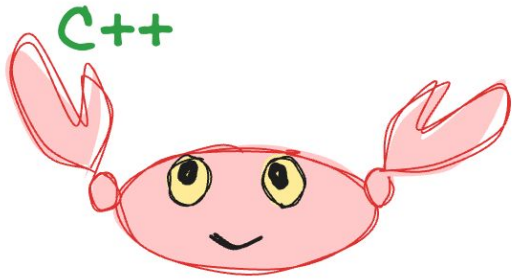
Rust / C++ Workshop

Aida Getoeva



Contents

Contents

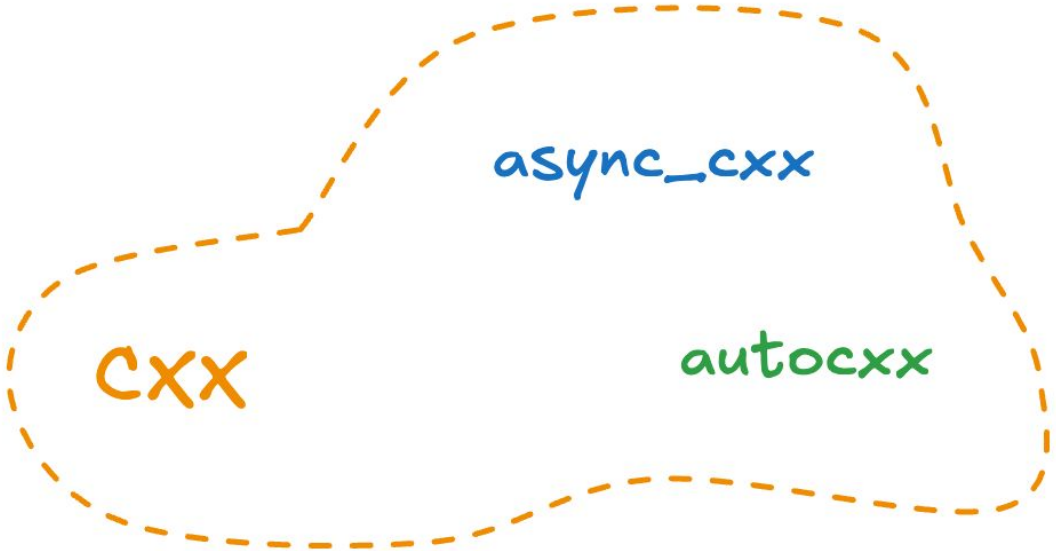
- Chapter 4: Synchronous interop
  - Opaque types, shared and transparent types
  - Functions and member functions
  - Constant vs mutable references
  - Built-in types and smart pointers
- Chapter 5: Asynchronous interop
  - Introducing C++ folly
  - What’s different about async?
  - Bridge or tunnel?

Chapter 1: Why C++

- Extra responsibilities
  - Writing your own library means taking all the responsibilities for maintaining and updating said library
- New code, new bugs
  - Designing a new solution from scratch may solve some existing problems but bring many new ones
- Limited resources
  - Complete rewrite can potentially improve maintainability and reduce the tech-debt, but it will also bring negative short-term impact

Rust ❤️ C++

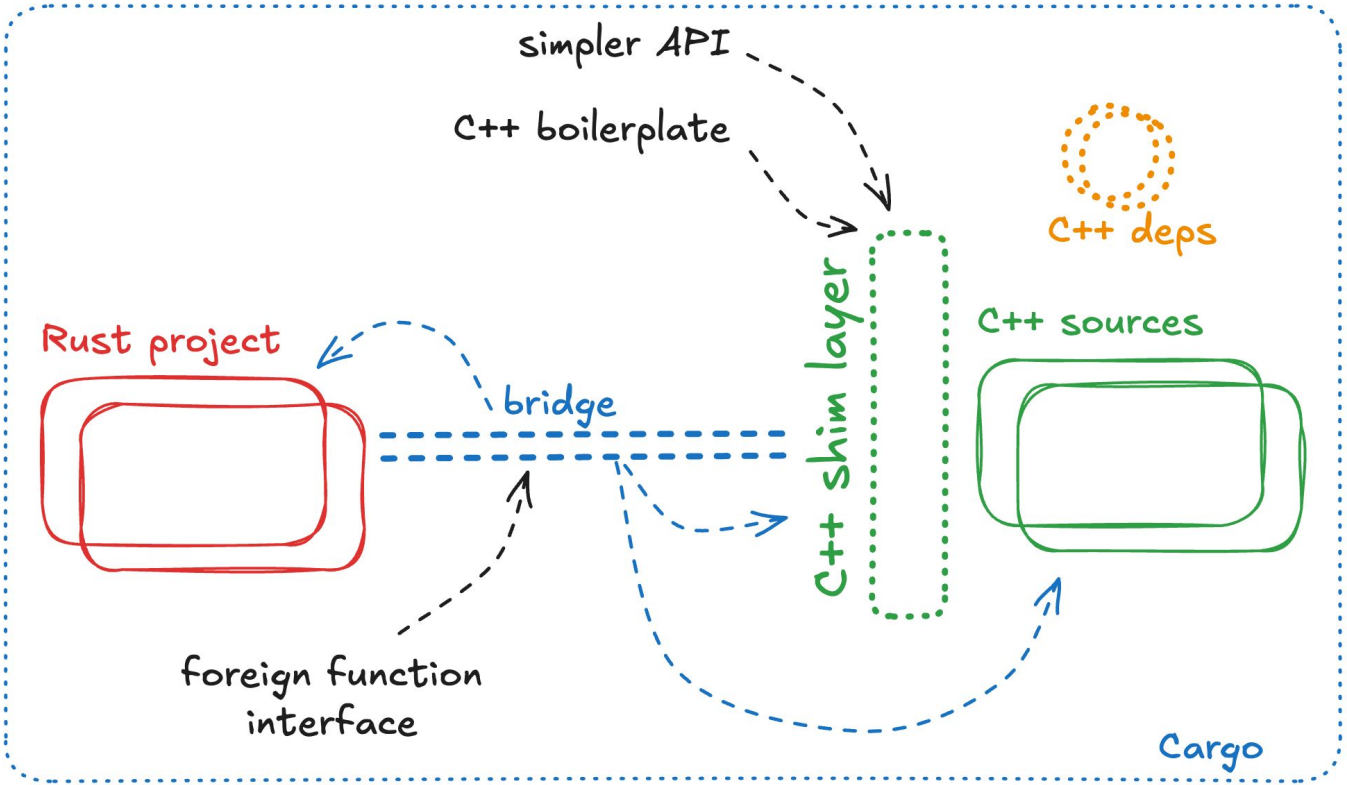
bindgen



Rust ❤️ C++

- Bindgen
  - Seamless conversion between primitive types and raw pointers
  - Unsafe memory handling and lack of implicit destructors
  - Generated *extern “C”* functions are unsafe
  - Handles huge amount of code on best-effort policy
- CXX
  - Seamless translation between vectors, strings and smart pointers
  - A two-way bridge between C++ and Rust
  - Safe external bindings
  - No silent errors, no surprises
  - Smart pointers!

