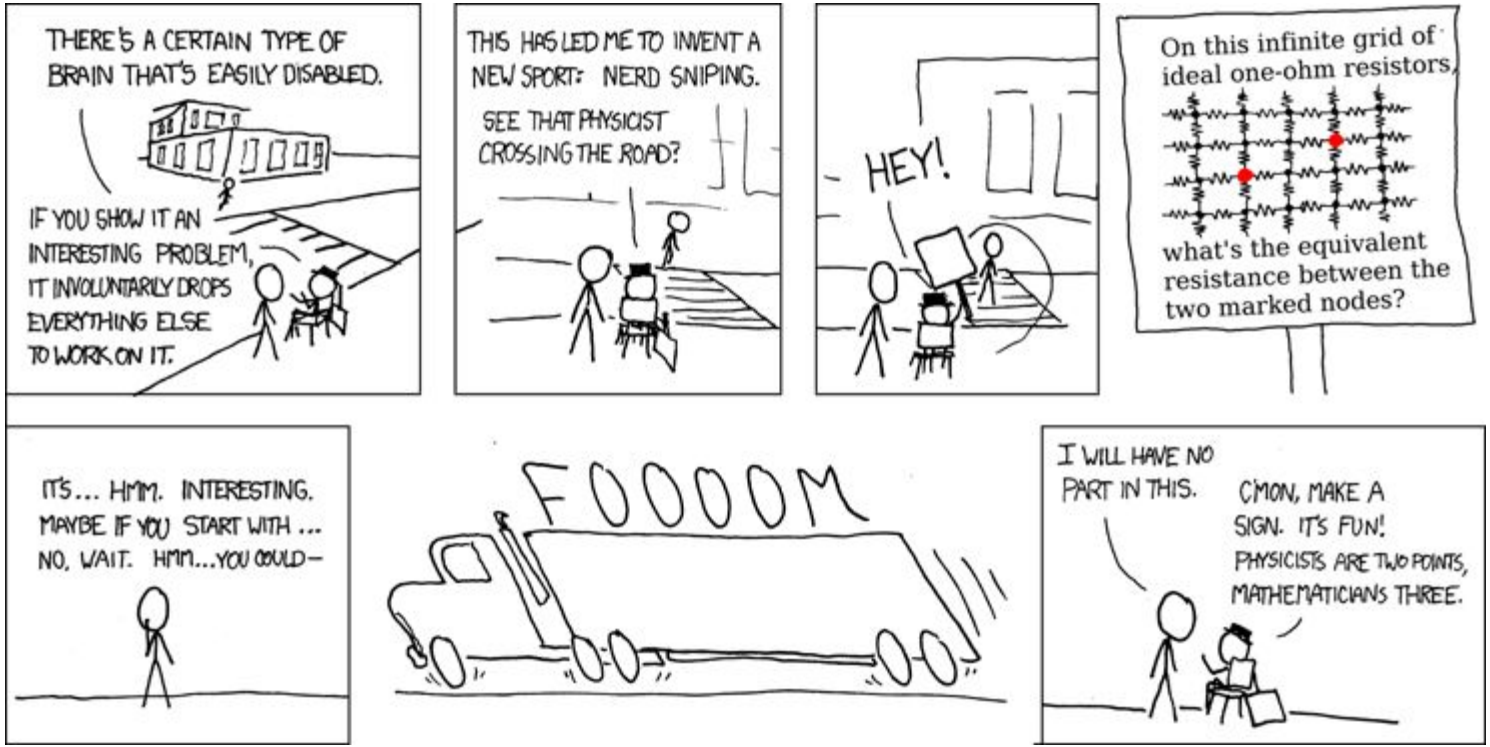
C++/Rust Interop:
Using Bridges in Practice

