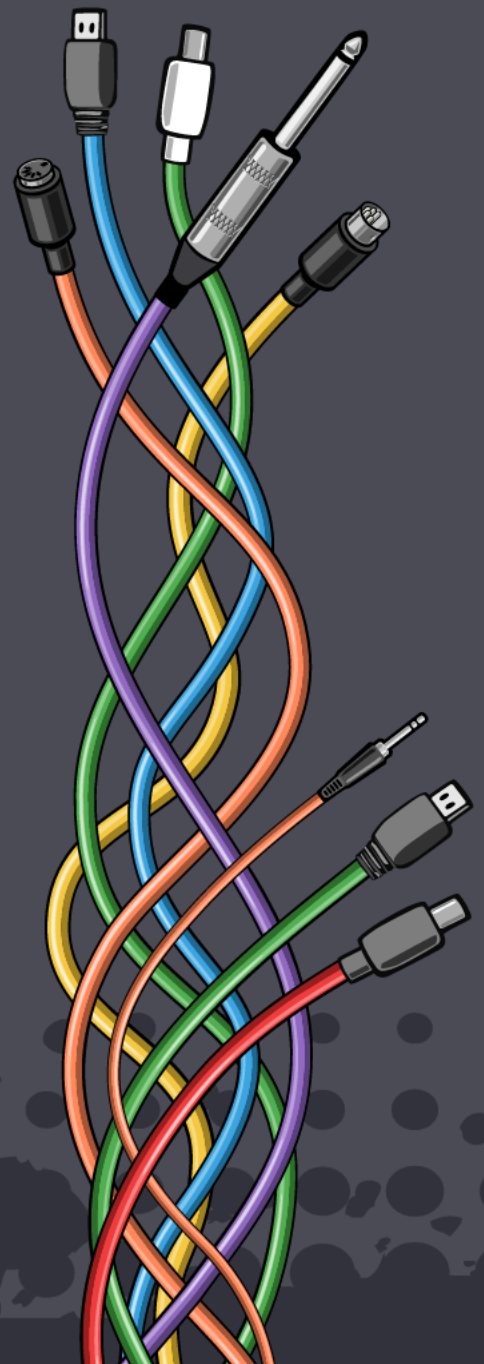


# SAFE AND SOUND

## *USING C++ AUDIO LIBRARIES FROM RUST*

**JAMES HALLOWELL**

ADC25 Bristol

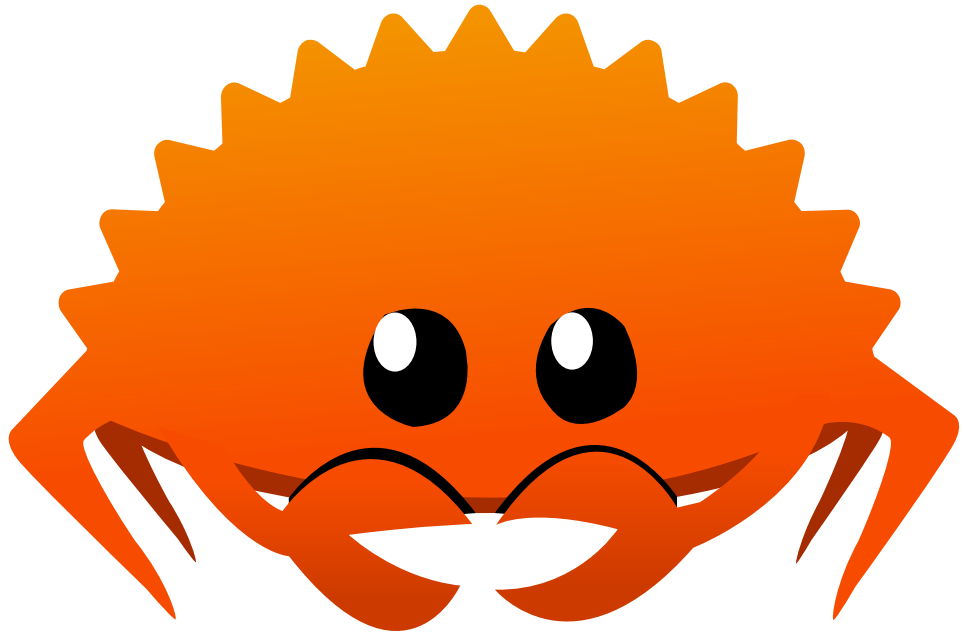
# Hello 👋

- Senior Software Developer @ Focusrite
  - Working on internal tools
- Exploring Rust since ~2018 🦀

# Rust

**"A language empowering everyone to build reliable and efficient software"**

- Why Rust?
  - Performance
  - Reliability
  - Productivity
- Most loved/admired language in Stack Overflow survey 2016-2025
- Increasing industry adoption



# In theory Rust is a great fit for audio applications! 🎵

Rust Audio Community

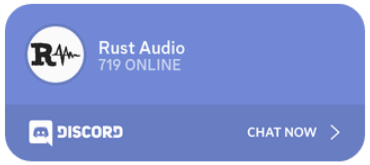
## About

rust.audio is a collection of resources dedicated to the development of audio applications in the Rust programming language. These topics vary from general DSP, to synthesis, to plugins and associated crates.

## Community

### Discord

Join Rust Audio on Discord to keep up with audio development within the Rust ecosystem, get help with DSP, collaborate on projects, and everything in-between.




Join Discord

## Audio crates

Bindings CLAP DAWs Devices DSP Embedded FFT Frameworks Games GUI Languages MIDI Playback Plugin Resampling Streaming Synthesizer VST2 VST3 Windowing

**fundsp**
Audio processing and synthesis library.
DSP

**cpal**
Low-level cross-platform audio I/O library in pure Rust.
Devices

**coremidi**
CoreMIDI library for Rust
MIDI

**rodio**
Audio playback library
Playback

**kira**
Expressive audio library for games
Games Playback

**awedio**
A low-overhead and adaptable audio playback library
Playback Embedded

**rubato**
Asynchronous resampling library intended for audio data
Resampling DSP

**realfft**
Real-to-complex forward FFT and complex-to-real inverse FFT for Rust
FFT DSP

**jack**
Real time audio and midi with JACK.
Devices

# Barriers for adoption 🚧

- C and C++ are the dominant languages for audio
  - *Lots* of existing code
  - *Lots* of expertise in these languages
- "Rewrite it in Rust" not always practical