Chapter 2: The landscape



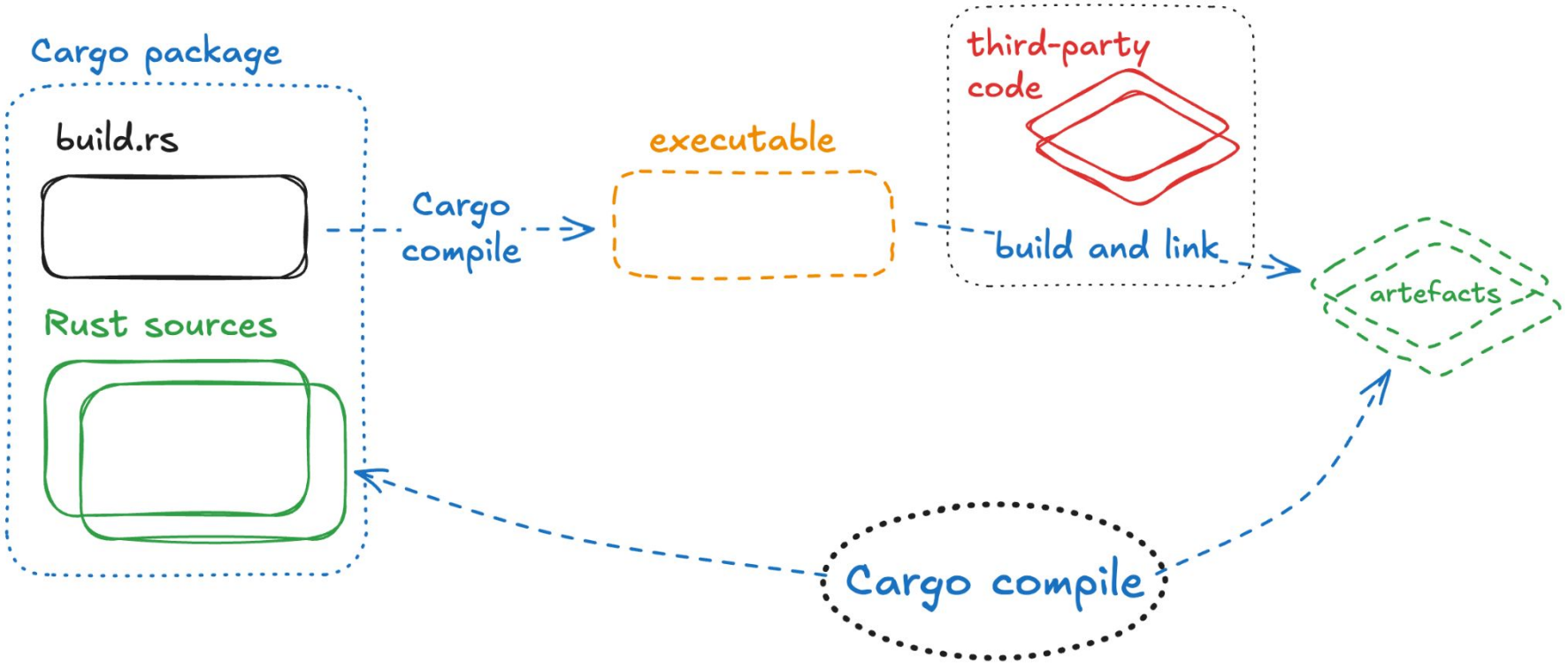
Build scripts

- Cargo allows for Rust packages to link third-party C++ libraries or built them from the source.
- Cargo build script – build.rs – lives in the root of the projects and defines the necessary configuration to compile or link external code.

```
// Example custom build script.
fn main() {
    // Tell Cargo that if the given file changes, to rerun this build script.
    println!("cargo::rerun-if-changed=src/hello.c");
    // Use the `cc` crate to build a C file and statically link it.
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
}
```

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

Lifecycle of the build script



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

Introducing `cxx_build`

- The CXX code generator for constructing and compiling C++ code
- Is used as a part of Cargo build script `build.rs`

```rust
fn main() {
    cxx_build::bridge("src/main.rs")
        .file("src/indexshim.cpp")
        .include("/opt/homebrew/include")
        .std("c++17")
        .compile("rust_cxx");

    println!("cargo:rustc-link-search=native=/opt/homebrew/lib");
    println!("cargo:rustc-link-lib=static=folly");
    println!("cargo:rerun-if-changed=include/index.h");
}
```

https://docs.rs/cxx-build/latest/cxx_build/

Chapter 3: Playground

- GitHub:
  - `https://github.com/kris1319/rust_cxx_workshop`
- In terminal:
  - `$ git clone https://github.com/kris1319/rust_cxx_workshop.git`
- Build and run:
  - `../rust_cxx_workshop$ cargo run`

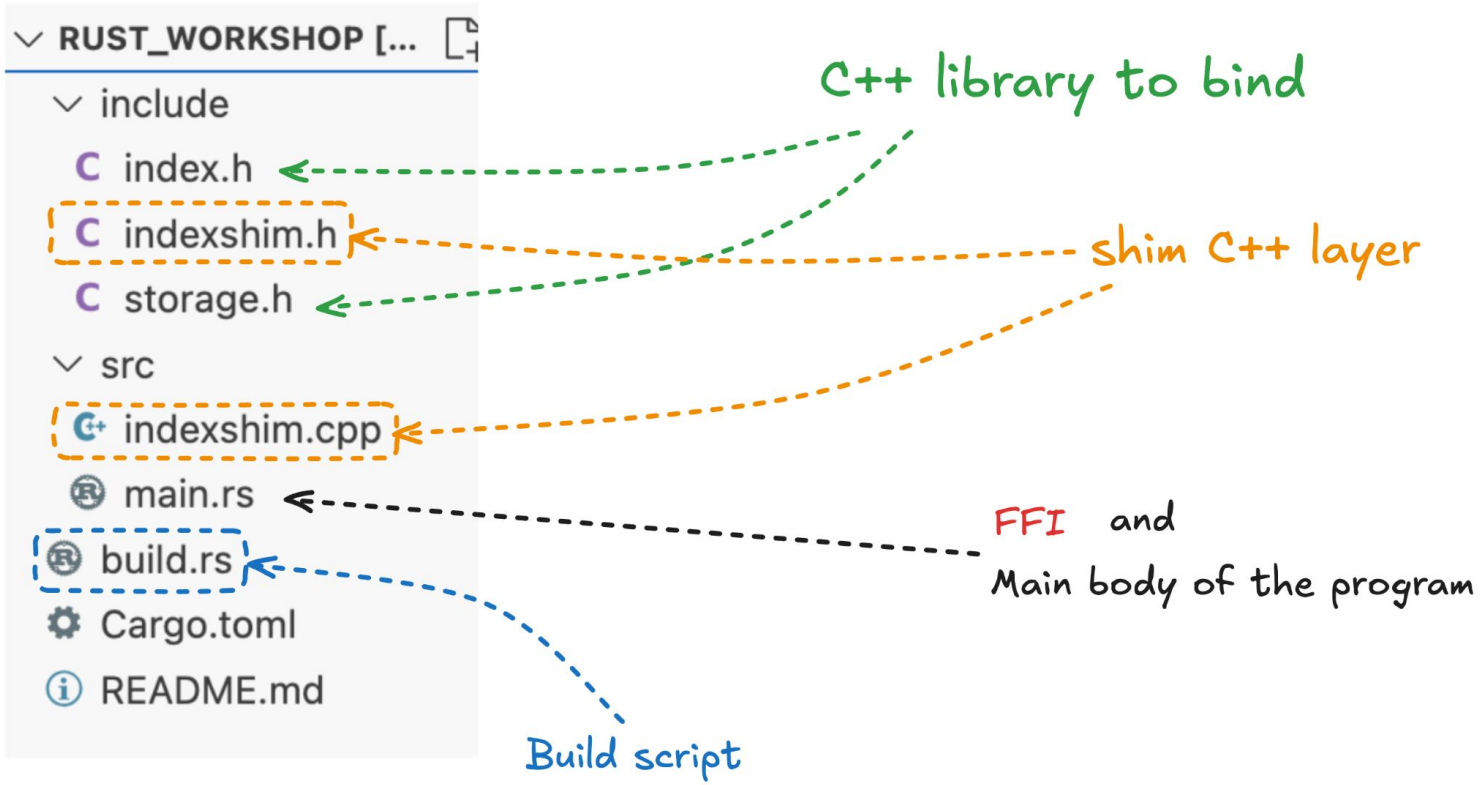
Folly setup: Unix / macOS

- Folly:
  - **`https://github.com/facebook/folly`**
- Homebrew formulae:
  - **`$ brew install folly`**
- Build and install Folly with all the dependencies:
  - **`$ git clone https://github.com/facebook/folly`**
  - **`../folly$ ./build.sh`**
- Update path to the dependencies in `build.rs`:
  - **`/usr/lib/`** or **`/usr/local/lib/`**

Playground

- Build and run:
  - `rust_cxx_workshop$ cargo run`

Playground


RUST_WORKSHOP [...
include
index.h
indexshim.h
storage.h
src
indexshim.cpp
main.rs
build.rs
Cargo.toml
README.md
C++ library to bind
shim C++ layer
FFI and
Main body of the program
Build script