TYLER WEAVER

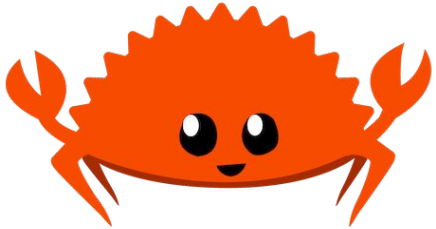


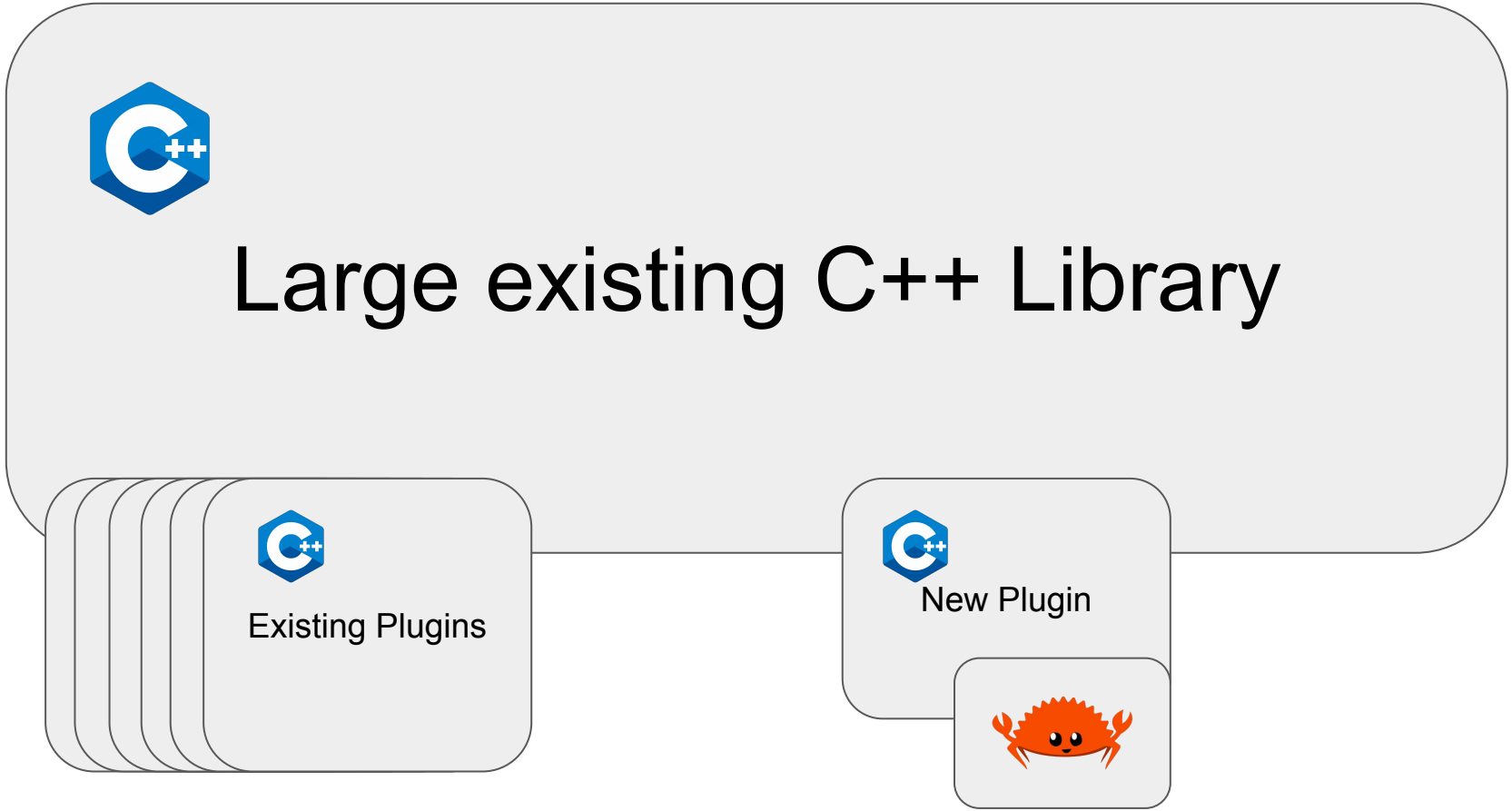
https://xkcd.com/356/

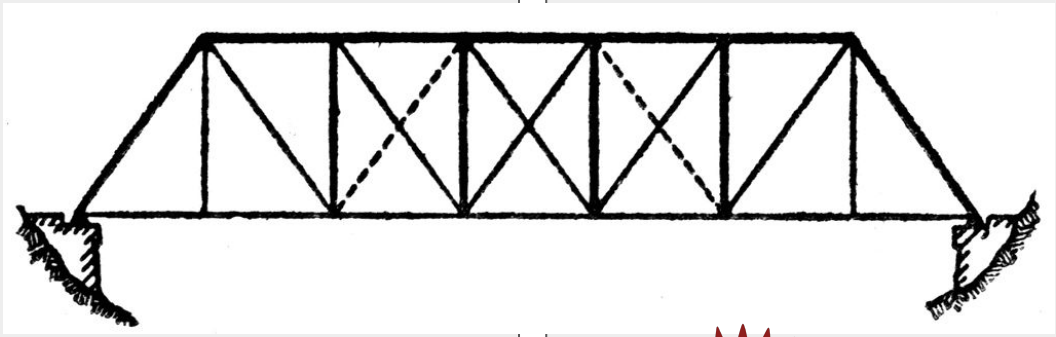
https://twitter.com/timur_audio/status/1004017362381795329

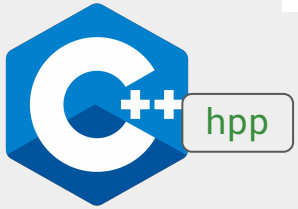

I fear no man.
But that thing...
it scares me.
imgflip.com

Part 1 - New Rust Extends a C++ Project

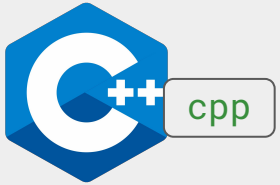



Large existing C++ Library
Existing Plugins
New Plugin

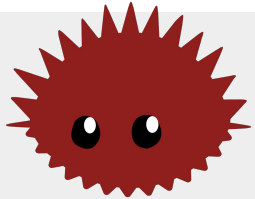




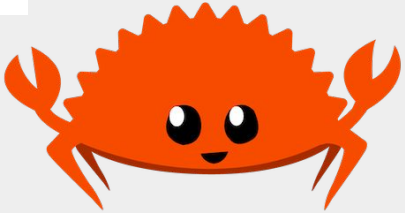
C++ Header
Class with
Methods



C++ Source
Uses extern
“C” from Rust



Unsafe Rust
C ABI

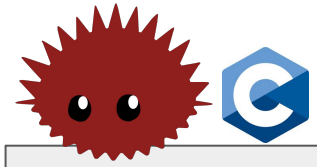


Safe Rust

```rust
pub struct Joint {
    name: String,
    parent_link_to_joint_origin: Isometry3<f64>,
}

impl Joint {
    pub fn new() -> Self;
}
```

hpp

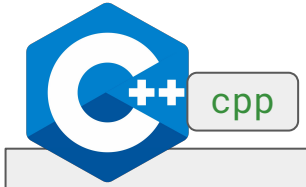
```cpp
class Joint {
public:
  Joint() noexcept;
  ~Joint() noexcept = default;

  Joint(Joint&& other) noexcept            = default;