# ~~Rewrite~~ Reuse it in Rust? 🤔

## How do we call C++ from Rust?

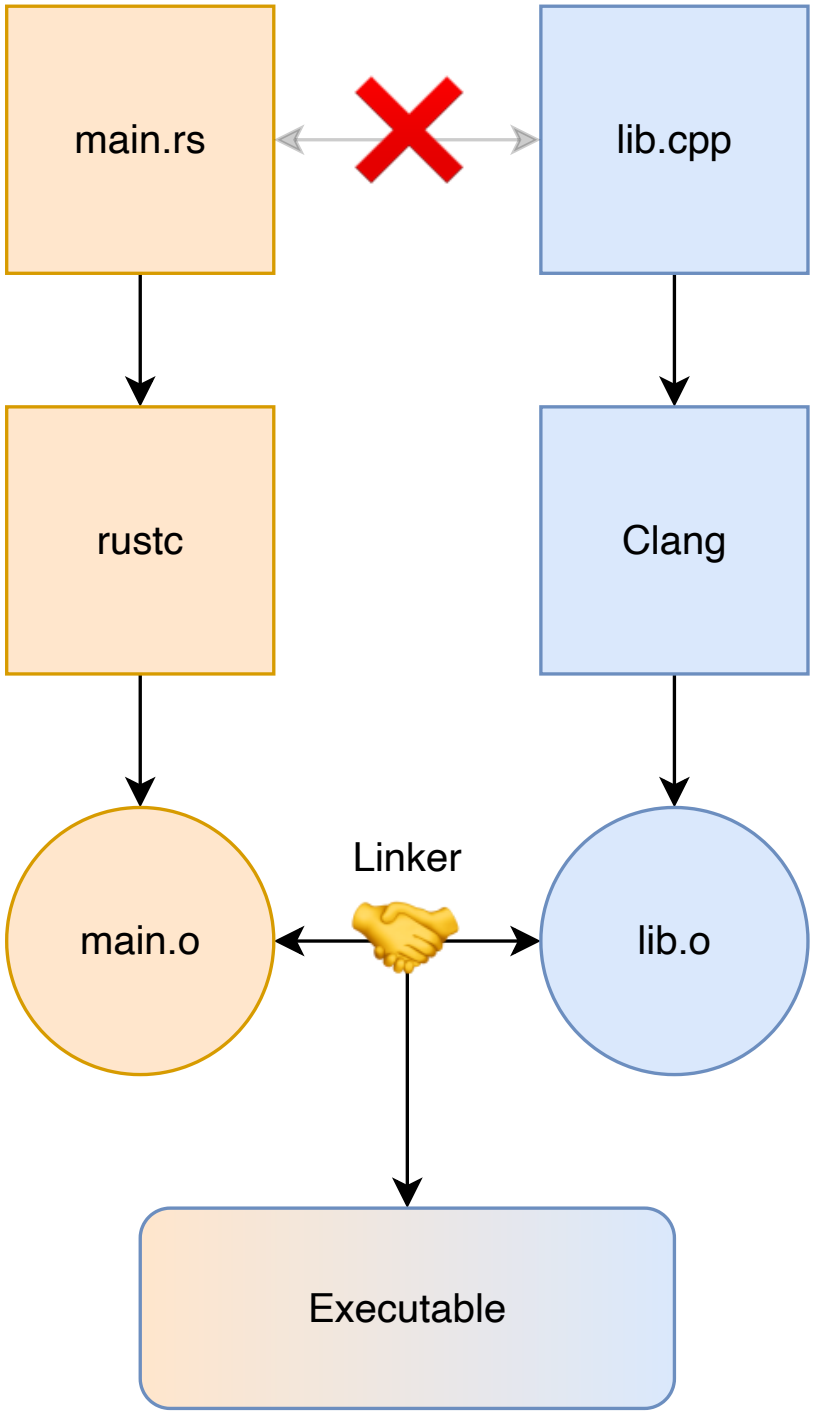
`lib.hpp`

```
void process_audio(AudioBuffer<float>& buffer);
```

`main.rs`

```
include!("lib.hpp");

fn audio_callback(buffer: &mut AudioBuffer<f32>) {
    process_audio(buffer);
}
```

Unfortunately its not going to be this easy...


main.rs
lib.cpp
rustc
Clang
Linker
main.o
lib.o
Executable

# Application Binary Interface (ABI)

- An interface to compiled code
- Size, layout and alignment of types
- Calling conventions
- Name mangling of symbols

# The ABI situation for C++ ⚠️

- No standard C++ ABI
- Dependent on compiler, platform, library...

# C as the *lingua franca* of programming 🗣️

- Rust and C++ both "speak" C
- Platform C ABI is relatively stable and predictable
- Need to lower to C for Rust/C++ interop

# Certain C++ features can't be expressed in Rust

- Move constructors
- Exceptions
- Overloaded functions
- ...

# Moves in Rust

- Move-by-default
- Destructive moves
- A move is just a bitwise copy*

```rust
let x = String::from("Hello, world!");
let y = x;

// error: borrow of moved value: `x`
println!("{x}");
```

# Moves in C++

- Copy-by-default
- Non-destructive moves
- Move constructors

```cpp
std::string x = "Hello, world!";
std::string y = std::move(x);

// x is still accessible
std::println("{}", x);
```

