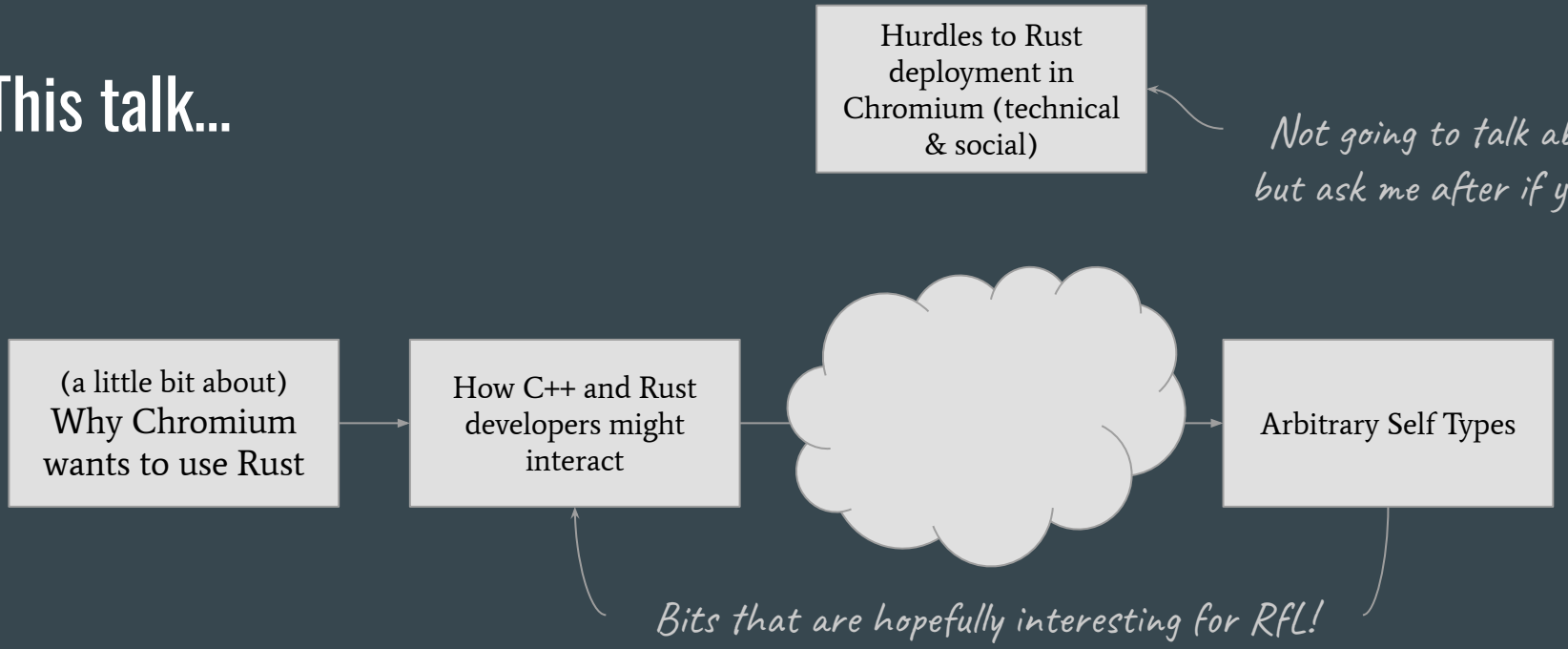
Rust references considered harmful...?

...

(at least, if they're pointing to C/C++ things)

Adrian Taylor, Chromium, Google
ade@hohum.me.uk

This talk...


Hurdles to Rust deployment in Chromium (technical & social)
Not going to talk about - but ask me after if you like!
(a little bit about) Why Chromium wants to use Rust
How C++ and Rust developers might interact
Arbitrary Self Types
Bits that are hopefully interesting for RfL!

