Bridging Rust and C++ with cxx

Igor Khanin

Presentation for RustTLV meetup
January 2025

About Me

- Pretending to write C++ professionally for ~10 years
  - And a Rust enthusiast for less than that
- Currently working as a Senior Software Engineer @ Fireblocks
  - Opinions are my own

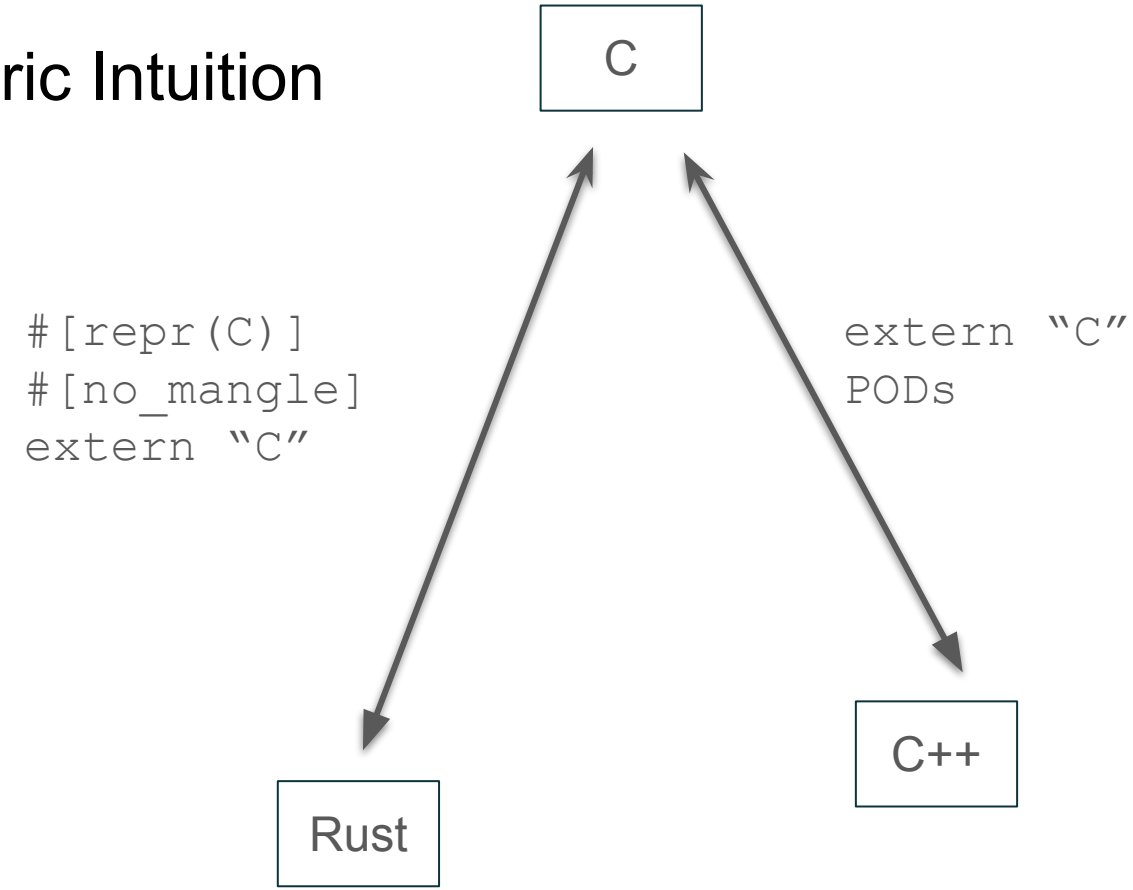
GitHub: https://github.com/IgKh/

E-mail: igor@khanin.biz

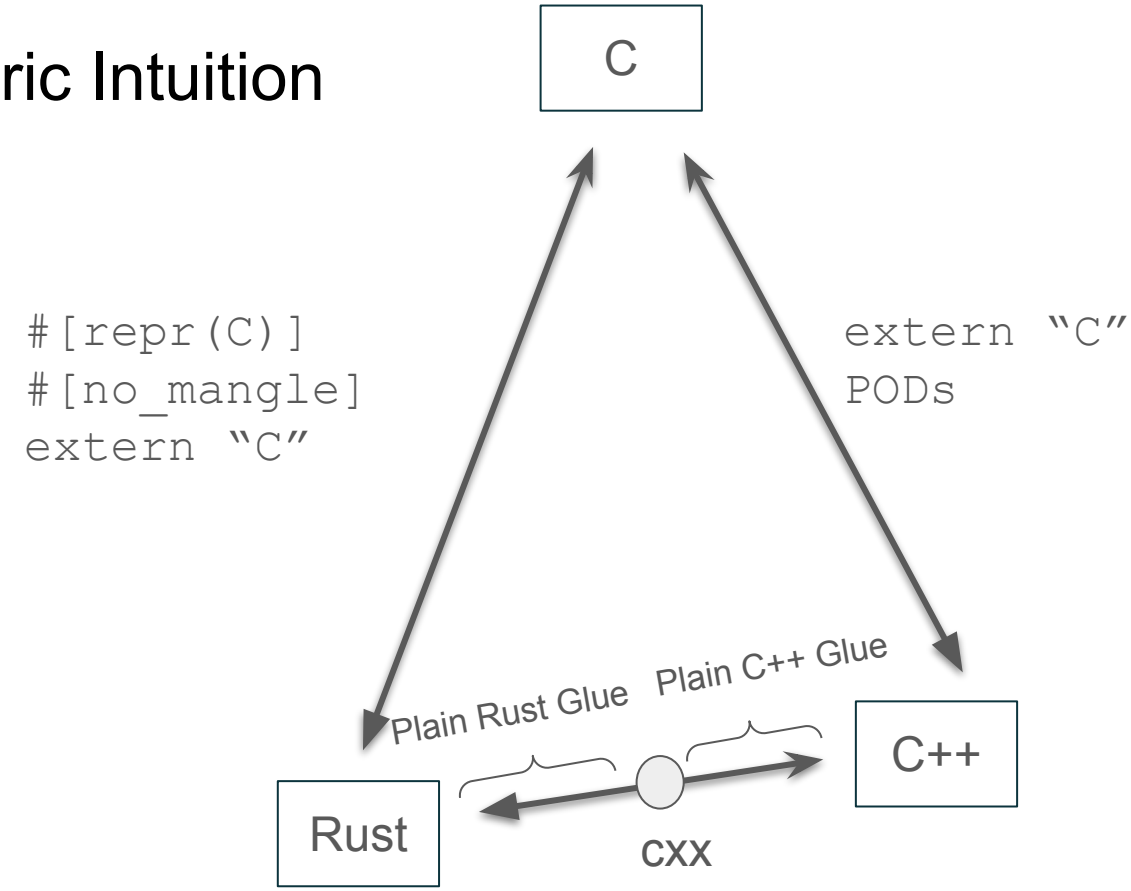
Motivation

- C++ and Rust have many similar concepts
  - RAII, references, strong type system, Zero-overhead principle…
- Gradual introduction of Rust is therefore natural for organizations with large C++ code bases that want to improve safety
- There is need to both call from Rust to C++ and from C++ to Rust
- BUT:
  - There are also differing concepts that don’t easily map, e.g. traits vs inheritance, lifetimes
  - Neither has a stable ABI
    - C ABI is the least common denominator

Geometric Intuition



Geometric Intuition



Enter cxx.rs

- Started by David Tolnay in 2019
- Fundamental approach: by controlling both sides of the FFI boundary, it is possible to ensure that they agree on the memory layout and semantics of all types that cross the boundary
- This has some benefits:
  - Eliminate the intrinsic unsafety introduced by the bindings themselves
  - Direct bindings that don’t necessarily require marshalling
- But also some disadvantages…

What Can Cross the FFI Boundary?

- Simple (primitive) types, for example:
  - `i32 <-> int32_t`
  - `u8 <-> uint8_t, unsigned char`
  - `usize <-> size_t`
  - `f64 <-> double`
  - `bool <-> bool`
- Opaque C++ types
  - Types that are defined in C++. Rust code can only access them indirectly (by reference) to call exposed methods.
- Opaque Rust types
  - Types that are defined in Rust. C++ code can only access them indirectly to call exposed methods.

What Can Cross the FFI Boundary?