*may be optimised away by the compiler

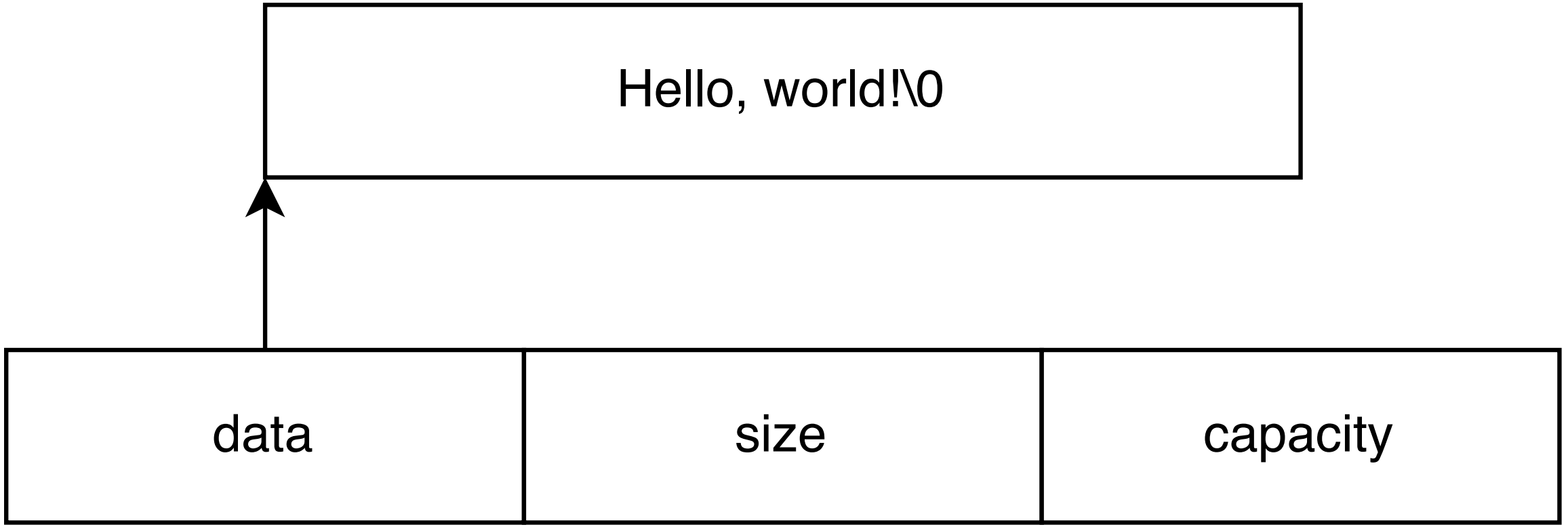
# Move semantics mismatch ⚠️

## Rust

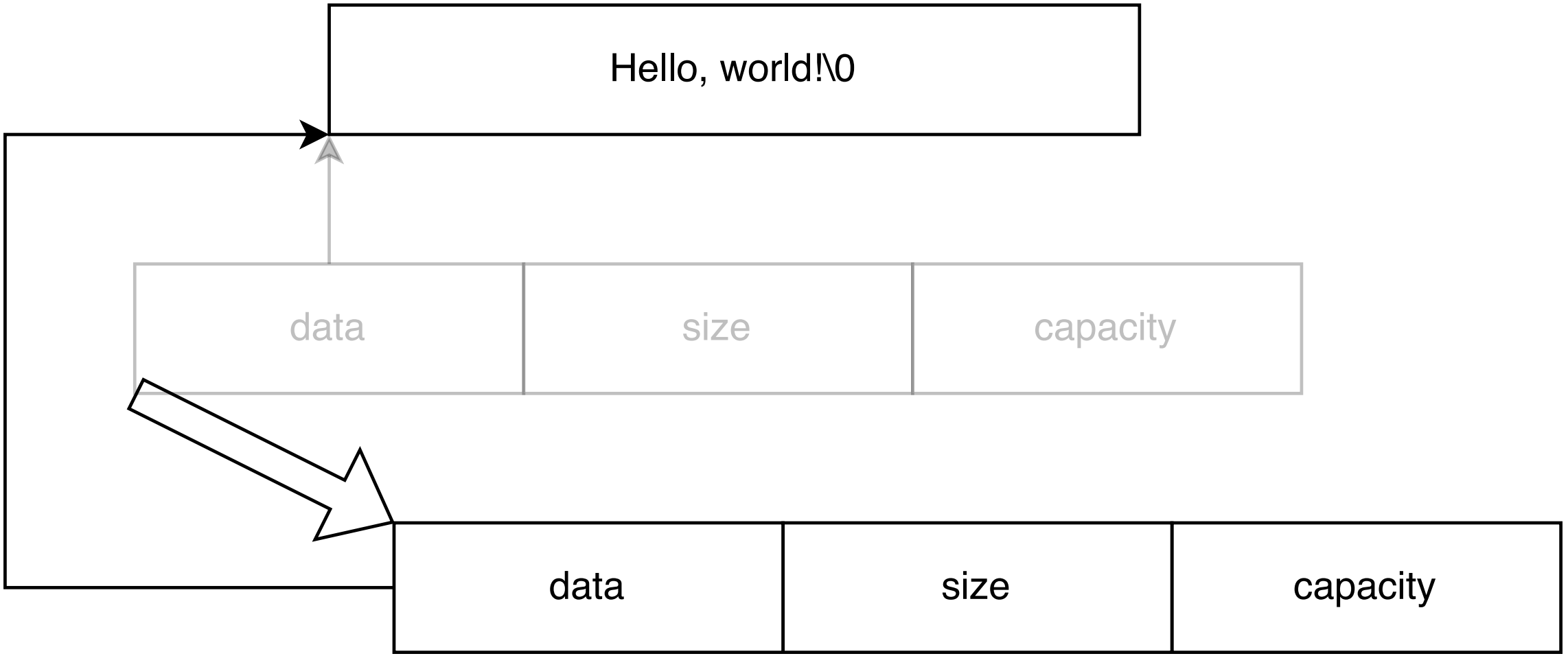
- All types are **trivially relocatable**
  - Relocated via bitwise copy

## C++

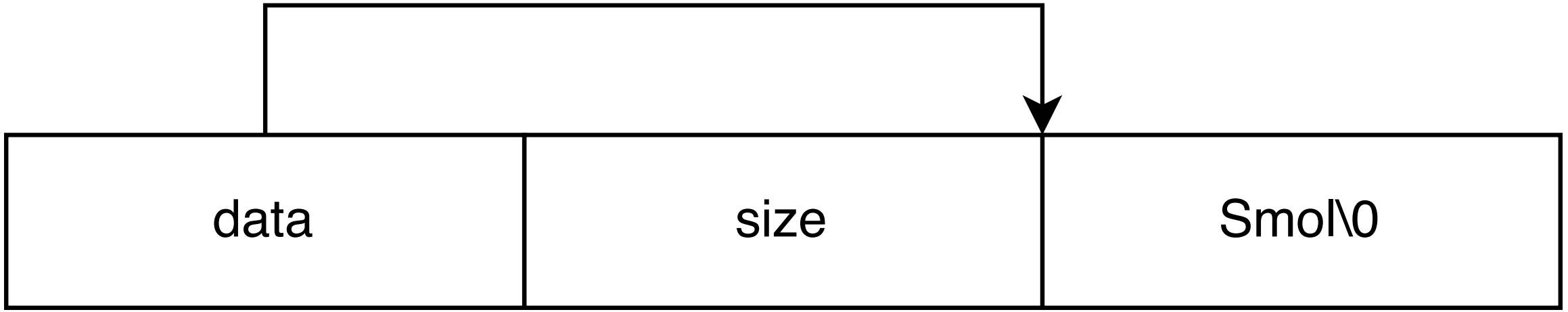
- Types may be **non-trivially relocatable**
  - Require running additional code to maintain invariants when relocated