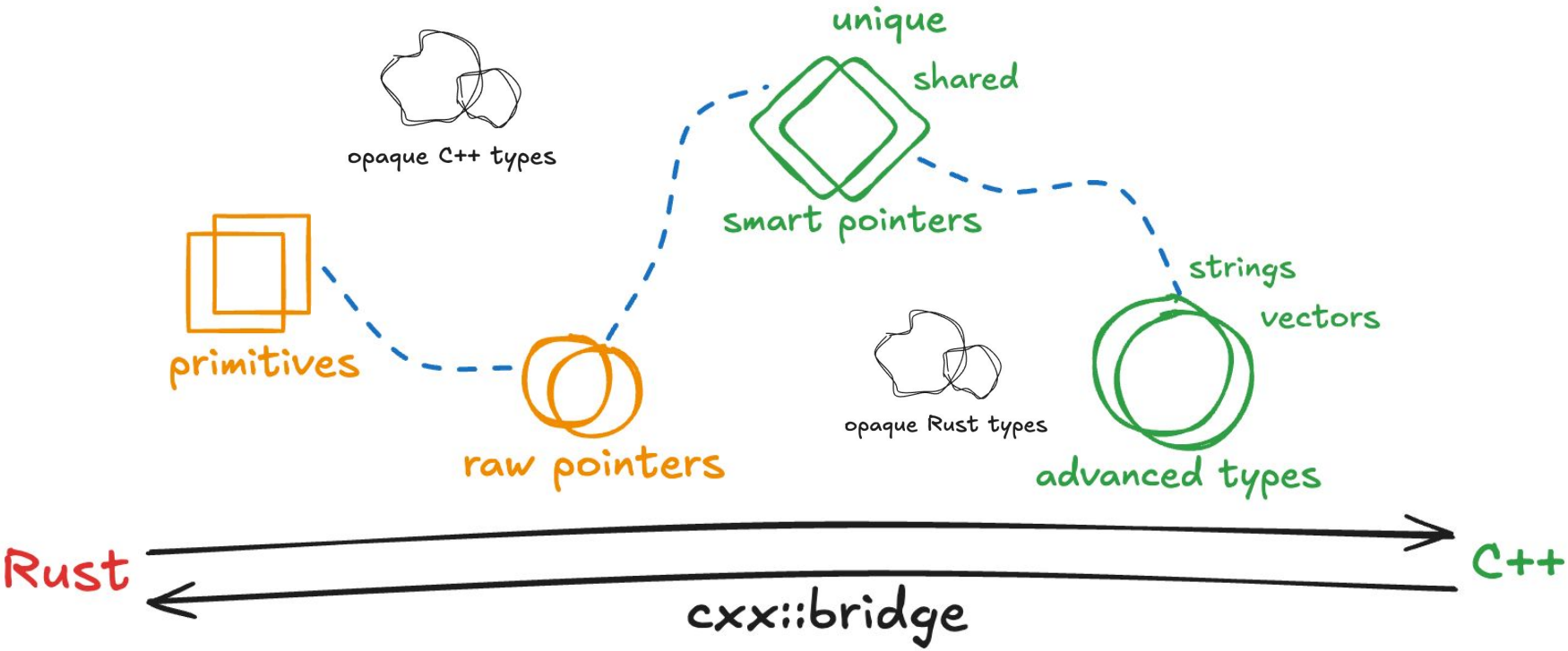
Chapter 4: Synchronous interop

Playground: Index

`index.h`: **`Index`** – extract words with positions from the given text

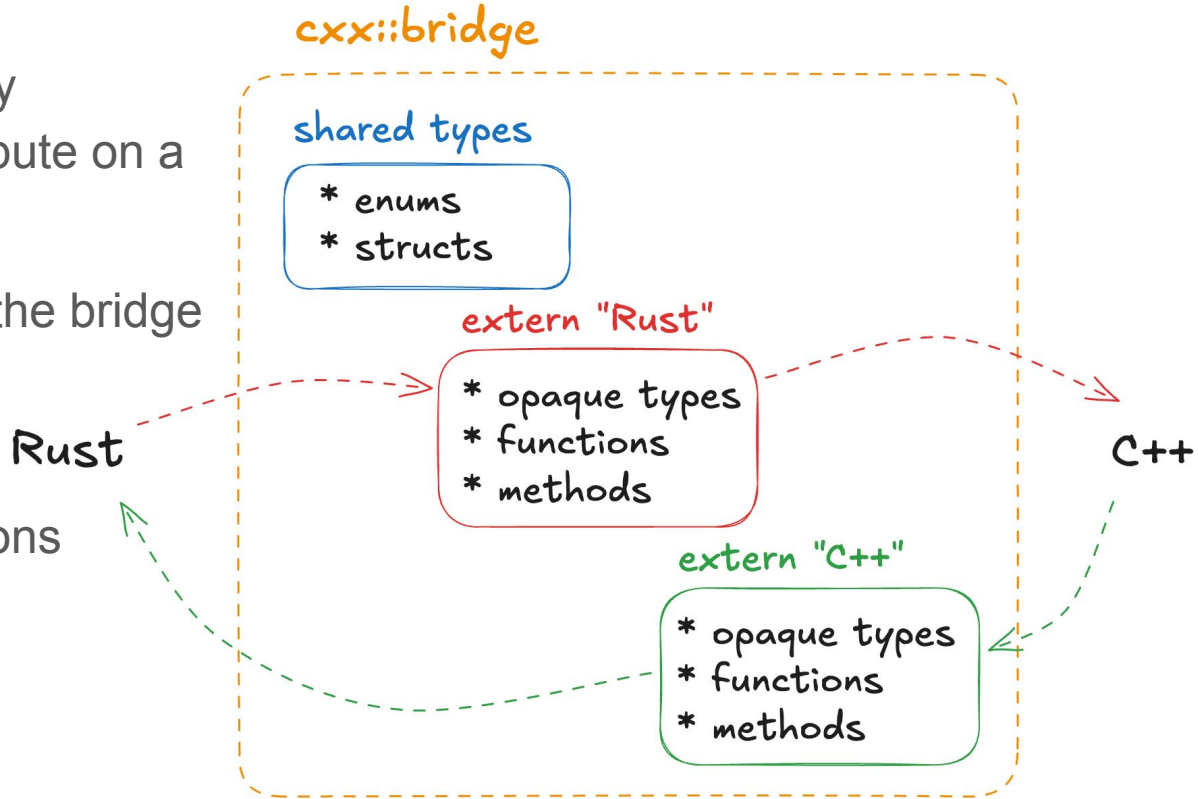
```
class Index {
public:
    using WordIndex = std::unordered_map<std::string, std::vector<size_t>>;

    Index(IndexType type): _type(type) {}

    int dictionary_size() const;
    IndexType index_type() const;

    void index(const std::string& name, const std::string& text);

    WordIndex search(const std::string& word) const;

private:
    IndexType _type;
    std::unordered_map<std::string, WordIndex> _index;
};
```

The world of `cxx`



Step 0: The `cxx` bridge

- CXX bridge is defined by `#[cxx::bridge]` attribute on a module
- All types defined within the bridge module are **shared**
- Declarations for the foreign types and functions live in the **extern** blocks



Step 0: The `cxx` bridge

```
#[cxx::bridge]
mod ffi {
    enum FruitFlavour {
        Apple,
        Pear,
    }

    unsafe extern "C++" {
        type MyCppFruit;

        #[Self = "MyCppFruit"]
        fn member();
        fn create_my_fruit(flavour: FruitFlavour) -> UniquePtr<MyCppFruit>;
    }
}

fn main() {
    let obj = ffi::create_my_fruit();
    ...
}
```

Step 1: Opaque types

- **Opaque types** – C++ types that are made available to Rust **only** behind an **indirection**
- Rust is not aware of the size or structure of the underlying object
- Using key-word `type` allows to declare the C++ type in the bridge
- Types of **indirection**:

Step 1: Opaque types

- **Opaque types** – C++ types that are made available to Rust **only** behind an **indirection**
- Rust is not aware of the size or structure of the underlying object
- Using key-word `type` allows to declare the C++ type in the bridge
- Types of **indirection**:
  - Smart pointers: `UniquePtr<MyCppType>` `SharedPtr<MyCppType>`

Step 1: Opaque types

- **Opaque types** – C++ types that are made available to Rust **only** behind an **indirection**
- Rust is not aware of the size or structure of the underlying object
- Using key-word `type` allows to declare the C++ type in the bridge
- Types of **indirection**:
  - Smart pointers: `UniquePtr<MyCppType>` `SharedPtr<MyCppType>`
  - Immutable reference: `&MyCppType`

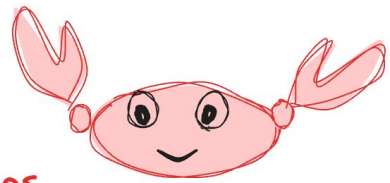
Step 1: Opaque types

- **Opaque types** – C++ types that are made available to Rust **only** behind an **indirection**
- Rust is not aware of the size or structure of the underlying object
- Using key-word `type` allows to declare the C++ type in the bridge
- Types of **indirection**:
  - Smart pointers: `UniquePtr<MyCppType>` `SharedPtr<MyCppType>`
  - Immutable reference: `&MyCppType`
  - Pinned mutable reference: `Pin<&mut MyCppType>`

Exercise 1: Exposing Index to Rust

- Add the `Index` declaration to the cxx bridge
- Use `include!` macro to include build artefacts, results of the build script

```rust
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("workshop/include/mycppfruit.h");
        type MyCppFruit;
    }
}
```



Step 2.0: Extern C++ functions

- **Functions** are declared in the bridge extern block the same way the opaque types are
- Functions can consist of any primitive types, shared types or CXX bindings
- CXX performs static assertions on the declared and C++ signatures

```
unsafe extern "C++" {
    fn create_my_fruit(number: i32) -> UniquePtr<MyCppFruit>;
}
```

```
std::unique_ptr<MyCppFruit> create_my_fruit(int number);
```

Step 2.1: Extern block safety

If an **extern** block contains at least one **safe**-to-call signature it should be marked as `unsafe extern` block.