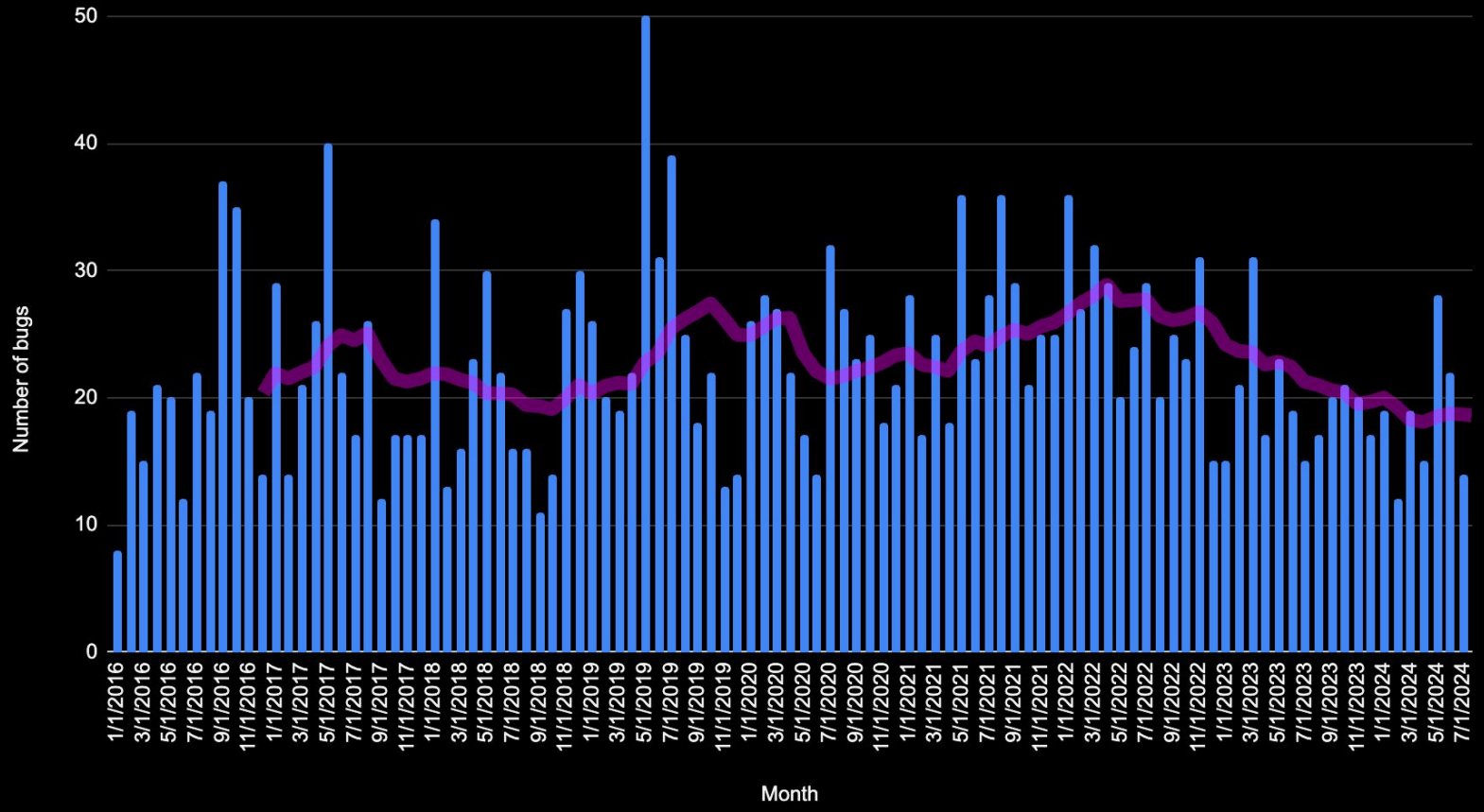
Rust references must never refer to C++ data

At least, that's Chromium's current belief
Does it apply to the Linux kernel?