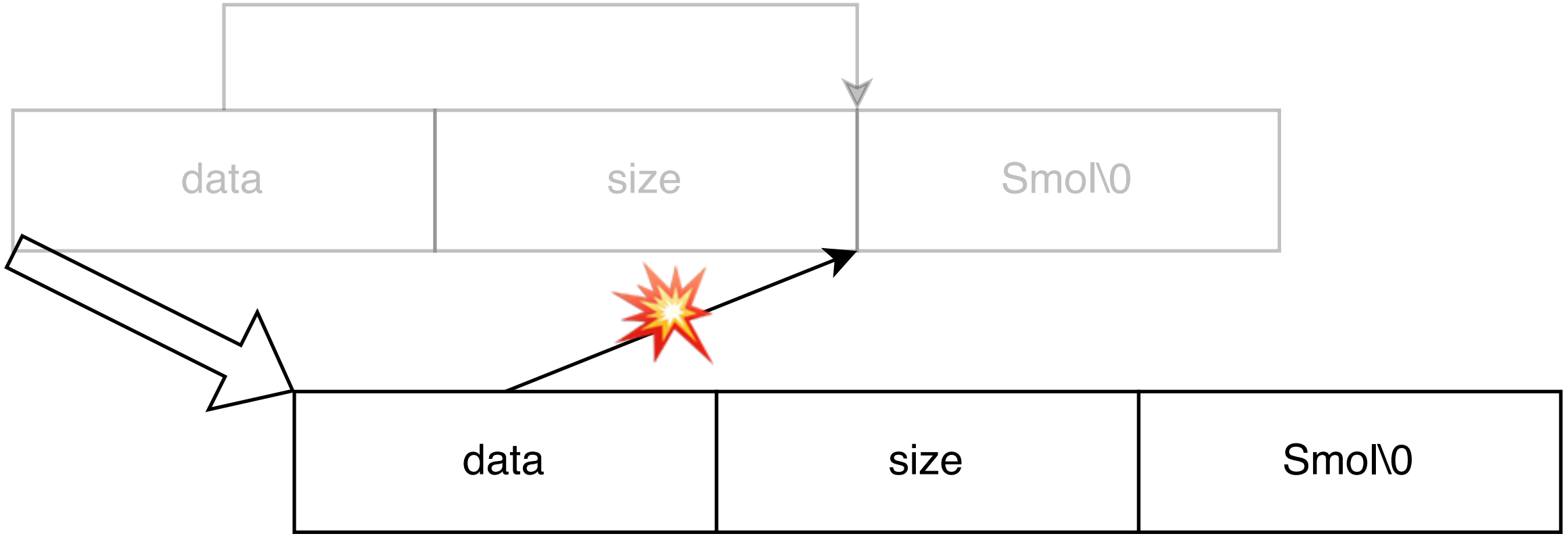

`std::string`




`std::string`




`std::string` with Small String Optimization (SSO)




`std::string` with Small String Optimization (SSO)

# Core Challenges ⚠️

1. Lack of stable ABI for C++
    - We need to interop via C ABI
2. C++ language features that we can't express in Rust
    - Particularly move semantics mismatch

# A Rust interop example 🎯

We want to add audio plugin hosting to our Rust application...

# The project layout 🗺️

```
Cargo.toml
src/
├── main.rs
cpp/
├── include/
│   └── lib.hpp
├── src/
│   └── lib.cpp
└── CMakeLists.txt
```

# Adding a C interface

`cpp/src/lib.cpp`

```cpp
#include <juce_audio_processors/juce_audio_processors.h>

extern "C" {
    juce::AudioPluginFormatManager* audio_plugin_format_manager_new() {
        return std::make_unique<juce::AudioPluginFormatManager>().release();
    }

    void audio_plugin_format_manager_delete(juce::AudioPluginFormatManager* mgr) {
        std::default_delete<juce::AudioPluginFormatManager>{}(mgr);
    }
}
```

# Declaring the C functions in Rust

`src/main.rs`

```rust
use std::ffi::c_void; // useful module w/ FFI types

unsafe extern "C" {
    fn audio_plugin_format_manager_new() -> *mut c_void;
    fn audio_plugin_format_manager_delete(ptr: *mut c_void);
}
```

⚠️ `unsafe` since we are responsible for ensuring the function declarations are correct!

`unsafe extern` is required since Rust 2024 edition, earlier versions didn't require them to be marked unsafe...

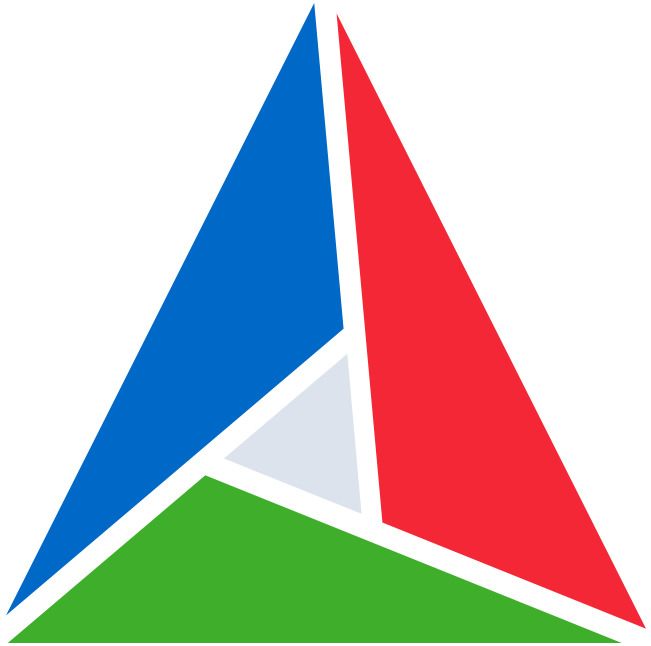
# Build Scripts 🛠️

- A Rust program that is compiled and executed before building your crate
- Writes to stdout to influence the build process ( `cargo:` prefix)
  - `println!("cargo:rustc-link-lib=static=example");`

# Build Dependencies 📦

`Cargo.toml`

```toml
[build-dependencies]
cmake = "0.1.54"
```



# Building and linking our C++ code 

`build.rs`

```rust
fn main() {
    let path = cmake::Config::new("cpp")
        .build_target("example")
        .build();

    println!("cargo:rustc-link-search=native={}/build", path.display());
    println!("cargo:rustc-link-lib=static=example");

    if cfg!(target_os = "macos") {
        println!("cargo:rustc-link-lib=c++");
        println!("cargo:rustc-link-lib=framework=CoreAudio");
        // ...
    }
}
```

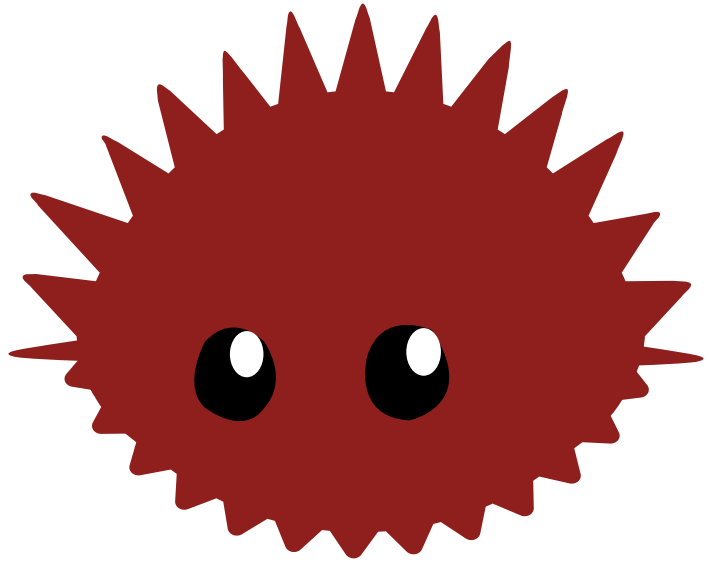
## We can now call our C++ function from Rust!