  Joint& operator=(Joint&& other) noexcept = default;
};
```



```rust
#[no_mangle]
extern "C" fn robot_joint_new() -> *mut Joint {
    Box::into_raw(Box::new(Joint::new()))
}

#[no_mangle]
extern "C" fn robot_joint_free(joint: *mut Joint) {
    unsafe {
        drop(Box::from_raw(joint));
    }
}
```


```hpp
namespace robot_joint {

class Joint {
public:
  Joint() noexcept;
  ~Joint() noexcept = default;

  Joint(Joint&& other) noexcept            = default;
  Joint& operator=(Joint&& other) noexcept = default;

private:
  std::unique_ptr<rust::Joint, deleter_from_fn<robot_joint_free>> robot_joint_;
};

} // namespace robot_joint
```

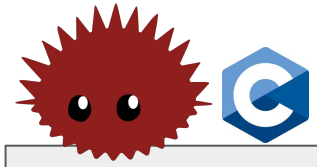

```hpp
namespace robot_joint::rust {
struct Joint;
} // namespace robot_joint::rust

extern "C" {
  extern void robot_joint_free(robot_joint::rust::Joint*);
}

template<auto fn>
struct deleter_from_fn {
  template<typename T>
  constexpr void operator()(T* arg) const {
    fn(arg);
  }
};
```