- `unsafe` indicates that unchecked safety claim about the block contents
- Functions within unsafe block are **safe** to call

```
unsafe extern "C++" {
    fn safe_to_call();
}
```

```
extern "C++" {
    unsafe fn unsafe_to_call();
}
```

Step 2.2: Static member functions

- **Static member** functions are functions
- Attribute `#[Self = "MyType"]` forces CXX to interpret function as a native static member function of the specified type

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        type MyCppFruit;

        #[Self = "MyCppFruit"]
        fn static_method();
    }
}
```

```
fn main() {
    ffi::MyCppFruit::static_method();
    ...
}
```

Exercise 2: Creating Index

- Can we use C++ constructor?
- Is a type constructor some kind of a static member function?..

Exercise 2: Creating Index

- Can we use C++ constructor?
- Is a type constructor some kind of a static member function?..

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("workshop/include/index.h");
        type Index;

        #[Self = "Index"]
        fn Index() -> Index;
    }
}
```

Exercise 2: Creating Index

- Can we use C++ constructor?
- Is a type constructor some kind of a static member function?..

```rust
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("workshop/include/index.h");
        type Index;

        #[Self = "Index"]
        fn Index() -> Index;
    }
}
```

ERROR!

Exercise 2: Creating Index

- Can we use C++ constructor?
- Is a type constructor some kind of a static member function?..

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("workshop/include/index.h");
        type Index;

        #[Self = "Index"]
        fn Index() -> Index;
    }
}
```

We need a layer of indirection!

Step 3: Smart pointers

`CXX` provides seamless bindings between Rust and C++ smart pointers

- `CXX` defines both shared and unique pointers: **`cxx::UniquePtr<_>`**, **`cxx::SharedPtr<_>`**
- No need to manually implement conversions and operate on raw pointers

Exercise 3: Creating Index

- Let’s introduce a shim layer to the Index creation
- Should return a smart pointer
    - A new Index static method function
    - A separate builder function in the shim layer

Exercise 3: Creating Index

- Let’s introduce a shim layer to the Index creation
- Should return a smart pointer

```
unsafe extern "C++" {
    include!("workshop/include/indexshim.h");

    type Index;
    fn new_index() -> SharedPtr<Index>;
}
```

```
std::shared_ptr<Index> new_index() {
  return std::make_shared<Index>(IndexType::HashMap);
}
```

Step 4: Enums

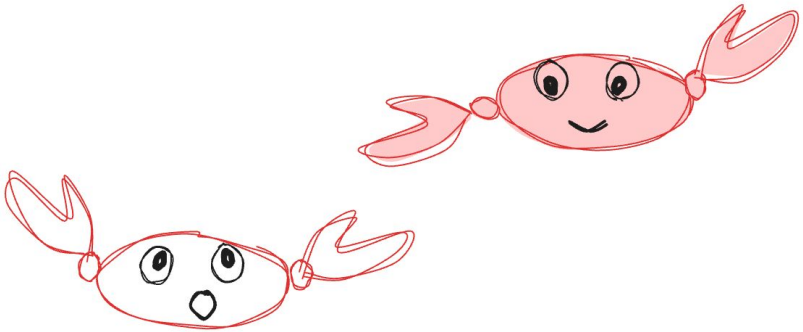
- **Enum** can be defined as an opaque or shared type across CXX bridge
- Shared enum compiles into C++ **enum class** with sufficient integral base type

```rust
#[cxx::bridge]
mod ffi {
    #[repr(i32)]
    enum FruitFlavour {
        Apple,
        Pear,
    }
}
```

```cpp
enum class FruitFlavour : int32_t {
  Apple = 0,
  Pear = 1,
};
```

Step 4: Enums

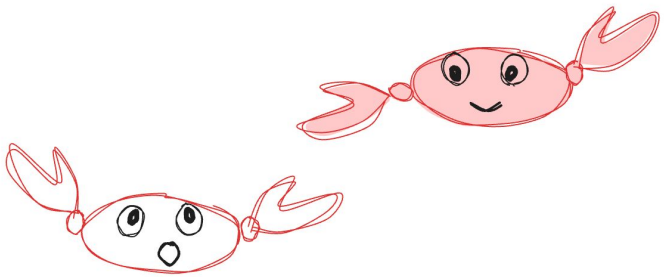
- Extern enums allow to define **transparent** C++/Rust enum type in the bridge
- CXX generates static assertions for the variants and discriminant values
- Rust integer representation correctly matches the C++ enum definition
- CXX allows to specify Copy, Clone, Debug attributes for shared types



Step 4: Enums

- Defining enum as a shared type **and** declaring it in the extern C++ block creates a **transparent enum** type.

```
#[cxx::bridge]
mod ffi {
    #[repr(i32)]
    enum FruitFlavour {
        Apple,
        Pear,
    }

    extern "C++" {
        include!("workshop/include/mycppfruit.h");
        type FruitFlavour;
    }
}
```



Exercise 4: Parameterised Index creation

- Original Index constructor takes an enum as an argument. Let's
  - Expose the `IndexType` enum
  - And parametrise the earlier defined `Index` creation function

```
#[cxx::bridge]
mod ffi {
    #[repr(i32)]
    enum FruitFlavour {
        Apple,
        Pear,
    }

    extern "C++" {
        include!("workshop/include/mycppfruit.h");
        type FruitFlavour;
    }
}
```

Exercise 4: Parameterised Index creation

```
#[cxx::bridge]
mod ffi {
    #[repr(i32)]
    #[derive(Clone, Copy, Debug)]
    enum IndexType {
        Tree,
        HashMap,
    }

    unsafe extern "C++" {
        include!("workshop/include/index.h");
        type IndexType;
    }

    unsafe extern "C++" {
        include!("workshop/include/indexshim.h");
        type Index;

        fn new_index(type_: IndexType) -> SharedPtr<Index>;
    }
}
```

Step 5: Const member functions

- CXX bridge recognises C++ member functions via **`&self`** argument
- All `&self` parameterized functions within an extern C++ block are compiled into methods of the opaque type associated with the block
- If there are more than one type per extern block, the type of `self` should be explicitly specified

```
unsafe extern "C++" {
    type Fruit;

    fn get_flavour(&self) -> FruitFlavour
}
```

```
fn main() {
    let fruit = ffi::create_my_fruit();
    fruit.get_flavour();
    ...
}
```

Exercise 5: Exposing const Index methods

- Add following bindings to the bridge:
  - `index_type()` – Type of the current index?
  - `dictionary_size()` – How many words are indexed?
- Add print-lines to the `main()` and run the code

```
fn main() {
    let index = ffi::new_index(ffi::IndexType::HashMap);
    println!("Index size: {:?}", index.dictionary_size());
    println!("Index type: {:?}", index.index_type());
}
```

Exercise 5: Exposing const Index methods

```
unsafe extern "C++" {
    include!("workshop/include/indexshim.h");
    type Index;

    fn new_index(type_: IndexType) -> SharedPtr<Index>;

    fn dictionary_size(&self) -> i32;
    fn index_type(&self) -> IndexType;
}
```

Step 6: Non-const member functions

- CXX **does not** allow mutable references `&mut self` of the opaque types
  - Explicit ban on memory swapping for objects with unknown size and structure
- Mutation support in the bridge requires memory pinning:
  `Pin<&mut OpaqueType>`
- Both smart pointers provide the required API:
  - `cxx::UniquePtr<OpaqueType>::pin_mut()`
  - `cxx::SharedPtr<OpaqueType>::pin_mut_unchecked()`

Exercise 6: Exposing Index indexer

Expose method:

```
void Index::index(const std::string& name, const std::string& text)
```

Important:

- Both smart pointers provide memory pinning:
  - `pin_mut()`, `pin_mut_unchecked()`
- For the seamless translation of the C++ standard strings:
  - `cxx::CxxString` as a type for `const std::string &`
  - `let_cxx_string!(var_name = "text");`

Exercise 6: Exposing Index indexer

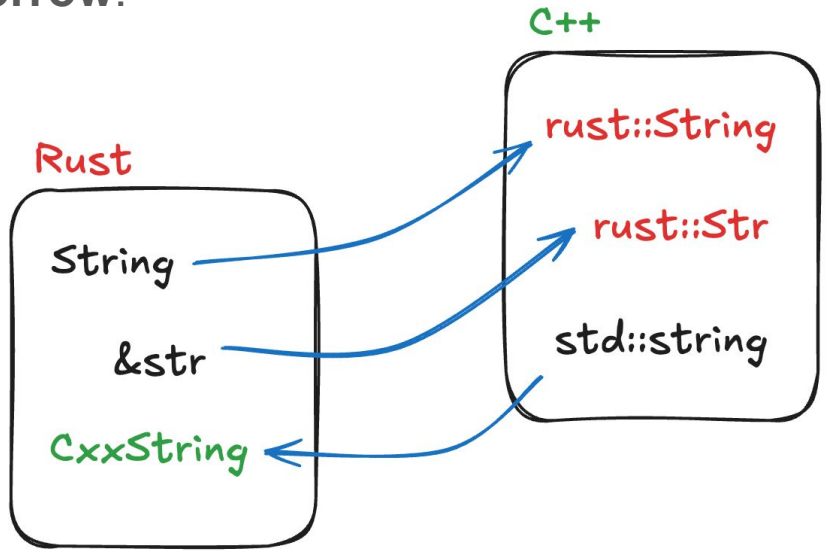
```
unsafe extern "C++" {
    include!("workshop/include/indexshim.h");
    type Index;
    ...
    fn index(self: Pin<&mut Index>, name: &CxxString, text: &CxxString);
}
```

Don't forget to run and test!

```
fn main() {
    let index = ffi::new_index(ffi::IndexType::HashMap);
    println!("Index size: {:?}", index.dictionary_size());

    let_cxx_string!(name = "doc1");
    let_cxx_string!(text = "This is a test document with 10 words to index.");
    index.pin_mut().index(&name, &text);
    println!("Index size after indexing: {:?}", index.dictionary_size());
}
```

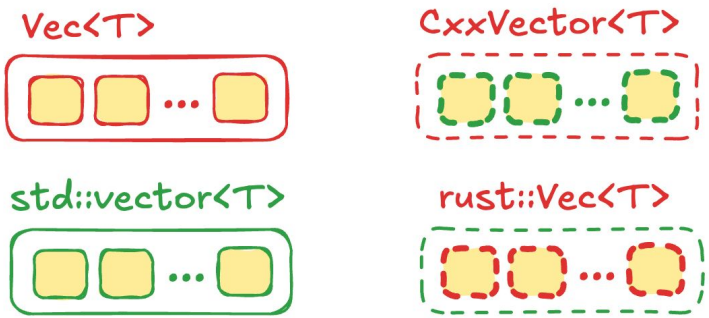
Step 7.1: String types

- `String` – standard Rust string mapped into `rust::String`, **no restrictions**
- `&str` – mapped into `rust::Str`. It is a **borrow**: watch the lifetime!
- `cxx::CxxString` – C++ `std::string`, **cannot** be returned by **value**



Step 7.2: Vector types

- `Vec<T>` – Rust vector of T, where T **cannot** be an opaque C++ type, maps to `rust::Vec<T>`
- `cxx::CxxVector<T>` – C++ `std::vector<T>`, where T **cannot** be an opaque Rust type. `cxx::CxxVector` cannot be passed by value
- Both bindings allow for conversion between the vector types



Step 7.3: Shared types

Shared types – types defined in the `cxx` bridge and visible to **both** Rust and C++

- Defined as native Rust structs
- Can consist of **primitives**, **opaque types** behind indirection and `cxx` **bindings** like smart pointers, strings or vectors

Step 7.3: Shared types

Shared types – types defined in the `cxx` bridge and visible to **both** Rust and C++

- Defined as native Rust structs
- Can consist of **primitives**, **opaque types** behind indirection and `cxx` **bindings** like smart pointers, strings or vectors

```rust
#[cxx::bridge]
mod ffi {
    #[derive(Clone, Debug)]
    pub struct FruitBasket {
        name: String,
        fruits: Vec<FruitFlavour>,
    }
}
```

Exercise 7: Exposing Index search

- Create a shared type that will hold the results of search operation on Index
  - A list of document names that hold the word
  - A list of all document names with the positions for the word in them
- Define the shim-functions that perform the search and convert the results into Rust vector and the shared type defined above
- Expose those to Rust in the `cxx` bridge

```
fn search_index(index: &Index, word: &str) -> Vec<String>;

fn search_index_with_positions(index: &Index, word: &str) -> Vec<IndexResult>;
```

Exercise 7: Exposing Index search

```
rust::Vec<IndexResult> search_index_with_positions(const Index& index, rust::Str word) {
    auto results = index.search((std::string)word);
    rust::Vec<IndexResult> rust_results;
    for (const auto& [doc_name, positions] : results) {
        rust::Vec<unsigned> rust_positions;
        for (size_t pos : positions) {
            rust_positions.push_back(static_cast<unsigned>(pos));
        }
        rust_results.push_back(IndexResult{doc_name, std::move(rust_positions)});
    }

    return rust_results;
}
```

```
#[derive(Clone, Debug)]
pub struct IndexResult {
    name: String,
    positions: Vec<u32>,
}
```

Exercise 7: Exposing Index search

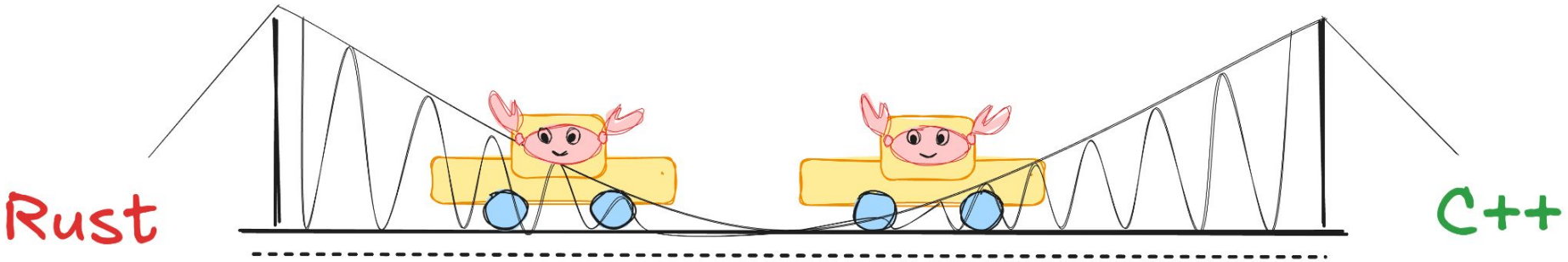
```
rust::Vec<IndexResult> search_index_with_positions(const Index& index, rust::Str word) {
    auto results = index.search((std::string)word);
    rust::Vec<IndexResult> rust_results;
    for (const auto& [doc_name, positions] : results) {
        rust::Vec<unsigned> rust_positions;
        for (size_t pos : positions) {
            rust_positions.push_back(static_cast<unsigned>(pos));
        }
        rust_results.push_back(IndexResult{doc_name, std::move(rust_positions)});
    }

    return rust_results;
}
```

Exercise 7: Exposing Index search

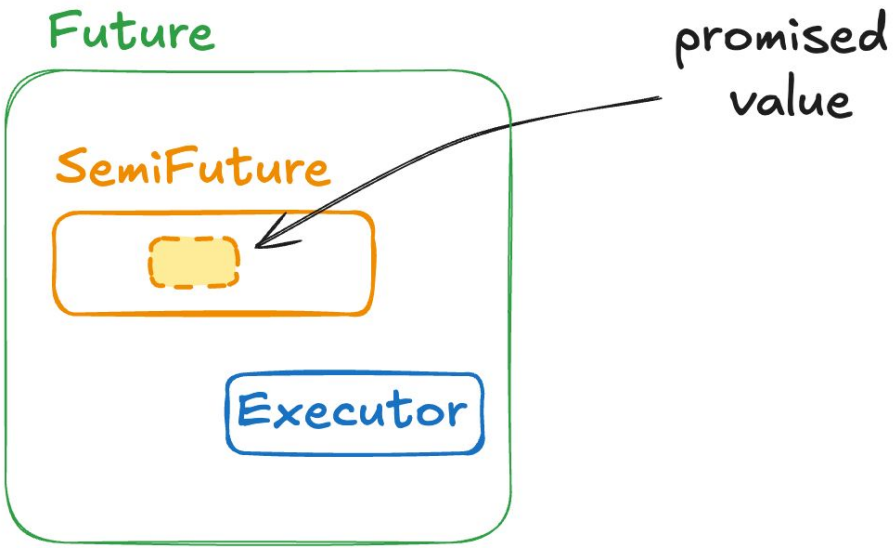
```
rust::Vec<IndexResult> search_index_with_positions(const Index& index, rust::Str word) {
    auto results = index.search((std::string)word);
    rust::Vec<IndexResult> rust_results;
    for (const auto& [doc_name, positions] : results) {
        rust::Vec<unsigned> rust_positions;
        for (size_t pos : positions) {
            rust_positions.push_back(static_cast<unsigned>(pos));
        }
        rust_results.push_back(IndexResult{doc_name, std::move(rust_positions)});
    }