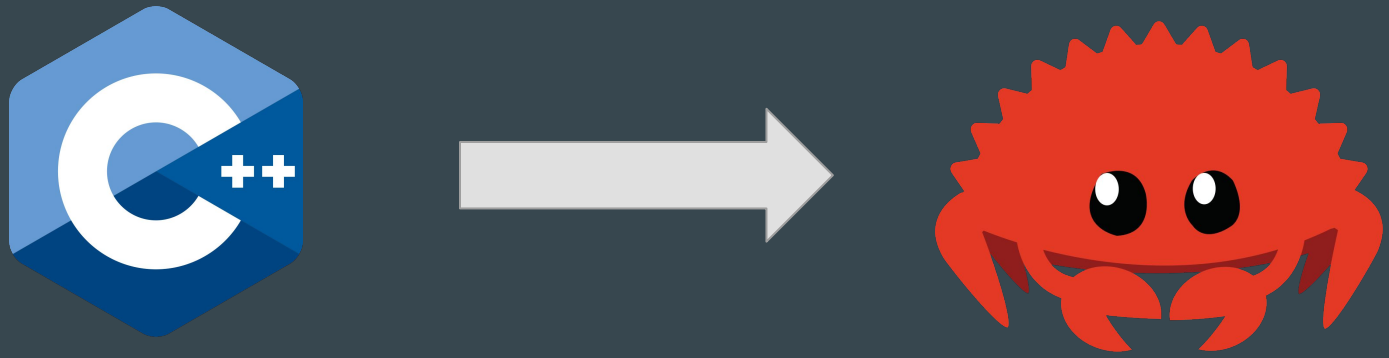
Background: Chromium

C++ has errors

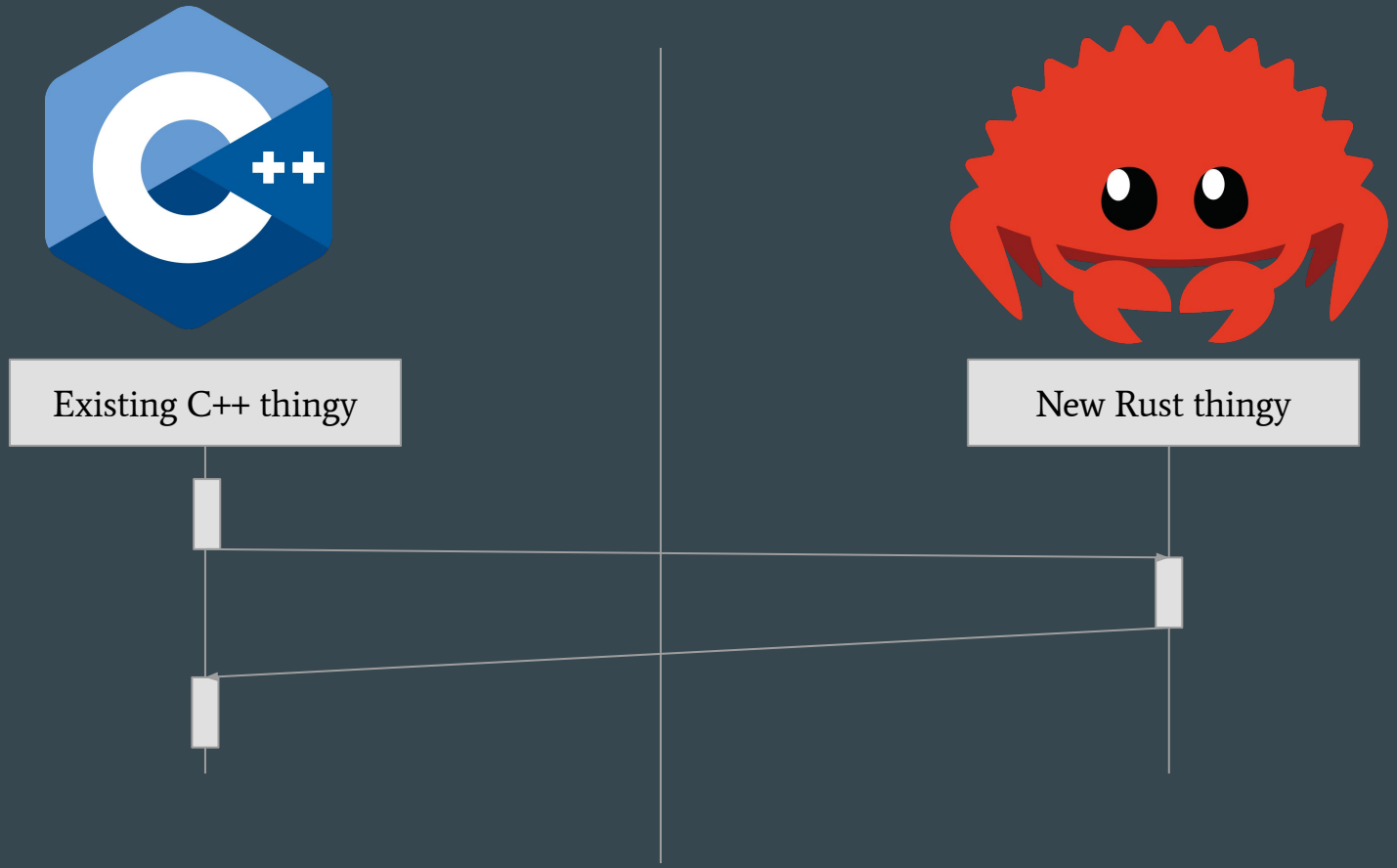
Use-after-free bugs with security consequences in Chromium per month



So, rewrite Chromium in Rust?



