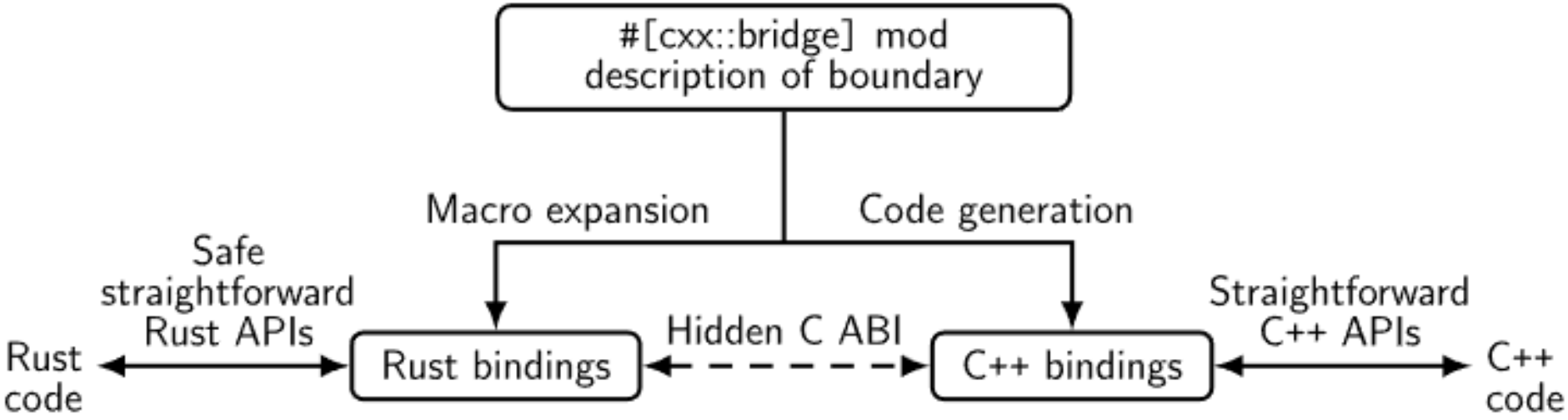
CXX — how to use tools for bindings between Rust and C++

PATRYK JĘDRZEJCZAK

CXX

CXX is a Rust library that provides a safe mechanism for calling C++ code from Rust and Rust code from C++.

FFI boundary



FFI boundary – items

In CXX, the language of the FFI boundary involves 3 kinds of items:

- **Shared structs** – data structures whose fields are made visible to both languages.
- **Opaque types** – their fields are secret from the other language. These cannot be passed across the FFI by value but only behind an indirection, such as a reference *&*, a Rust *Box*, or a C++ *unique_ptr*.
- **Functions** – implemented in either language, callable from the other language.

FFI boundary – example, part 1

```rust
#[cxx::bridge]
mod ffi {
    // Shared structs, whose fields will be visible to both languages.
    struct SharedStruct {
        field1: usize,
        field2: Vec<String>,
    }

    // Code exported from Rust to C++.
    extern "Rust" {
        // Opaque types, only Rust can see the fields.
        type RustType;

        // Functions implemented in Rust.
        fn rust_function(parameter: &RustType) -> bool;
    }
}
```

FFI boundary – example, part 2

```rust
#[cxx::bridge]
mod ffi {
    // Code exported from C++ to Rust.
    unsafe extern "C++" {
        // Headers with the matching C++ declarations.
        include!("crate-name/include/header.h");

        // Opaque types, only C++ can see the fields.
        type CppType;

        // Functions implemented in C++.
        fn cpp_function() -> UniquePtr<CppType>;
    }
}
```

CXX usage – example

We have:

Point – Rust struct,

Line – C++ class.

We want to:

implement method **bool Line::contains_point(const Point &p) const**,

use **contains_point** method in Rust.

CXX usage – example, Rust code

src/main.rs

```rust
pub struct Point {
    x: f64,
    y: f64,
}

impl Point {
    pub fn x(&self) -> f64 {
        self.x
    }

    pub fn y(&self) -> f64 {
        self.y
    }
}

fn main() {
    let line = ffi::new_line(2., -3.);
    let point = Point { x: 1., y: -1. };
    println!("Is point on line? {}", line.contains_point(&point));
}
```

CXX usage – example, C++ code

include/line.h

```cpp
#pragma once
#include <memory>

struct Point;

class Line {
private:
    double a, b;
public:
    Line(double a, double b);
    bool contains_point(const Point &p) const;
};

std::unique_ptr<Line> new_line(double a, double b);
```

src/line.cc

```cpp
#include "example1/include/line.h"
#include <cmath>

const double EPSILON = 1e-5;

Line::Line(double a, double b) : a(a), b(b) {}

bool Line::contains_point(const Point &p) const {
    return fabs(p.y() - (a * p.x() + b)) < EPSILON;
}

std::unique_ptr<Line> new_line(double a, double b) {
    return std::make_unique<Line>(a, b);
}
```