    return rust_results;
}
```

```
Hello, world!
Index size: 0
Index type: HashMap
Index size after indexing: 10
Index size after 2 indexing: 14
Search results for 'test': ["doc1", "doc2"]
Document: doc1, Positions: [10]
Document: doc2, Positions: [17]
```

Chapter 5: Asynchronous interop

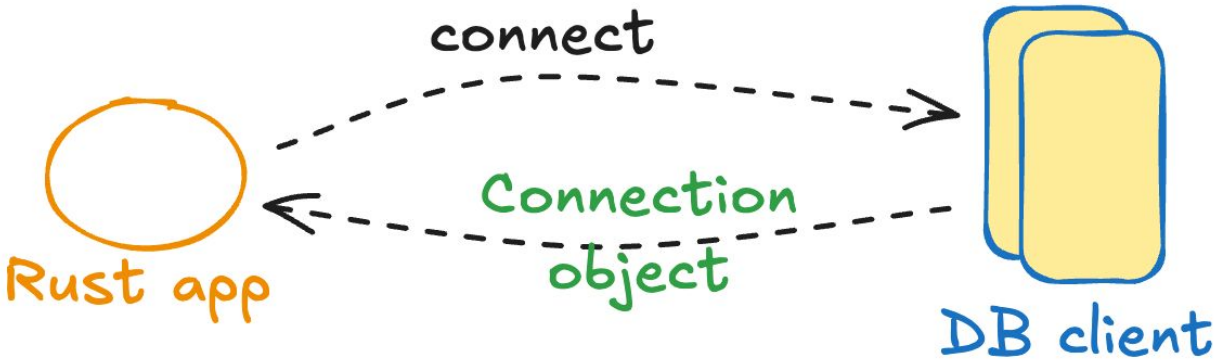


Simplified world of Folly async



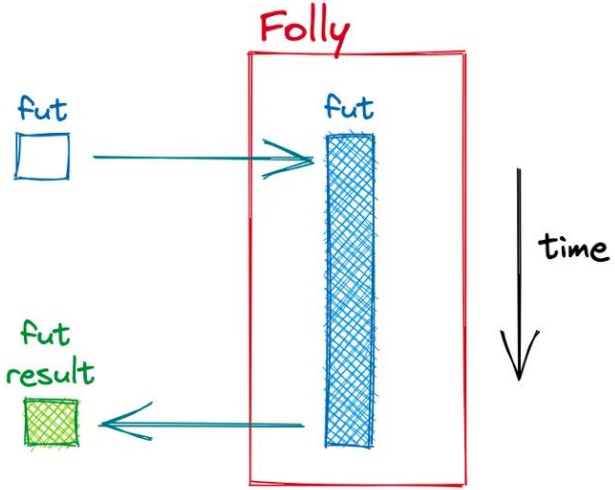
Example: DB client

```
class DBClient {
public:
    // asynchronously connects to the DB
    folly::SemiFuture<std::unique_ptr<Connection>> connect();
};
```



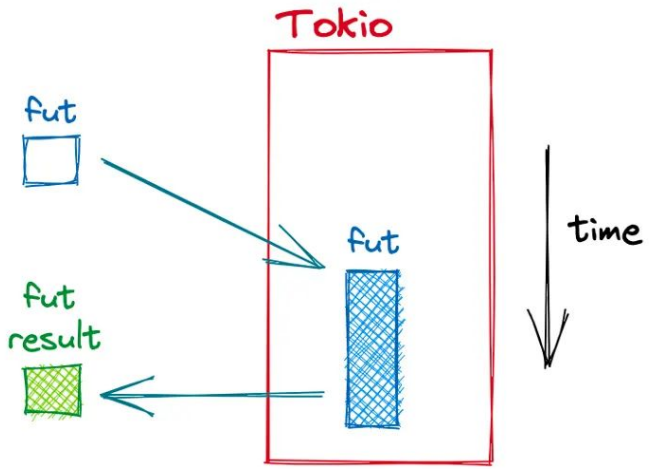
Simplified world of Folly async