Nope.



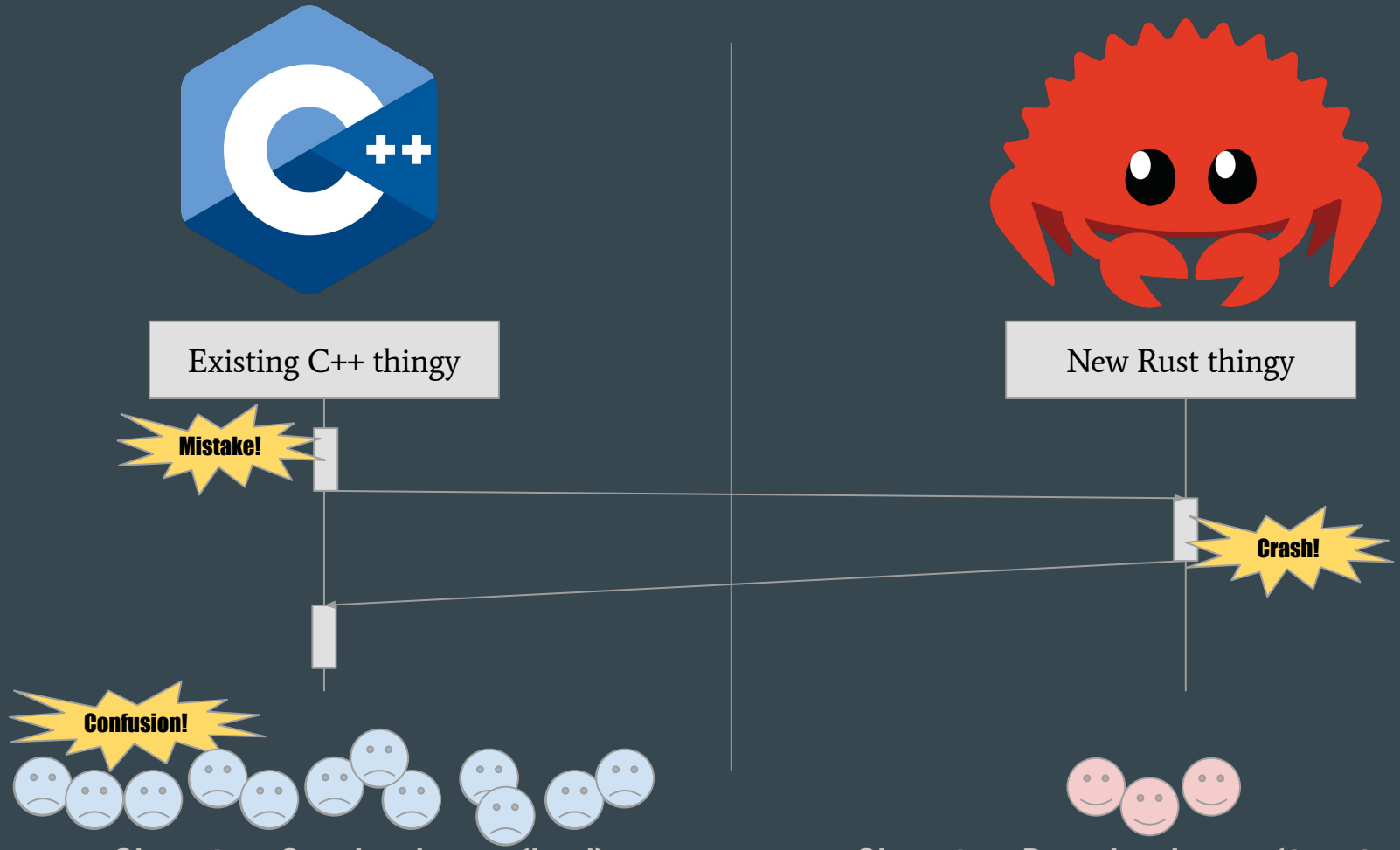
Write *new* bits of Chromium in Rust. Interop!

