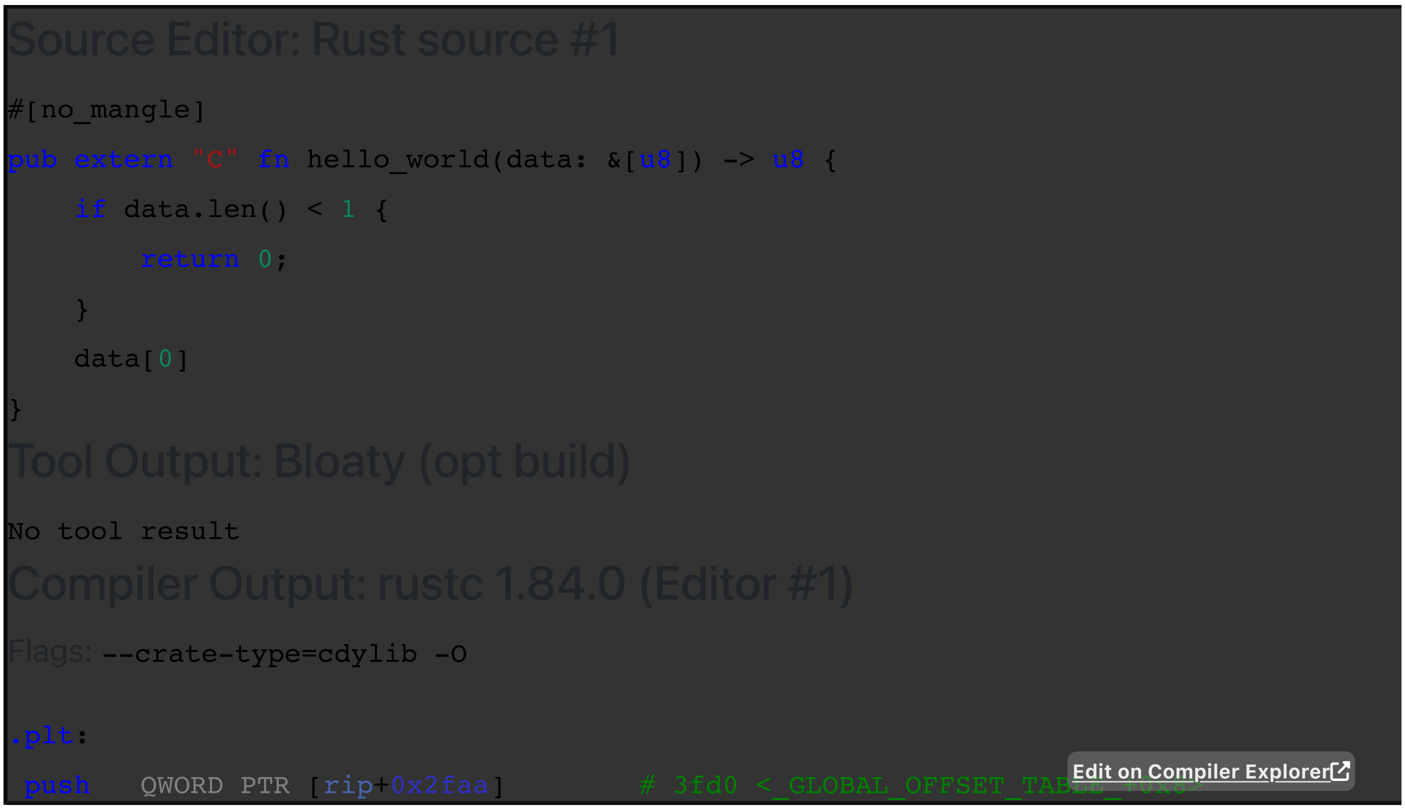
Here is another trick question: is this no-panic Rust?

```rust
#[no_mangle]
pub extern "C" fn hello_world(data: &[u8]) -> u8 {
    if data.len() < 1 {
        return 0;
    }
    data[0]
}
```

On one hand, the slice index operation clearly documents that it may panic. On the other hand, the docs say that this panic will only be triggered if the index is out of bounds, and we have inserted a guard to ensure that it never is. So is the panic reachable?