```
{
    auto fut = connect(client, options)
        .via(getExecutor())
        .thenTry(...);
    ...
}
```

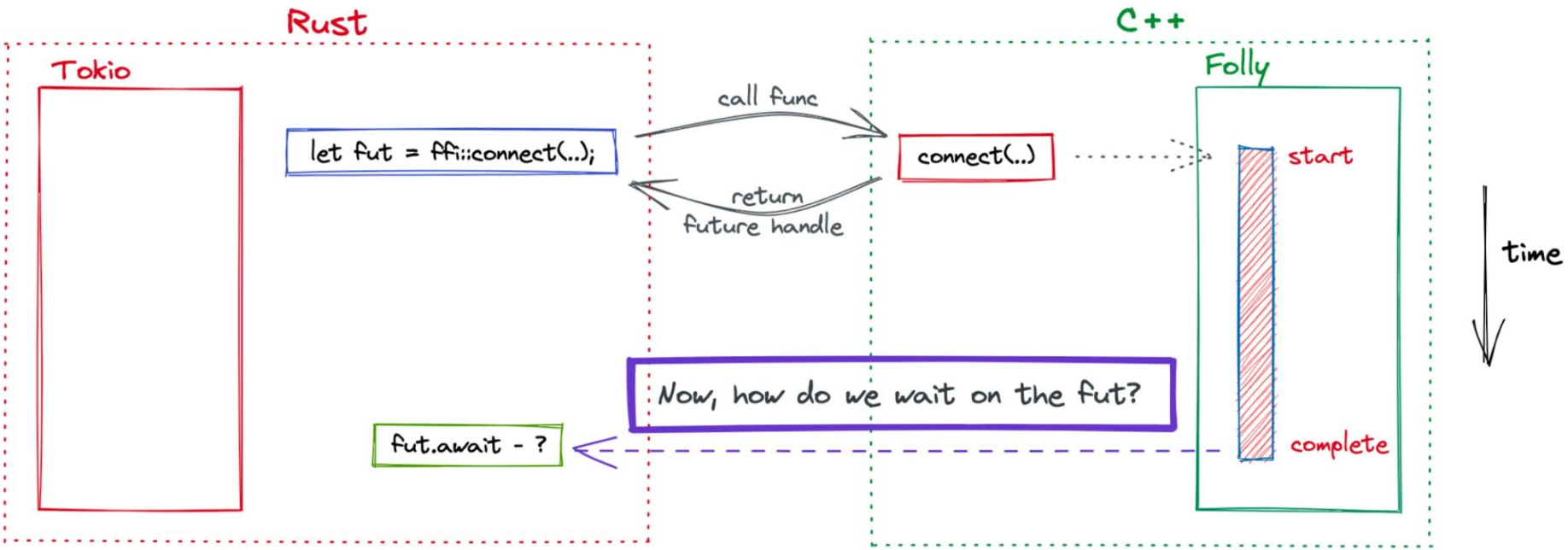


Async Tokio Rust

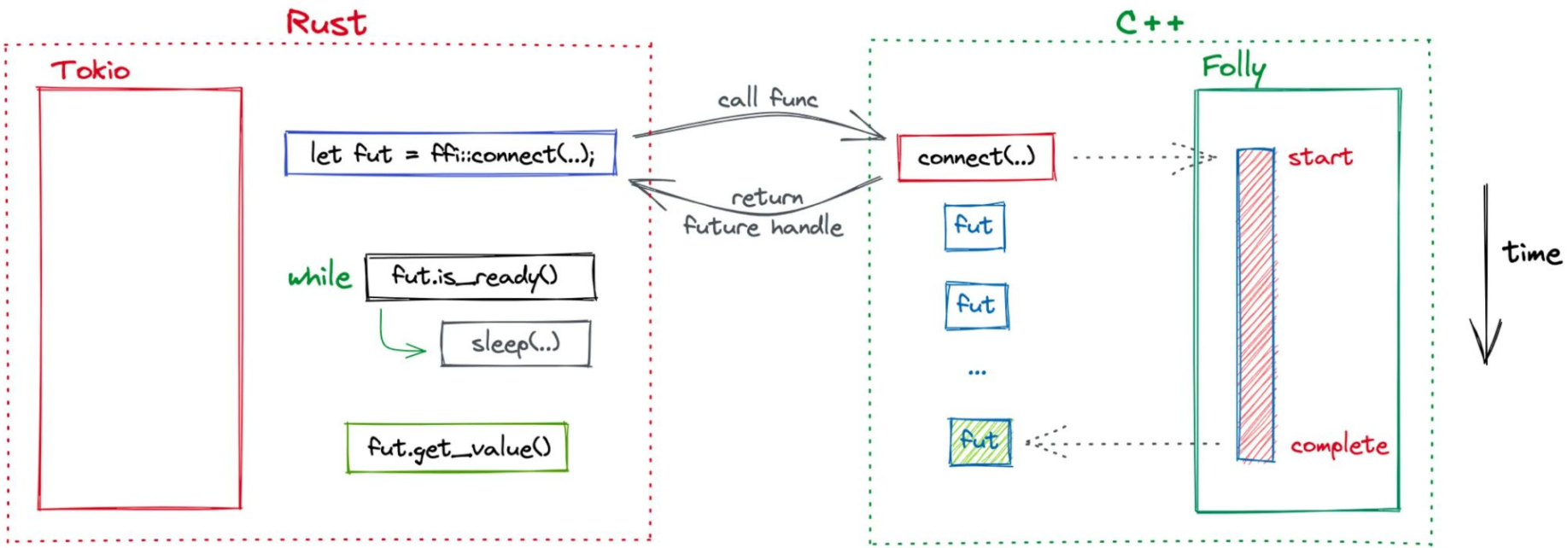
```
{
    let fut = connect(&client, &options);
    ...
    fut.await?;
}
```



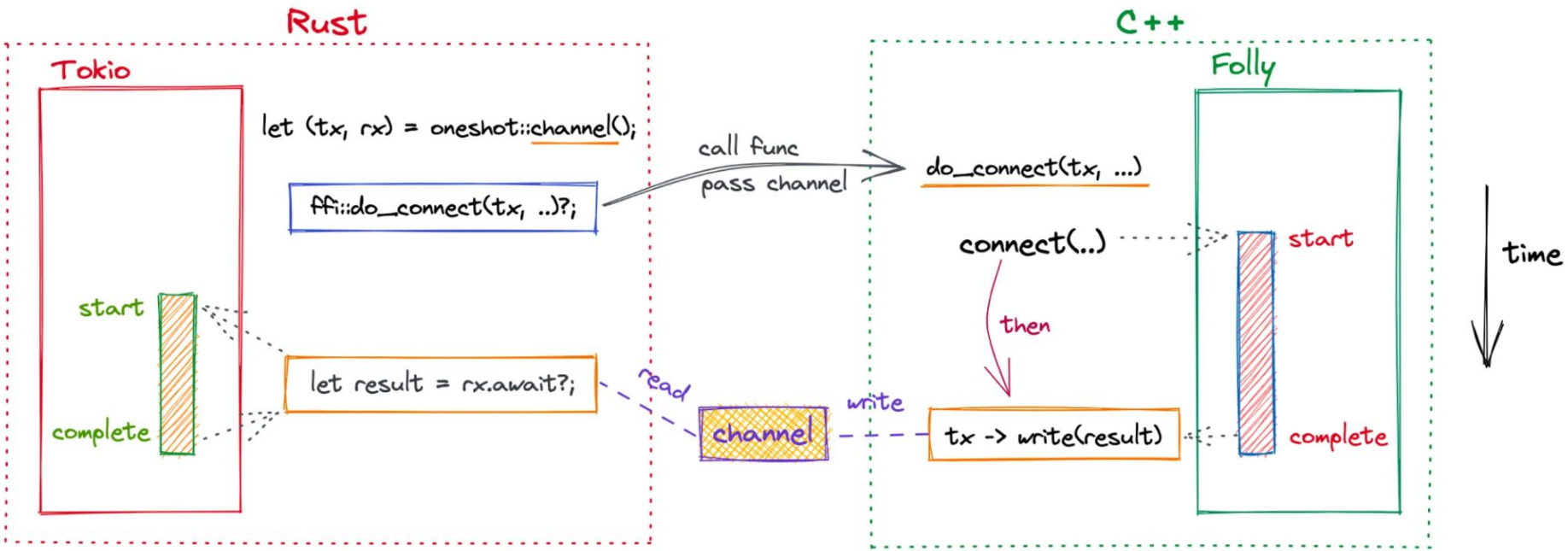
Building async bridge



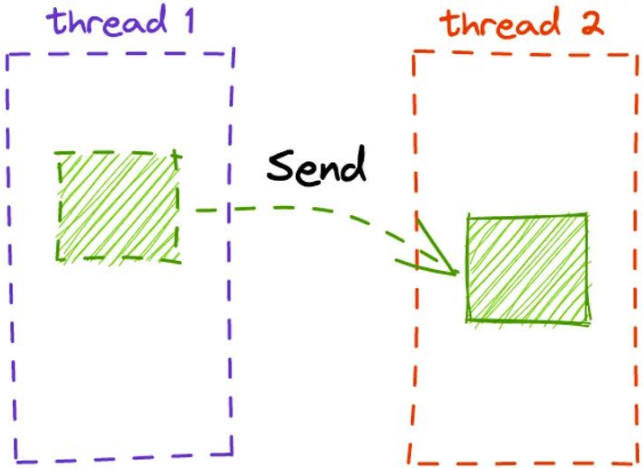
Building async bridge

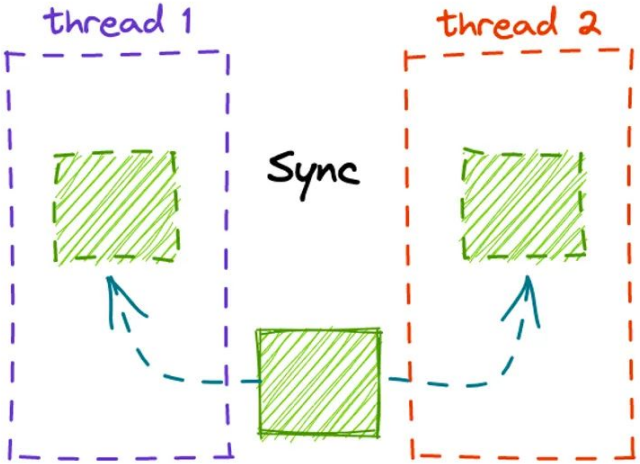


Building async bridge: Channel



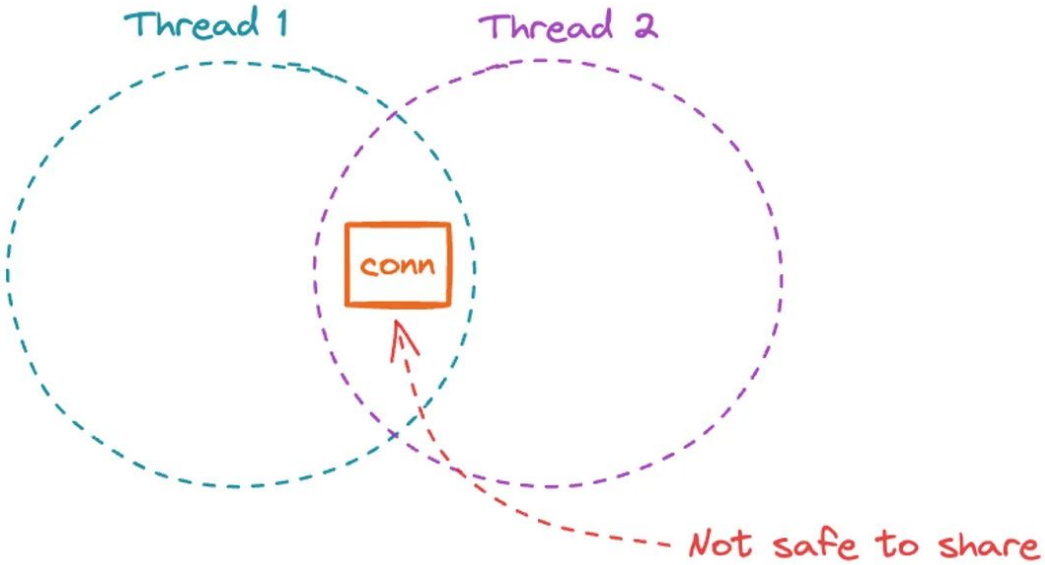
Async Thread Safety



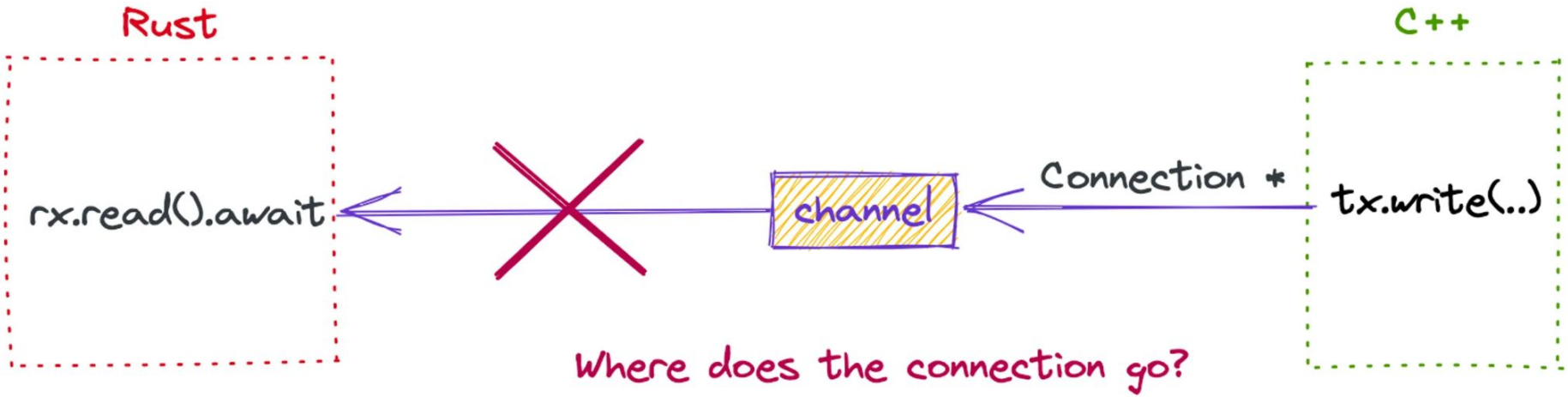


Async Thread Safety

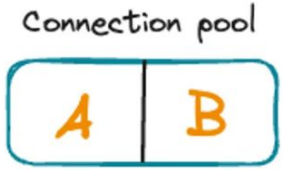
```
unsafe impl Send for Connection {}