`src/main.rs`

```
fn main() {
    let mgr: *mut c_void = unsafe { audio_plugin_format_manager_new() };
    unsafe { audio_plugin_format_manager_delete(mgr) };
}
```

# Unsafe Rust

- There are limits to what the Rust compiler is able to check
- `unsafe` keyword
  - "With great power comes great responsibility" 🦸
- We can layer on a safe API to enforce correct usage



# Wrapping `unsafe` in a safe abstraction

`src/main.rs`

```
struct AudioPluginFormatManager {
    ptr: *mut c_void,
}

impl AudioPluginFormatManager {
    pub fn new() -> Self {
        AudioPluginFormatManager { ptr: unsafe { audio_plugin_format_manager_new() } }
    }
}

impl Drop for AudioPluginFormatManager {
    fn drop(&mut self) {
        unsafe { audio_plugin_format_manager_delete(self.ptr) }
    }
}
```

`src/main.rs`

```
fn main() {
    let mgr = AudioPluginFormatManager::new();
}
```

## Public Member Functions

**AudioPluginFormatManager** ()

**~AudioPluginFormatManager** ()
Destructor.

void **addDefaultFormats** ()
Adds the set of available standard formats, e.g.

int **getNumFormats** () const
Returns the number of types of format that are available.

**AudioPluginFormat** * **getFormat** (int index) const
Returns one of the available formats.

**Array**< **AudioPluginFormat** * > **getFormats** () const
Returns a list of all the registered formats.

void **addFormat** (**AudioPluginFormat** *)
Adds a format to the list.

std::unique_ptr< **AudioPluginInstance** > **createPluginInstance** (const **PluginDescription** &description, double initialSampleRate, int initialBufferSize, **String** &errorMessage) const
Tries to load the type for this description, by trying all the formats that this manager knows about.

# Automating the process 🛠️

There are various tools to help us generate Rust bindings for C++ code:

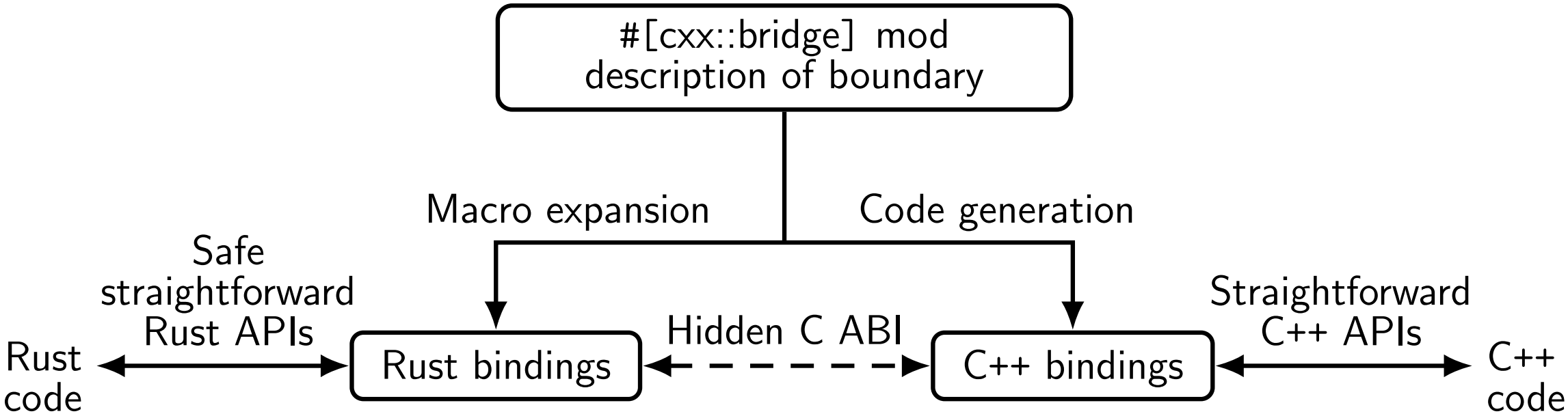
- binden
- crubit
- cxx 👈

# `cxx`

**Safe* bidirectional interop between Rust and C++**

- Authored by the prolific David Tolnay (dtolnay)
- "zero or negligible overhead"
- Provides bindings for common C++ types ( `vector` , `unique_ptr` , ...)
- Used by Meta and KDAB (Qt)

*The bindings are safe, C++ code is still unsafe


#[cxx::bridge] mod
description of boundary
Macro expansion
Code generation
Safe
straightforward
Rust APIs
Hidden C ABI
Straightforward
C++ APIs
Rust
code
Rust bindings
C++ bindings
C++
code

# Revisting our previous example with cxx

`Cargo.toml`

```
[dependencies]
cxx = "1.0.186"
```

`cpp/include/lib.hpp`

```cpp
#pragma once

#include <juce_audio_processors/juce_audio_processors.h>

std::unique_ptr<juce::AudioPluginFormatManager> make_audio_plugin_format_manager()
{
    return std::make_unique<juce::AudioPluginFormatManager>();
}
```

main.rs

```
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        include!("lib.hpp");

        #[namespace = "juce"]
        type AudioPluginFormatManager;

        fn make_audio_plugin_format_manager() -> UniquePtr<AudioPluginFormatManager>;
    }
}
```

# What does `cxx::bridge` generate?

`src/main.rs`

```
pub mod juce {
    #[repr(C)]
    pub struct AudioPluginFormatManager {
        _private: ::cxx::private::Opaque,
    }

    pub fn make_audio_plugin_format_manager() -> ::cxx::UniquePtr<AudioPluginFormatManager> {
        unsafe extern "C" {
            #[link_name = "cxxbridge1$make_audio_plugin_format_manager"]
            fn __make_audio_plugin_format_manager() -> *mut AudioPluginFormatManager;

        }
        unsafe { ::cxx::UniquePtr::from_raw(__make_audio_plugin_format_manager()) }
    }

    // ...
```

# cxx_build

Cargo.toml

```
[build-dependencies]
cxx-build = "1.0.186"
```

# cxx_build::bridge

build.rs

```
fn main() {
    let _ = cxx_build::bridge("src/main.rs");
    // ...
}
```

```
OUT_DIR
└── cxxbridge
    └── sources
        └── example
            └── src
                └── main.rs.cc
```