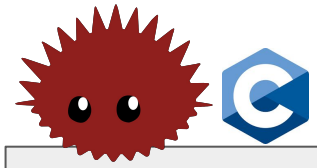

```cpp
extern "C" {
  extern robot_joint::rust::Joint* robot_joint_new();
}

namespace robot_joint {

Joint::Joint() noexcept : robot_joint_{robot_joint_new()} {}

} // namespace robot_joint
```

```hpp
class Joint {
  public:
    Eigen::Isometry3d calculate_transform(const Eigen::VectorXd& variables);
};
```

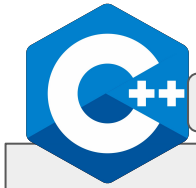

```rust
impl Joint {
    pub fn calculate_transform(&self, variables: &[f64]) -> Isometry3<f64>;
}
```



```rust
use std::ffi::c_double;

#[repr(C)]
struct Mat4d {
    data: [c_double; 16],
}
```

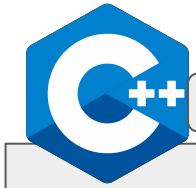


```rust
use std::ffi::{c_double, c_uint};

#[no_mangle]
extern "C" fn robot_joint_calculate_transform(
    joint: *const Joint,
    variables: *const c_double,
    size: c_uint,
) -> Mat4d {
    unsafe {
        let joint = joint.as_ref().expect("Invalid pointer to Joint");
        let variables = std::slice::from_raw_parts(variables, size as usize);
        let transform = joint.calculate_transform(variables);
        Mat4d {
            data: transform.to_matrix().as_slice().try_into().unwrap(),
        }
    }
}
```



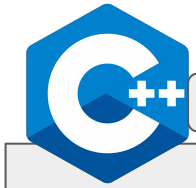
cpp

```cpp
struct Mat4d {
  double data[16];
};

extern "C" {
extern struct Mat4d robot_joint_calculate_transform(
  const robot_joint::rust::Joint*, const double*, unsigned int);
}
```



cpp

```cpp
namespace robot_joint {
Eigen::Isometry3d Joint::calculate_transform(
  const Eigen::VectorXd& variables
)
{
  const auto rust_isometry = robot_joint_calculate_transform(
    robot_joint_.get(), variables.data(), variables.size());
  Eigen::Isometry3d transform;
  transform.matrix() = Eigen::Map<Eigen::Matrix4d>(rust_isometry.data);
  return transform;
}
}  // namespace robot_joint
```

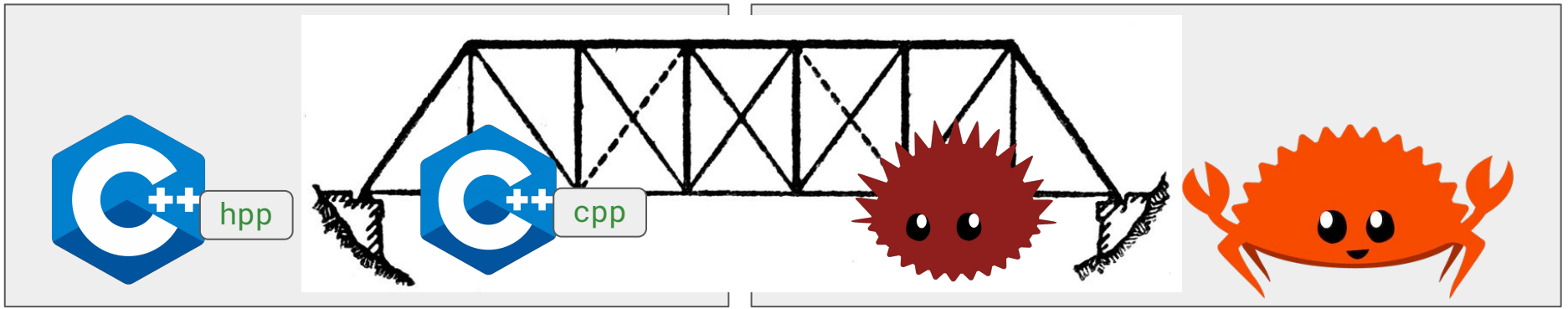



```hpp
class Joint {
  public:
    Eigen::Isometry3d calculate_transform(const Eigen::VectorXd& variables);
};
```