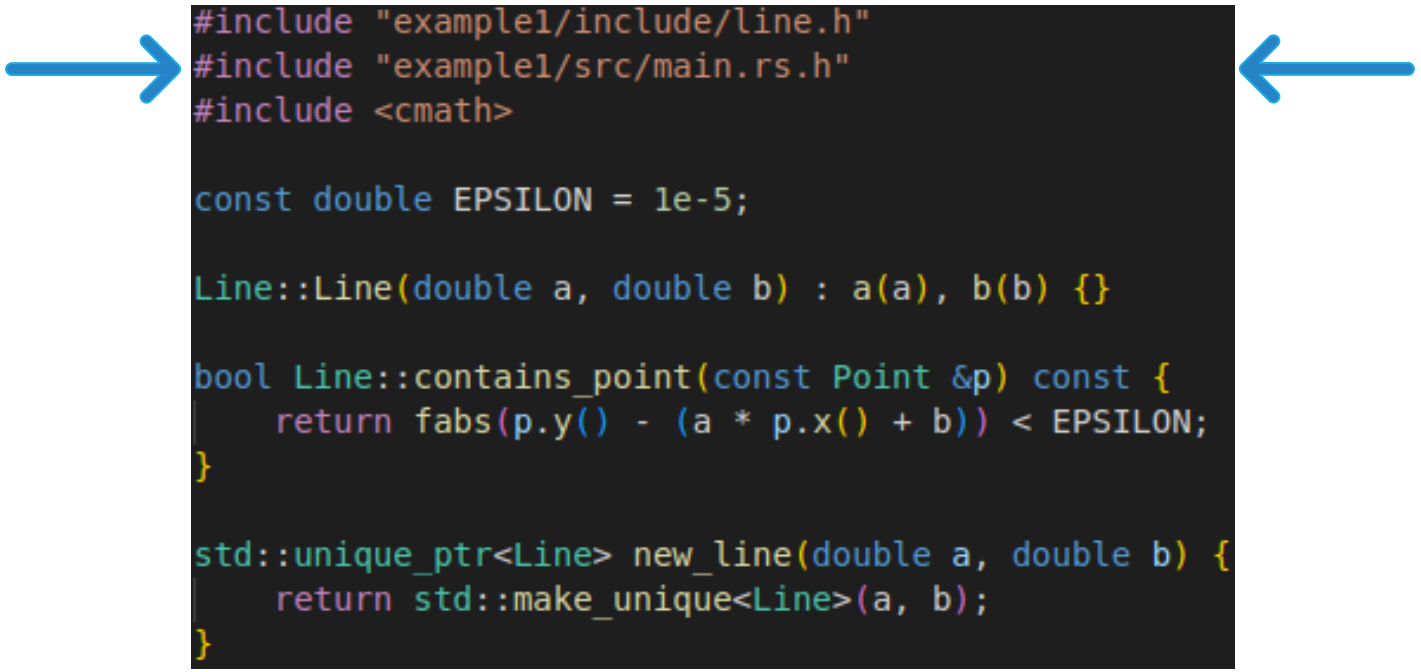
CXX usage – example, FFI boundary

src/main.rs

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

        pub fn x(&self) -> f64;
        pub fn y(&self) -> f64;
    }

    unsafe extern "C++" {
        include!("example1/include/line.h");

        type Line;

        fn contains_point(&self, p: &Point) -> bool;
        fn new_line(a: f64, b: f64) -> UniquePtr<Line>;
    }
}

...
```

CXX usage – example, Rust in C++

src/line.cc



```cpp
#include "example1/include/line.h"
#include "example1/src/main.rs.h"
#include <cmath>

const double EPSILON = 1e-5;

Line::Line(double a, double b) : a(a), b(b) {}

bool Line::contains_point(const Point &p) const {
    return fabs(p.y() - (a * p.x() + b)) < EPSILON;
}

std::unique_ptr<Line> new_line(double a, double b) {
    return std::make_unique<Line>(a, b);
}
```

CXX usage – example, build

build.rs

```rust
fn main() {
    cxx_build::bridge("src/main.rs")
        .file("src/line.cc")
        .flag_if_supported("-std=c++17")
        .compile("example1");
}
```

Cargo.toml

```toml
[dependencies]
cxx = "1.0"

[build-dependencies]
cxx-build = "1.0"
```

More about build scripts: https://doc.rust-lang.org/cargo/reference/build-scripts.html

Built-in bindings

Shared structs and extern functions can use primitive types (*i64* <-> *int64_t)*.

Additionally, some common types can be used. Examples:

Rust	C++
String	rust::String
&[T]	rust::Slice<const T>
CxxString	std::string
Box<T>	rust::Box<T>
UniquePtr<T>	std::unique_ptr<T>
[T; N]	std::array<T, N>

Full list of built-in bindings with restrictions: https://cxx.rs/bindings.html

Built-in bindings – example (*&str <-> rust::Str*)

src/main.rs

```rust
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn print_rust(message: &str);
    }

    unsafe extern "C++" {
        include!("example-str/include/print_cpp.h");
        fn print_cpp(message: &str);
    }
}

fn print_rust(message: &str) {
    println!("{}", message);
}

fn main() {
    ffi::print_cpp("hello from Rust");
}
```

include/print_cpp.h

```cpp
#pragma once
#include "example-str/src/main.rs.h"
#include "rust/cxx.h"

void print_cpp(rust::Str greeting);
```

src/print_cpp.cc

```cpp
#include "example-str/include/print_cpp.h"
#include <iostream>

void print_cpp(rust::Str message) {
  std::cout << message << std::endl;
  print_rust("hello from C++");
}
```

How does CXX work?

CXX uses static analysis of the types and function signatures.

Then it uses a pair of code generators to implement FFI bridge efficiently.

The resulting FFI bridge operates at negligible overhead, i.e. no copying, no serialization, no memory allocation, no runtime checks needed.

C++ generated code can be easily viewed as it is linked into Cargo's target directory under *target/cxxbridge/* (files *main.rs.cc*, *main.rs.h*, *cxx.h*).

More FFI tools

CXX-async:

Extends CXX.

Provides interoperability between asynchronous Rust code using async/await and C++20 coroutines using *co_await*.

Autocxx:

Allows calling C++ from Rust in a more automated way.

It has the safety of CXX whilst generating interfaces automatically from existing C++ headers.

The end

CXX: https://github.com/dtolnay/cxx

CXX tutorial: https://cxx.rs/index.html

Examples used during the presentation: https://github.com/patjed41/CXX-examples