## What does `cxx_build` generate for us?

`OUT_DIR/cxxbridge/sources/example_cxx/src/main.rs.cc`

```
#include "lib.hpp" // hey, its our code! 👋

// ...

extern "C" {
    ::juce::AudioPluginFormatManager* cxxbridge1$make_audio_plugin_format_manager() noexcept {
        return ::make_audio_plugin_format_manager().release();
    }
}
```

## Back where we started...

`src/main.rs`

```
impl juce::AudioPluginFormatManager {
    pub fn new() -> UniquePtr<Self> {
        juce::make_audio_plugin_format_manager()
    }
}

fn main() {
    let mgr = juce::AudioPluginFormatManager::new();
}
```

## Adding bindings is now much easier 😌

```
AudioPluginFormat* AudioPluginFormatManager::getFormat (int index) const;
```

```
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        // ...

        #[namespace = "juce"]
        type AudioPluginFormat;

        #[cxx_name = "getFormat"]
        fn get_format(self: &AudioPluginFormatManager, index: i32) -> *mut AudioPluginFormat;
    }
}
```

## How can we bind to non-const methods?

```
void AudioPluginFormatManager::addDefaultFormats ();
```

```
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        // ...

        #[cxx_name = "getNumFormats"]
        fn get_num_formats(self: &AudioPluginFormatManager) -> i32;

        #[cxx_name = "addDefaultFormats"]
        fn add_default_formats(self: &mut AudioPluginFormatManager);
    }
}
```

```
error[cxxbridge]: mutable reference to opaque C++ type requires a pin
                  -- use `self: Pin<&mut AudioPluginFormatManager>`
    ┌─ src/main.rs:28:38
    │
28  │         fn add_default_formats(self: &mut AudioPluginFormatManager);
    │                                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    │
```

# `Pin<Ptr>` 📌

## A pointer which pins its pointee in place

For our purposes, `Pin<&mut T>` is a `&mut T` that stops us accidentally moving `T`.

src/main.rs

```rust
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        // ...

        #[cxx_name = "addDefaultFormats"]
        fn add_default_formats(self: Pin<&mut AudioPluginFormatManager>);
    }
}

fn main () {
    let mut mgr: UniquePtr<juce::AudioPluginFormatManager> = ...;

    let mgr: Pin<&mut juce::AudioPluginFormatManager> = mgr.pin_mut();
    mgr.add_default_formats();
}
```

# Layering on safety 🦺

The C++ interface may not be safe to use directly from Rust...

```
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        // ...

        #[cxx_name = "getFormat"]
        fn get_format(self: &AudioPluginFormatManager, index: i32) -> *mut AudioPluginFormat;
    }
}
```

## Raw pointers are not tracked by the borrow checker

```
let mgr: UniquePtr<juce::AudioPluginFormatManager> = ...;
let format: *mut juce::AudioPluginFormat = mgr.get_format(0)

drop(mgr);

let format = unsafe { &*format }; // 💥 use after free
```

## We can teach the borrow checker about the lifetimes involved

```
impl juce::AudioPluginFormatManager {
    fn get_format_ref<'a>(
        &'a self,
        index: i32
    ) -> Option<&'a juce::AudioPluginFormat> {
        let format_ptr: *mut juce::AudioPluginFormat = self.get_format(index);

        unsafe { format_ptr.as_ref() }
    }
}
```

We could elide the lifetime in this instance, but being explicit for the slide.

## Borrow checker will now reject this!

```
let mgr: UniquePtr<juce::AudioPluginFormatManager> = ...;
let format: Option<&juce::AudioPluginFormat> = mgr.get_format_ref(0)

drop(mgr);

let format: &juce::AudioPluginFormat = format.unwrap();
```

```
error[E0505]: cannot move out of `mgr` because it is borrowed
```

# Types with borrowed data

Two ways to construct a `juce::AudioBuffer<T>`:

```
// 1. Allocate its own memory internally
AudioBuffer (
    int numChannelsToAllocate,
    int numSamplesToAllocate
);

// 2. Pass in a existing buffer for it to refer to
AudioBuffer (
    T *const *dataToReferTo,
    int numChannelsToUse,
    int numSamples
);
```

`cpp/include/lib.hpp`

```
namespace juce {
    using AudioBufferFloat = juce::AudioBuffer<float>;
}
```

`src/main.rs`

```
#[cxx::bridge]
mod juce {
    #[namespace = "juce"]
    type AudioBufferFloat<'buffer>;
}
```

```
impl juce::AudioBufferFloat<'_> {
    fn new(
        num_channels: i32,
        num_samples: i32
    ) -> UniquePtr<juce::AudioBufferFloat<'static>> {
        // ...
    }

    fn from_buffer<'a>(
        buffer: &'a mut [f32],
    ) -> UniquePtr<juce::AudioBufferFloat<'a>> {
        // ...
    }
}
```

`from_buffer` takes a single channel for simplicity

# Passing C++ types around by value

`cxx` can be overly conservative in prohibiting passing C++ types by value...

```
namespace juce {
    Range<float> AudioBuffer<float>::findMinMax (...) const;
}
```

`cpp/include/lib.hpp`

```cpp
namespace juce {
    using AudioBufferFloat = juce::AudioBuffer<float>;
    using RangeFloat = juce::Range<float>;
}
```

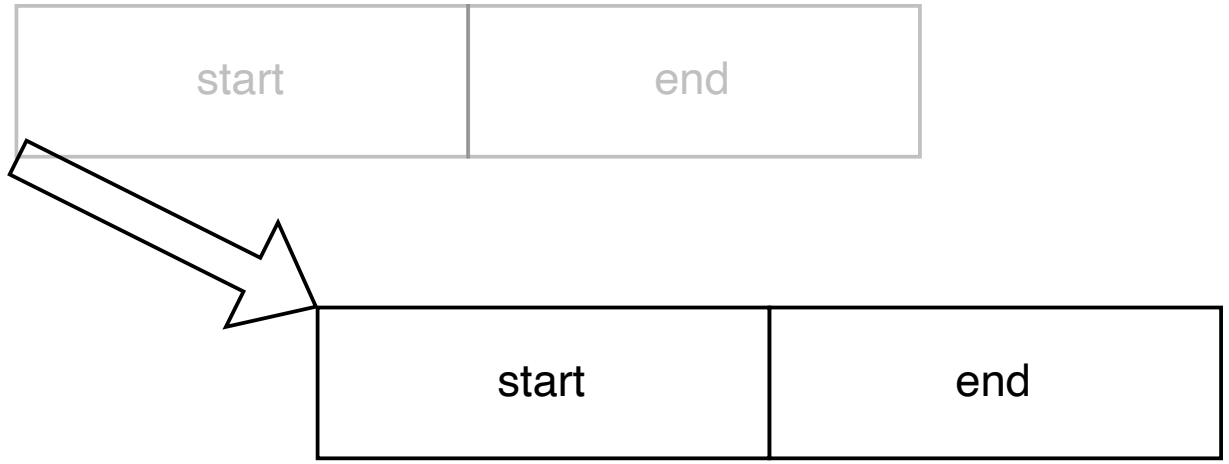
`src/main.rs` (in our `cxx::bridge`)

```rust
#[namespace = "juce"]
type AudioBufferFloat;

#[namespace = "juce"]
type RangeFloat;

#[cxx_name = "findMinMax"]
fn find_min_max(self: &AudioBufferFloat, ...) -> RangeFloat;
```

```
error[cxxbridge]: returning opaque C++ type by value is not supported
    ┌─ src/main.rs:110:96
    │
110 │             fn find_min_max(self: &AudioBufferFloat, ...) -> RangeFloat;
    │                                                              ^^^^^^^^^^^
    │
    = hint: wrap it in a UniquePtr<> or SharedPtr<>
```