Existing C++ thingy

New Rust thingy

Chromium C++ developers (lots!)

Chromium Rust developers (few, for now)


Existing C++ thingy
New Rust thingy
Mistake!
Crash!
Confusion!
Chromium C++ developers (lots!)
Chromium Rust developers (few, for now)

Crashes at-a-distance in Rust

Tolerable crashes

- Buffer overflows
- Use-after-free
- Hitting assertions

Intolerable crashes

- UB caused by a reference pointing to uninitialized data
- UB caused by multiple concurrent mutable references
- UB caused by mutation of underlying data while a reference exists

It **must not be possible** to cause these Rust crashes by mistakes over in C++

The logical conclusion:

Rust references must never refer to C++ data

Will C kernel developers get cross if they cause weird UB crashes-at-a-distance in Rust?

... but maybe we're wrong...?

The happy place: cxx!

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        include!("example/include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
    }
}
```

Why cxx is OK

- Used for **narrow** interfaces
- Forces you to spell out the entire language boundary ⇒ you'll think through lifetimes
- (plus, for opaque C++ types, references made non-overlapping: no UB)

- Experience shows cxx is *safe in practice* everywhere it's been used, even though references pass across the language boundary
- But for **wider** interfaces, **automatically generated**, we need something different

So for broad-scale, autogenerated interfaces, what do we do?

Can we use cxx-like opaque types for autogenerated interfaces?

Maybe...?

- With `MaybeUninit` and `UnsafeCell`, we can make `&T` *technically* safe to point at C++ data without risk of UB
- **But** not `&mut T` - so we'd have to model all of these as &T which seems to be too coarse
  - `const T*` (especially the `this` pointer)
  - `T*` (also `this`)
  - `const T&`
  - `T&`
- So we still don't want to use Rust references to point to C++ data

CppRef<T>

`CppRef<T>` / `CppPtr<T>`

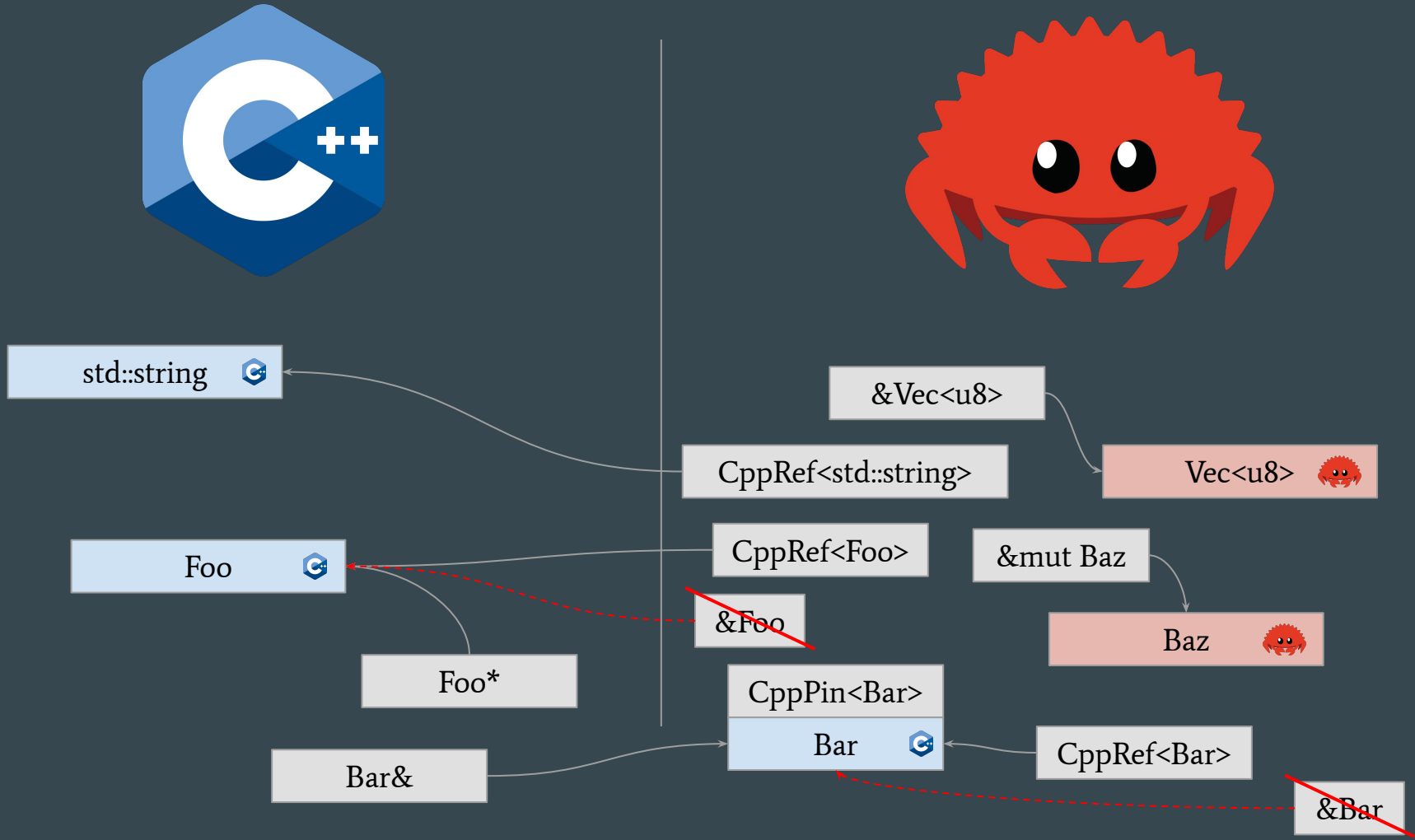
`CppPin<T>` / `CppValue<T>`