- Specific complex types. For example:

Rust “Leg”	C++ “Leg”
`Box<T>`	`rust::Box<T>`
`cxx::UniquePtr<T>`	`std::unique_ptr<T>`
`String`	`rust::String`
`cxx::CxxString`	`std::string`
`&T`	`const T&`
`Pin<&mut T>`	`T&`
`Result<T, E>`	`Exceptions!`

More at https://cxx.rs/bindings.html

What can Cross the FFI Boundary?

- Shared Types
  - Types defined as part of the FFI definition. Both sides know their definition and can access fields, hold by value, etc.

- Shared types can be:
  - Simple enums
  - Structs of anything else supported by cxx

Example

```rust
pub trait LogSink {
    fn log_message(&self, message: &str);
}

pub struct PageExtractor {
    logger: &'static dyn LogSink,
}

impl PageExtractor {
    pub fn new(logger: &'static dyn LogSink) -> Self {
        Self { logger }
    }

    pub fn extract_from_pdf(&mut self, source: &[u8]) -> Result<Vec<String>, Error> {
        if source.is_empty() {
            return Err(Error::EmptySource);
        }
        self.logger.log_message(&format!("Got source of {} bytes", source.len()));
        Ok(vec!["First text".to_string(), "Second text".to_string()])
    }
}
```

Interface Definition

```rust
#[cxx::bridge]
mod ffi {
    // Shared types go here

    extern "Rust" {
        type PageExtractorWrapper;

        fn create_extractor(logger: &'static RustLogSink) -> Box<PageExtractorWrapper>;

        fn extract_from_pdf(&mut self, source: &[u8]) -> Result<Vec<String>>;
    }

    unsafe extern "C++" {
        include!("rustlogsink.h");

        type RustLogSink;

        fn send_message(&self, message: &CxxString);
    }
}
```

Rust Glue Code

```rust
// Needed due to orphan rule
struct PageExtractorWrapper {
    inner: PageExtractor
}

impl PageExtractorWrapper {
    fn extract_from_pdf(&mut self, source: &[u8]) -> Result<Vec<String>, magiclib::Error>
        self.inner.extract_from_pdf(source)
    }
}

impl magiclib::LogSink for ffi::RustLogSink {
    fn log_message(&self, message: &str) {
        cxx::let_cxx_string!(cxx_message = message);
        self.send_message(&cxx_message);
    }
}

fn create_extractor(logger: &'static ffi::RustLogSink) -> Box<PageExtractorWrapper> {
    Box::new(PageExtractorWrapper { inner: PageExtractor::new(logger) })
}
```

Using from C++

```cpp
#include "rust/cxx.h"

void process_pages(const std::vector<uint8_t>& data)
{
    RustLogSink logger;
    rust::Box<PageExtractorWrapper> extractor = create_extractor(logger);

    try {
        rust::Vec<rust::String> pages = extractor->extract_from_pdf(rust::Slice<const uint8_t>(data));

        for (auto& page : pages) {
            std::cout << "Page " << page.c_str() << std::endl;
        }
    }
    catch (rust::Error& e) {
        std::cout << "Extraction failed: " << e.what() << std::endl;
    }
}
```

Who Drives the Linker?

- Option 1 - cargo handles everything
  - For Rust-first projects with smaller amounts of C++ glue code
  - Using `cxx-build` crate in `build.rs`
- Option 2 - integrate into existing C++ build system
  - Configure cargo to build the crate containing bridge definition as a `staticlib` crate.
  - Have the build system install and run the `cxxbridge` CLI to generate the C++ side of the FFI.
  - Compile all written and generated C++ code, and have the C++ compiler link it with the static library emitted by cargo / `rustc`.
  - For CMake builds, Corrosion automates all of this nicely.

Who Drives the Linker?

- Beware of pitfalls with option 2:
  - Linking with multiple crates containing bridges
  - GCC vs Clang, libc++ vs libstdc++
  - MSVCRT debug runtime mismatch
  - LTO builds and GCC
  - …

Alternatives

- autocxx
  - Not an alternative per-se, but builds on cxx to eliminate the need to write most bridge modules and glue code for projects that mainly call Rust from C++.
- zngur
  - Similar to cxx in basic approach, with different choices that affect the type of glue code that needs to be written.

Thank You!

https://github.com/lgKh/rustlv-cxx-example