# juce::Range<T>

```
template <typename T>
class Range {
public:
    constexpr Range (const Range&)
        = default;

    // ...
private:
    T start{}, end{};
};
```



`main.rs`

```rust
#[repr(C)]
pub struct RangeFloat {
    _space: MaybeUninit<[f32; 2]>,
}

unsafe impl cxx::ExternType for RangeFloat {
    type Id = cxx::type_id!("juce::RangeFloat");
    type Kind = cxx::kind::Trivial; // default is kind::Opaque
}
```

`cpp/src/lib.cpp`

```cpp
static_assert (sizeof (juce::RangeFloat) == 8);
static_assert (alignof (juce::RangeFloat) == 4);
```

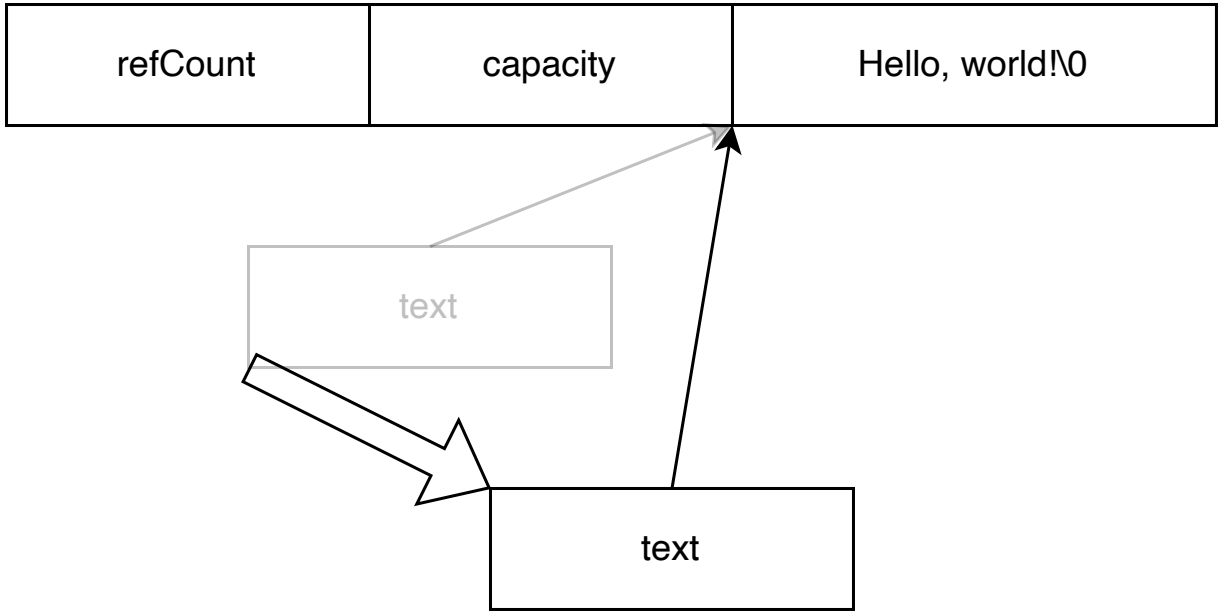
## `juce::Range<float>` by-value in Rust! 🥳

```
fn process_audio (buffer: Pin<&mut juce::AudioBufferFloat>) {
    let range: juce::RangeFloat = buffer.find_min_max(0, 0, buffer.getNumSamples());

    let min = range.get_start();
    let max = range.get_end();

    // ...
}
```

We might start to wonder where else we can use this... 👀

## Is `juce::String` trivially relocatable? 🤔

```
class String
{
public:
    String (String&& other) noexcept
        : text (other.text)
    {
        other.text = emptyString.text;
    }

    // ...
private:
    CharPointerType text;

    // ...
};
```



`main.rs`

```rust
#[repr(C)]
pub struct JuceString {
    _space: MaybeUninit<*mut c_void>,
}

unsafe impl cxx::ExternType for JuceString {
    type Id = cxx::type_id!("juce::String");
    type Kind = cxx::kind::Trivial;
}
```

`cpp/src/lib.cpp`

```cpp
static_assert (sizeof (juce::String) == sizeof (void*));
static_assert (alignof (juce::String) == alignof (void*));
```

## We are now responsible for cleaning up 🗑️

`main.rs`

```rust
#[cxx::bridge]
pub mod juce {
    unsafe extern "C++" {
        // ...

        #[cxx_name = "drop"]
        fn drop_string(s: &mut JuceString);
    }
}

impl Drop for JuceString {
    fn drop(&mut self) {
        juce::drop_string(self)
    }
}
```

`cpp/include/lib.hpp`

```cpp
template <typename T>
void drop (T& value) {
    value.~T();
}
```

## `cxx` will check our claim that `juce::String` is trivially relocatable

```
static_assert(rust::IsRelocatable<juce::String>::value);
```

We need to provide a specialization for our type:

```
template <>
struct rust::IsRelocatable<juce::String> : std::true_type {};
```

## `juce::String` by-value in Rust! 🥳

`main.rs`

```
#[namespace = "juce"]
type SystemStats;

#[Self = "SystemStats"]
#[cxx_name = "getJUCEVersion"]
fn get_juce_version() -> JuceString;
```

```
let version = juce::SystemStats::get_juce_version();
```

# One slight issue... 💥

- *Technically* this is undefined behaviour in the language ☝️🤓
- According to the C++ object model we can't just `memcpy` objects and continue their lifetimes elsewhere!
- The exception is *TriviallyCopyable* types are safe to `memcpy`
  - e.g. `juce::Range<float>` is *TriviallyCopyable* ✅
  - e.g. `juce::String` is not *TriviallyCopyable* ⚠️

# "UB that works in practice" 😬

- Major compilers won't break code that does this
- Many popular C++ libraries exploit this for optimisation purposes
  - BSL (Bloomberg), Qt, Folly (Facebook), Abseil (Google), EASTL...
  - e.g. a common optimisation for vector-like containers