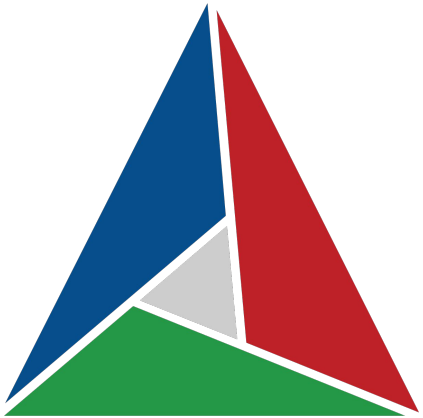

```rust
impl Joint {
    pub fn calculate_transform(&self, variables: &[f64]) -> Isometry3<f64>;
}
```

Manual Interop

- Create unsafe Rust functions for creating and deleting Rust objects
- Store the Rust object in a unique_ptr
- Move only C++ type containing unique_ptr of Rust type with methods
- Fixed sized arrays wrapped in structs can be used in FFI
- Use method implementations to bridge library types

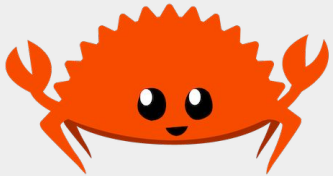


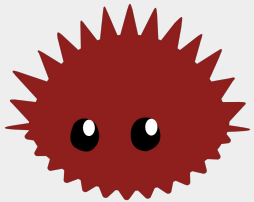
Part 2 - Exposing a CMake Target

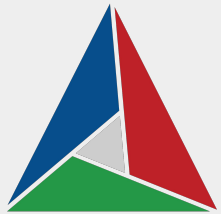


CMake With C++ Interop Layer

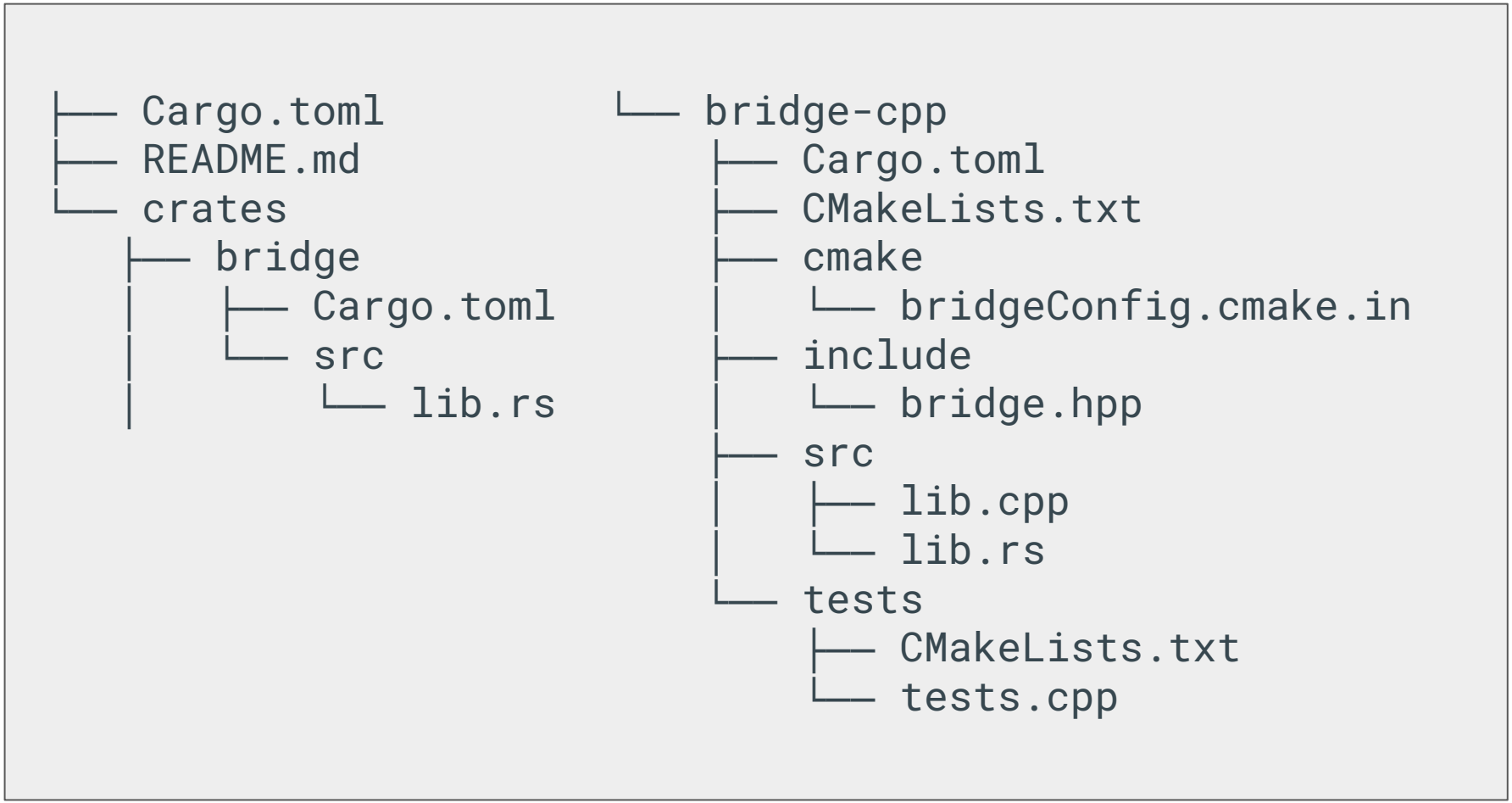
Rust Library with Unsafe







```
include(FetchContent)
FetchContent_Declare(
  bridge
  GIT_REPOSITORY https://github.com/you/bridge
  GIT_TAG main
  SOURCE_SUBDIR "crates/bridge-cpp")
FetchContent_MakeAvailable(bridge)

target_link_libraries(mytarget PRIVATE bridge::bridge)
```