```rust
#[repr(transparent)]
pub struct CppRef<'a, T: ?Sized> {
    ptr: *const T,
    phantom: PhantomData<&'a T>,
}


#[repr(transparent)]
pub struct CppPin<T: ?Sized>(T);
```

Like &T, but without any of the usual Rust rules

Never vends Rust references to its contents - only CppRef<T>


std::string
&Vec<u8>
CppRef<std::string>
Vec<u8>
Foo
CppRef<Foo>
&mut Baz
&Foo
Baz
Foo*
CppPin<Bar>
Bar
CppRef<Bar>
Bar&
&Bar

Is this ergonomic?

```
let farm = new_cpp_pin!(cpp::Farm);
let goat: CppRef<cpp::Goat> = farm.as_cpp_ref().get_goat();
goat.bleat();
```

CppRef<Goat> goes back to C++

CppRef<Goat> comes from C++

- No dereferencing in Rust
- No conversion to a Rust reference
- CppRef<T> is pretty much just an opaque token from Rust's perspective

Requires "arbitrary self types" unstable feature - working towards stabilizing

```
// Autogenerated

impl Goat {
  fn bleat(self: CppRef<Self>) {
    _call_cpp_Goat_bleat_via_c_abi(self.ptr)
  }
}
```

RfL needs this too for your kernel Arc<T> and similar

```
impl SomeKernelType {
  fn some_kernel_thing(self: KernelArc<T>) {
  }
}
```

Arbitrary self types

```
trait Receiver {
    type Target: ?Sized;
}
```

```
impl Foo {
    fn by_value(self /* self: Self */);
    fn by_ref(&self /* self: &Self */);
    fn by_ref_mut(&mut self /* self: &mut Self */);
    fn by_box(self: Box<Self>);
    fn by_rc(self: Rc<Self>);

    fn by_custom_ptr(self: CustomPtr<Self>);
}

struct CustomPtr<T>(*const T);

impl<T> Receiver for CustomPtr<T> {
    type Target = T;
}
```

RFC

RfL takeaways

- Please continue to help and support Arbitrary Self Types stabilization (and thanks for your help so far!)

- Decide whether kernel C programmers will get cross if their mistakes cause weird Rust UB crashes
  - If so, and if your Rust/C interface is sufficiently complex, maybe you want to ban Rust references to C types too
  - Or maybe it's good enough to keep using opaque types (UnsafeCell, MaybeUninit) and forbid &mut

- Maybe lessons can be learned more generally from our experiences (technical & social) in deploying Rust in Chromium - feel free to chat later!

Q&A/discussion