# Language support coming in C++26? 🤞

wg21.link/p2786

## Trivial Relocatability For C++26

**Proposal to safely relocate objects in memory**


| | |
|---|---|
| Document #: | P2786R13 |
| Date: | 2025-02-14 15:01 CET |
| Project: | Programming Language C++ |
| Audience: | CWG, LWG |
| Reply-to: | Alisdair Meredith <ameredith1@bloomberg.net><br>Mungo Gill <mgill83@bloomberg.net><br>Joshua Berne <jberne4@bloomberg.net><br>Corentin Jabot <corentinjabot@gmail.com><br>Pablo Halpern <phalpern@halpernwightsoftware.com><br>Lori Hughes <lori@lorihughes.com> |

# What about non-trivially relocatable types?

```
template <typename Type>
class AudioBuffer {
{
    // ...
    Type** channels = nullptr; // 👈
    HeapBlock<char, true> allocatedData;
    Type* preallocatedChannelSpace[32];
}
```

# Stack pinning 📌

`cxx` provides a macro that can safely create a C++ string on the stack:

```
cxx::let_cxx_string!(foo = "Hello, world!");
```

🦀 🪄

```
let foo: Pin<&mut cxx::CxxString> = ...;
```

## `pin-init` crate 

```
stack_pin_init!(let buffer = AudioBufferFloat::new(2, 512));
```

# We have options to avoid indirection!

- ✅ *TriviallyRelocatable* C++ types can be passed by value
  - ⚠️ "UB that works in practice"...
- 📌 Otherwise we can use stack pinning

# Our toolkit for Rust/C++ interop 🧰

We've covered:

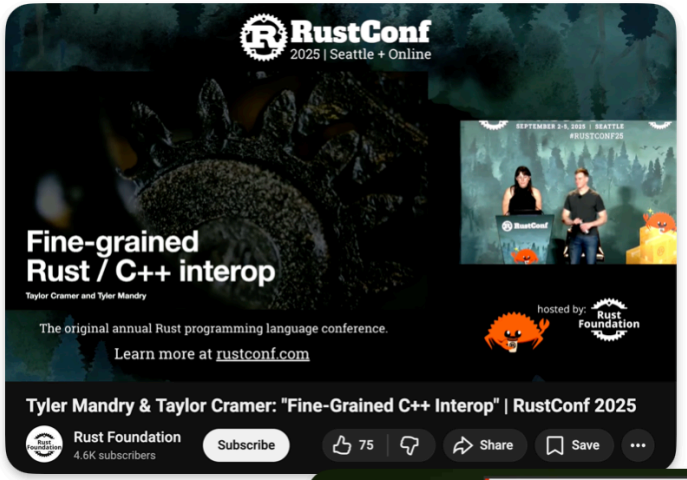
- DIY bindings via C FFI
- Building and linking our C++ code with build scripts
- Using `cxx` to automate binding generation
- Layering on safe, idiomatic Rust abstractions
- Removing indirection to reclaim performance and ergonomics

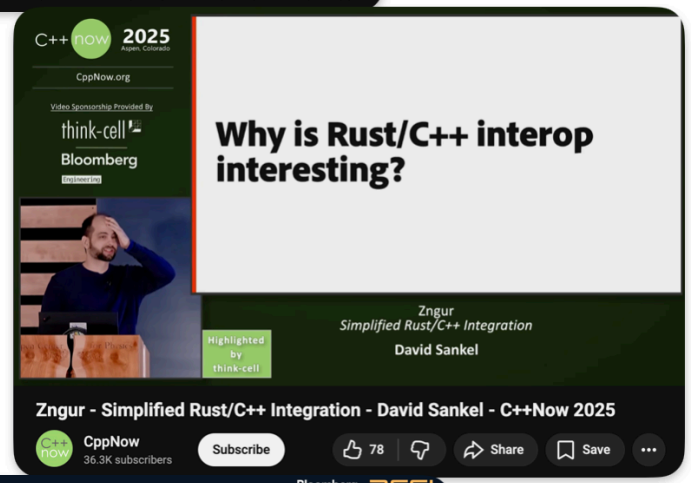
# Where can Rust/C++ interop work well?

- ✅ Relatively small, stable API
- ✅ Limited API vocabulary (basic types and language features)
  - ❌ Heavy use of templates, inheritance…
- ⚠️ Ability to wrap the API if necessary

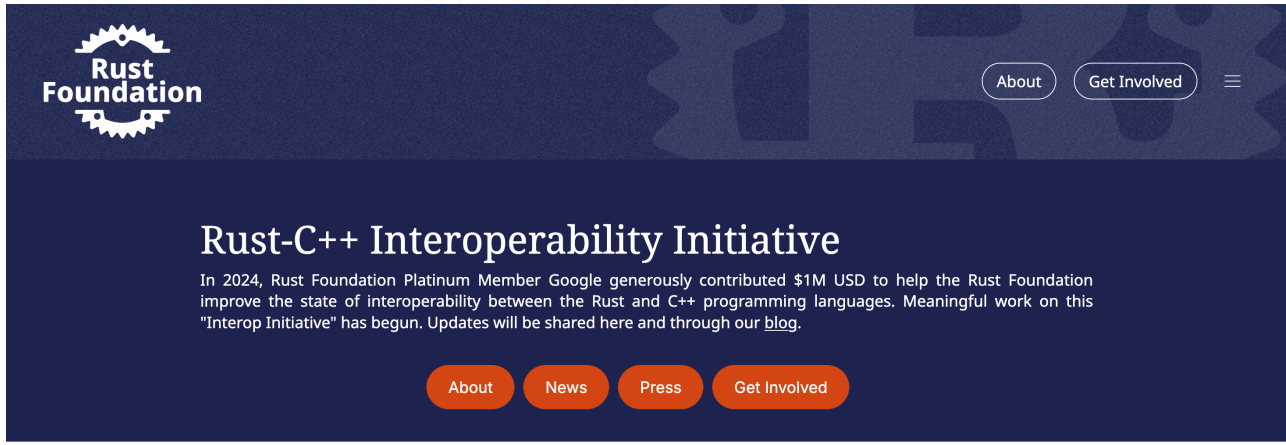
# Growing interest in Rust/C++ interop

- Major C++ organizations are investing in Rust interop
- Evolving ecosystem of tools:
  - cxx
  - crubit
  - zngur




# Future of Rust/C++ interop 🔮




*“Despite the growing popularity and adoption of Rust, it would be unrealistic to expect even the most technically advanced organization to easily pivot to Rust and away from the architecture of existing codebases. The Rust Foundation wants to help address this need in a harmonious and methodical way.”*

# Rust ❤️ C++

Thanks for listening! 🙏