├── Cargo.toml
├── README.md
└── crates
    ├── bridge
    │   ├── Cargo.toml
    │   └── src
    │   └── lib.rs
    └── bridge-cpp
        ├── Cargo.toml
        ├── CMakeLists.txt
        ├── cmake
        │   └── bridgeConfig.cmake.in
        ├── include
        │   └── bridge.hpp
        ├── src
        │   ├── lib.cpp
        │   └── lib.rs
        └── tests
            ├── CMakeLists.txt
            └── tests.cpp

Cargo.toml

```toml
[workspace]
members = ["crates/bridge", "crates/bridge-cpp"]

[workspace.package]
version = "0.1.0"
edition = "2021"
```

crates/bridge-cpp/Cargo.toml

```toml
[package]
name = "bridge-cpp"
version.workspace = true
edition.workspace = true

[lib]
name = "bridge_unsafe"
crate-type = ["staticlib"]

[dependencies]
bridge = { path = "../bridge" }
```

```
└── bridge-cpp
    ├── Cargo.toml
    ├── CMakeLists.txt
    ├── cmake
    │   └── bridgeConfig.cmake.in
    ├── include
    │   └── bridge.hpp
    ├── src
    │   ├── lib.cpp
    │   └── lib.rs
    └── tests
        ├── CMakeLists.txt
        └── tests.cpp
```

`crates/bridge-cpp/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.16)
project(bridge VERSION 0.1.0)

find_package(Eigen3 REQUIRED)
```

`crates/bridge-cpp/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.16)
project(bridge VERSION 0.1.0)

find_package(Eigen3 REQUIRED)

include(FetchContent)
FetchContent_Declare(
  Corrosion
  GIT_REPOSITORY
https://github.com/corrosion-rs/corrosion.git
  GIT_TAG v0.4)
FetchContent_MakeAvailable(Corrosion)
```

crates/bridge-cpp/CMakeLists.txt

```
include(FetchContent)
FetchContent_Declare(
  Corrosion
  GIT_REPOSITORY
https://github.com/corrosion-rs/corrosion.git
  GIT_TAG v0.4)
FetchContent_MakeAvailable(Corrosion)

corrosion_import_crate(
  MANIFEST_PATH Cargo.toml CRATES bridge-cpp)
```