If we use our minds to reason about the code, we would conclude that panic is unreachable. The compiler is capable of reaching the same conclusion, but only if we run the optimizer, which can prove through a series of optimizations that the bounds check will never fail.

So this example ends up being “opt no-panic”, just like our arithmetic operation, but for an entirely different reason!


Source Editor: Rust source #1

```rust
#[no_mangle]
pub extern "C" fn hello_world(data: &[u8]) -> u8 {
    if data.len() < 1 {
        return 0;
    }
    data[0]
}
```

Tool Output: Bloaty (opt build)

```
No tool result
```

Compiler Output: rustc 1.84.0 (Editor #1)

Flags: `--crate-type=cdylib -O`

```
.plt:
 push    QWORD PTR [rip+0x2faa]          # 3fd0 <_GLOBAL_OFFSET_TABLE_+0x8>
```

This is quite an interesting result that totally changed my thinking about Rust's bounds checks.

My previous perspective was that Rust will insert all of these unnecessary bounds checks, bloating the code and slowing it down for no reason. But our pre-existing C code is not throwing caution to the wind and hoping for the best. Every place that we perform an index operation in C, it's because we believe we have a proof that the index is in bounds. To avoid the bounds checks in Rust, we just need to express this proof in a way that the Rust optimizer can understand. This is what I call the "dance with the optimizer."

A Slightly More Dangerous Dance

In the example above, the bounds check is eliminated using only safe code, but there are other cases where we might need to use unsafe code to help the optimizer know about program invariants that cannot be easily derived from the program flow.

For example, consider this (admittedly contrived) program:

```rust
pub struct S<'a> {
    data: &'a[u8],
    ofs: usize,   // Invariant: ofs < data.len()
}

impl<'a> S<'a> {
    pub fn new(data: &[u8]) -> Option<S> {
        match data.len() {
            0 => None,
            n => Some(S{data: data, ofs: n - 1}),
        }
    }

    pub fn get(&self) -> u8 {
        self.data[self.ofs]
    }
}

#[no_mangle]
pub extern "C" fn hello_world(s: &S) -> u8 {
    s.get()
}
```

In this program, our struct `S` has an invariant that the offset `S::ofs` will always be in bounds. This invariant effectively guarantees that the bounds check in `S::get()` will never fail. And we can strongly guarantee that the invariant holds, because it is enforced by our `new()` function which is the only code that sets these struct members.

But the optimizer isn't capable of reasoning at this level, so it thinks that the panic is reachable, and keeps the bounds check in the program, even in opt mode:

Source Editor: Rust source #1

```
pub struct S<'a> {
    data: &'a[u8],
    ofs: usize,    // Invariant: ofs < data.len()
}

impl<'a> S<'a> {
    pub fn new(data: &[u8]) -> Option<S> {
        match data.len() {
            0 => None,
            n => Some(S{data: data, ofs: n - 1}),
        }
    }

    pub fn get(&self) -> u8 {
        self.data[self.ofs]
    }
}

#[no_mangle]
pub extern "C" fn hello_world(s: &S) -> u8 {
    s.get()
}
```

Edit on Compiler Explorer

To make this no-panic, we need to help the compiler out by reminding it that this struct invariant holds in the critical path:

```rust
use std::hint::assert_unchecked;

pub struct S<'a> {
    data: &'a[u8],
    ofs: usize,    // Invariant: ofs < data.len()
}

impl<'a> S<'a> {
    fn check_invariant(&self) {
        unsafe { assert_unchecked(self.ofs < self.data.len()) }
    }

    pub fn new(data: &[u8]) -> Option<S> {
        match data.len() {
            0 => None,
            n => {
                let s = S{data: data, ofs: n - 1};
                s.check_invariant();
                Some(s)
            }
        }
    }

    pub fn get(&self) -> u8 {
        self.check_invariant();
        self.data[self.ofs]
    }
}

#[no_mangle]
pub extern "C" fn hello_world(s: &S) -> u8 {
    s.get()
}
```