unsafe impl Sync for Connection {}
```

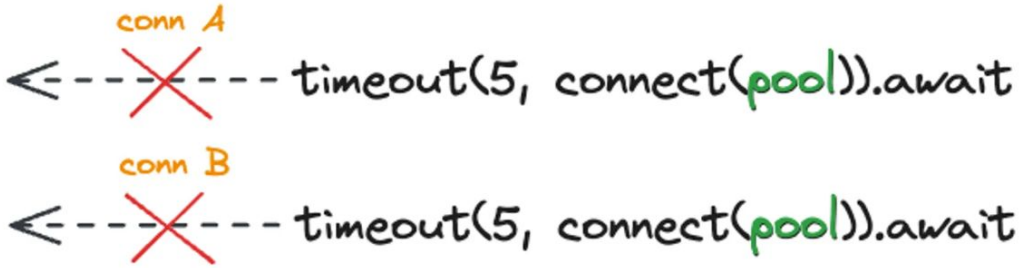


Danger of raw pointers

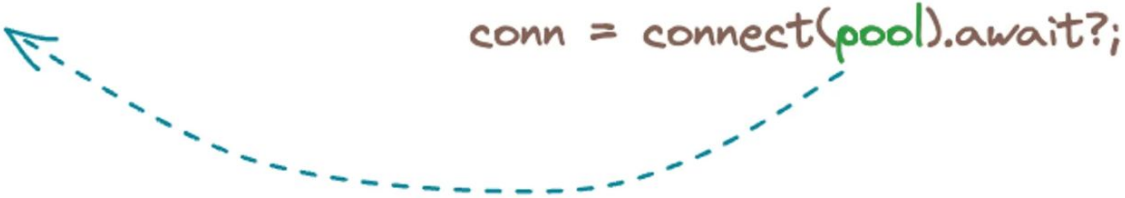


Danger of raw pointers

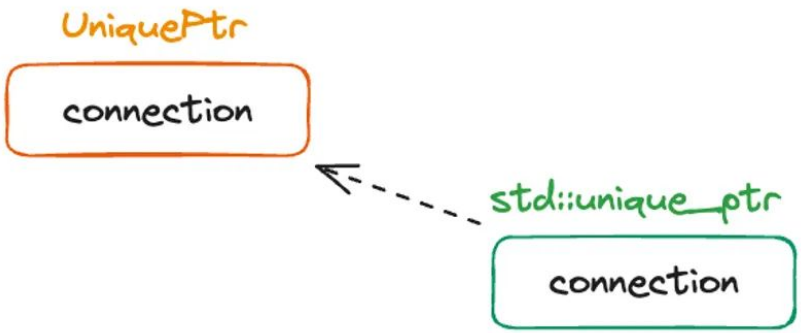




* new connect request
* waiting until there is a free connection to use



Danger of raw pointers




```rust
impl Drop for ConnectionWrapper {
    fn drop(&mut self) {
        ffi::free_connection(inner);
    }
}
```

Playground: Storage

`storage.h`: **`Storage`** – simple key-value store

```cpp
/// Simple key-value storage with asynchronous interface
/// Uses folly::Synchronized internally for thread safety
class Storage {
public:
    /// Create a new Storage instance
    static std::shared_ptr<Storage> init() {…

    /// Store a key-value pair
    folly::SemiFuture<folly::Unit> store(std::string key, std::string value) {…

    /// Fetch a value by key
    folly::SemiFuture<folly::Optional<std::string>> fetch(const std::string& key) {…

private:
    folly::Synchronized<std::unordered_map<std::string, std::string>> _data;
};

/// Get a reference to the global inline executor
folly::InlineExecutor& get_inline_executor() {
    return folly::InlineExecutor::instance();
}
```

Exercise 8: Sync shim API

- Let’s use Storage API via synchronous interface
  - Add layer of shim functions store/fetch that do a blocking wait until the futures are executed
  - Expose those to Rust via cxx bridge as usual
  - Run code and check!
- There are a couple of implemented shim functions as an example
  - Feel free to modify them
  - To add an optional shared value, for example

Exercise 8: Sync shim API

- Let’s use Storage API via synchronous interface
  - Add layer of shim functions store/fetch that do a blocking wait until the futures are executed
  - Expose those to Rust via cxx bridge as usual
  - Run code and check!
- There are a couple of implemented shim functions as an example
  - Feel free to modify them
  - To add an optional shared value, for example

Exercise 9.1: Define channel and callbacks

Let's create a channel and define callbacks

- Expose the transmitter as an opaque type to C++ using extern Rust block

```
struct StoreTransmitter(oneshot::Sender<Result<()>>);

fn store_ok(tx: Box<StoreTransmitter>) {
    let _ = tx.0.send(Ok(()));
}

fn store_fail(tx: Box<StoreTransmitter>, err: String) {
    let err = Error::msg(err);
    let _ = tx.0.send(Err(err));
}
```

Exercise 9.2: Expose `store` via sync API

- Implement the shim layer that sends the results over the channel in C++
- Expose the shim function to Rust
- Implement a wrapper that handles the channel work

```rust
async fn store_thing(
    storage: cxx::SharedPtr<ffi::Storage>,
    key: &cxx::CxxString,
    value: &cxx::CxxString,
) -> Result<()> {
    let (tx, rx) = oneshot::channel();
    let tx = Box::new(StoreTransmitter(tx));

    ffi::store(storage, key, value, store_ok, store_fail, tx);
    rx.await?
}
```

Exercise 9.3: Expose `fetch` via sync API

- Same as for the store but now we send a string result over the channel
- More so, we can send an optional value to handle “not-found” case

```
struct FetchTransmitter(oneshot::Sender<Result<Option<String>>>);

fn fetch_ok_found(tx: Box<FetchTransmitter>, result: String) {
    let _ = tx.0.send(Ok(Some(result)));
}
fn fetch_ok_not_found(tx: Box<FetchTransmitter>) {
    let _ = tx.0.send(Ok(None));
}

fn fetch_fail(tx: Box<FetchTransmitter>, err: String) {
    let err = Error::msg(err);
    let _ = tx.0.send(Err(err));
}
```

Thank you!