crates/bridge-cpp/CMakeLists.txt

```
corrosion_import_crate(
  MANIFEST_PATH Cargo.toml CRATES bridge-cpp)

add_library(bridge STATIC src/lib.cpp)
target_include_directories(
  bridge PUBLIC
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
                $<INSTALL_INTERFACE:include>)
target_link_libraries(bridge PUBLIC Eigen3::Eigen)
target_link_libraries(bridge PRIVATE bridge_unsafe)
set_property(TARGET bridge PROPERTY CXX_STANDARD 20)
set_property(TARGET bridge
  PROPERTY POSITION_INDEPENDENT_CODE ON)
```

crates/bridge-cpp/CMakeLists.txt

```
include(CMakePackageConfigHelpers)
include(GNUInstallDirs)

install(
  TARGETS bridge bridge_unsafe
  EXPORT ${PROJECT_NAME}Targets
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
  LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
  ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR})
```

crates/bridge-cpp/CMakeLists.txt

```
install(
  EXPORT ${PROJECT_NAME}Targets
  NAMESPACE bridge::
  DESTINATION
"${CMAKE_INSTALL_LIBDIR}/cmake/${PROJECT_NAME}")
```

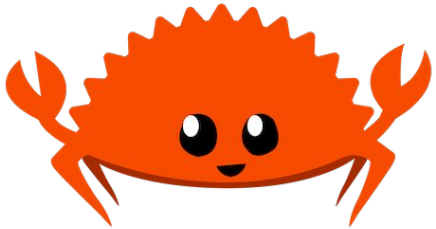
crates/bridge-cpp/CMakeLists.txt

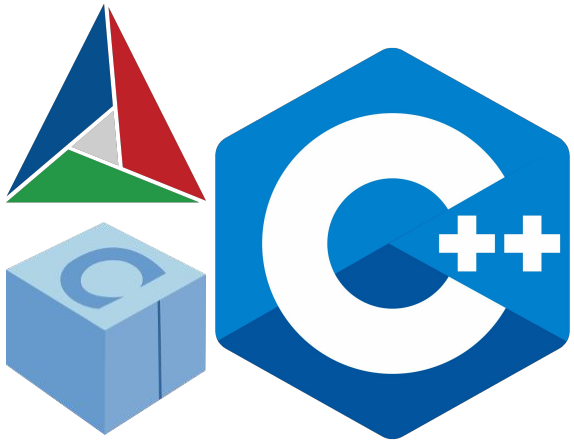
```
configure_package_config_file(
  cmake/bridgeConfig.cmake.in
  "${PROJECT_BINARY_DIR}/${PROJECT_NAME}Config.cmake"
  INSTALL_DESTINATION
  "${CMAKE_INSTALL_LIBDIR}/cmake/${PROJECT_NAME}")
install(
  FILES "${PROJECT_BINARY_DIR}/${PROJECT_NAME}Config.cmake"
  DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/${PROJECT_NAME}")

install(
  FILES include/bridge.hpp
  DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
```

crates/bridge-cpp/cmake/bridgeConfig.cmake.in

```
@PACKAGE_INIT@

include(CMakeFindDependencyMacro)
find_dependency(Eigen3)

include(
  "${CMAKE_CURRENT_LIST_DIR}/@PROJECT_NAME@Targets.cmake")
```

Part 3 - Using a C++ library in Rust





Cargo.toml

```
[dependencies]
cxx = "1.0"

[build-dependencies]
cxx-build = "1.0"
anyhow = "1.0.79"
git2 = "0.18.2"
conan2 = "0.1"
cmake = "0.1"
```

build.rs

```
use conan2::ConanInstall;
use std::path::{Path, PathBuf};

fn main() -> anyhow::Result<()> {
    println!("cargo:rerun-if-changed=build.rs");
    println!("cargo:rerun-if-changed=src");

    let out_dir: PathBuf = std::env::var_os("OUT_DIR")
        .expect("OUT_DIR environment variable must be set")
        .into();
```