This makes use of `std::hint::assert_unchecked`, a very sharp tool for making soundness promises to the compiler. Here we use it to inform the compiler of our struct invariant. This has the desired effect of making this “opt no-panic”:

Source Editor: Rust source #1

```rust
use std::hint::assert_unchecked;

pub struct S<'a> {
    data: &'a[u8],
    ofs: usize,    // Invariant: ofs < data.len()
}

impl<'a> S<'a> {
    fn check_invariant(&self) {
        unsafe { assert_unchecked(self.ofs < self.data.len()) }
    }

    pub fn new(data: &[u8]) -> Option<S> {
        match data.len() {
            0 => None,
            n => {
                let s = S{data: data, ofs: n - 1};
                s.check_invariant();
                Some(s)
            }
        }
    }
```

Edit on Compiler Explorer

This definitely requires care; we have to be very sure that the predicate we pass to `assert_unchecked` is true. Luckily we can fuzz against this assertion to increase our confidence (in debug mode, `assert_unchecked` will panic if the condition is not true). Used judiciously, it can be a powerful tool for explicitly expressing to Rust the invariants we were relying on to make index operations safe in C.

Conclusion

No-Panic Rust is not for the faint of heart. It requires a lot of careful, detailed work, and forces you to give up some niceties of Rust, like the standard library. But if we are diligent, it can give us the performance, code size, and error reporting behavior of a C library with the extra safety that comes from Rust.

This extra safety comes from the fact that Rust will automatically insert bounds checks anywhere it cannot prove that an access is safe. This puts the burden on us to justify to the compiler in every case why the bounds check is safe to elide. In some cases this will mean detecting a bounds violation explictly and reporting the error to the caller (especially in parsers, where we do not know whether the input is valid or not). In other cases, we may know through a program invariant that the index will always be in-bounds, and we will need to communicate this invariant to Rust.

I should be clear that I have not yet attempted this technique at scale, so I cannot report on how well it works in practice. For now it is an exciting future direction for upb, and one that I hope will pay off.

To make this technique practical, we need a tool that can diagnose where a panic handler was reachable from. The main technique we used in this article (looking at binary size) does not give us any information about where a panic came from. On macOS, the `-why_live` linker option is perfect for this. I hope other linkers like LLD will add support for this option also. If not, a standalone tool could be written that analyzes a binary after it's linked to find the chain of references that lead to a panic handler.

It would be nice if Rust made it easier to stay within the no-panic subset. It's clear that writing no-panic code is not a core use case that the language focuses on, but there are many situations (embedded, Linux Kernel, etc) where we want to avoid panics. It would be nice if functions or even crates could advertise themselves as no-panic and have the compiler enforce this transitively. Changing a function from no-panic to panicking would then be an API-breaking change.

1. There is some interesting discussion in Enforcing no-std and no-panic during build, which links to some relevant Linux kernel mailing list threads. Another interesting thread is Negative view on Rust: panicking. ↩

2. If we are willing to go `#![no_std]`, we can mitigate this code size overhead by writing our own panic handler, which we could engineer to be much smaller than the std one. This does address the code size concern, but it does not compose well, as there can only be one panic handler for an entire binary, so it doesn't make sense for a library to provide one. ↩

3. Some Rust panics can technically be caught with catch_unwind, but this is full of caveats and is not designed as an error recovery mechanism. ↩

4. We choose `cdylib` instead of `staticlib` for this exercise, because `cdylib` will invoke the linker to produce a `.so`. This will perform a garbage-collection pass that discards unreachable code, so that we only count code that would actually get pulled in if you statically linked the library into a binary. ↩

5. There are some crates specifically designed to test directly for the no-panic condition, including no_panic, panic_never, and no_panics_whatsoever. None of them are ideal for our purposes (the second two are only for binaries, not libraries, and the first has to be applied to every relevant function individually), and anyway none of them are available in Godbolt. These generally work by trying to create linker errors if the panic handler is linked in. ↩

6. Unfortunately this does not appear to work with the nightly Rust compiler. On nightly, the resulting binary is >300Kb, indicating that the panic runtime was linked in. This seems like a regression, and I have not been able to diagnose why this is. ↩