build.rs

```
use conan2::ConanInstall;
use std::path::{Path, PathBuf};

fn main() -> anyhow::Result<()> {
    println!("cargo:rerun-if-changed=build.rs");
    println!("cargo:rerun-if-changed=src");

    let out_dir: PathBuf = std::env::var_os("OUT_DIR")
        .expect("OUT_DIR environment variable must be set")
        .into();
```

build.rs

```
println!("cargo:rerun-if-changed=build.rs");
println!("cargo:rerun-if-changed=src");

let out_dir: PathBuf = std::env::var_os("OUT_DIR")
    .expect("OUT_DIR environment variable must be set")
    .into();

let data_tamer_url = "https://github.com/PickNikRobotics/data_tamer";
let data_tamer_source = out_dir.join(Path::new("data_tamer"));
if !data_tamer_source.exists() {
    git2::Repository::clone(data_tamer_url, data_tamer_source.as_path())?;
}
let data_tamer_cpp = out_dir.join(Path::new("data_tamer/data_tamer_cpp"));
```

build.rs

```
let data_tamer_url = "https://github.com/PickNikRobotics/data_tamer";
let data_tamer_source = out_dir.join(Path::new("data_tamer"));
if !data_tamer_source.exists() {
    git2::Repository::clone(data_tamer_url, data_tamer_source.as_path())?;
}
let data_tamer_cpp = out_dir.join(Path::new("data_tamer/data_tamer_cpp"));

let conan_instructions = ConanInstall::with_recipe(&data_tamer_cpp)
    .build("missing")
    .run()
    .parse();
let conan_includes = conan_instructions.include_paths();
let toolchain_file = out_dir.join(
    Path::new("build/Debug/generators/conan_toolchain.cmake"));
```

build.rs

```
let conan_instructions = ConanInstall::with_recipe(&data_tamer_cpp)
    .build("missing")
    .run()
    .parse();
let conan_includes = conan_instructions.include_paths();
let toolchain_file = out_dir.join(
    Path::new("build/Debug/generators/conan_toolchain.cmake"));

let data_tamer_install = cmake::Config::new(&data_tamer_cpp)
    .define("CMAKE_TOOLCHAIN_FILE", toolchain_file)
    .build();
let data_tamer_lib_path = data_tamer_install.join(Path::new("lib"));
let data_tamer_include_path =
    data_tamer_install.join(Path::new("include"));
```

build.rs

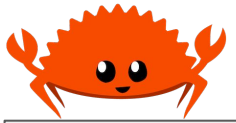
```
let data_tamer_install = cmake::Config::new(&data_tamer_cpp)
    .define("CMAKE_TOOLCHAIN_FILE", toolchain_file)
    .build();
let data_tamer_lib_path = data_tamer_install.join(Path::new("lib"));
let data_tamer_include_path =
    data_tamer_install.join(Path::new("include"));

cxx_build::bridge("src/main.rs")
    .includes(conan_includes)
    .include(data_tamer_include_path)
    .include("src")
    .std("c++17")
    .compile("demo");
```

build.rs

```
    cxx_build::bridge("src/main.rs")
        .includes(conan_includes)
        .include(data_tamer_include_path)
        .include("src")
        .std("c++17")
        .compile("demo");

    println!(
        "cargo:rustc-link-search=native={}",
        data_tamer_lib_path.display()
    );
    println!("cargo:rustc-link-lib=static=data_tamer");

    conan_instructions.emit();
    Ok(())
}
```

shim.hpp

```cpp
#pragma once

#include <memory>

namespace DataTamer
{
    template <typename T, typename... Args>
    std::unique_ptr<T> construct_unique(Args... args)
    {
        return std::make_unique<T>(args...);
    }
}
```



main.rs

```
#[cxx::bridge(namespace = "DataTamer")]
mod data_tamer {
    unsafe extern "C++" {
        include!("shim.hpp");
        include!("data_tamer/data_tamer.hpp");

        type ChannelsRegistry;

        #[rust_name = "channels_registry_new"]
        fn construct_unique() -> UniquePtr<ChannelsRegistry>;
    }
}

fn main() {
    let mut registry = data_tamer::channels_registry_new();
}
```

Rust and C++

Some creativity is required

Many helpful tools exist