slint

RUST AND C++

TOBIAS HUNGER

Speaker notes

Organisational:

- Please tell me if I am too fast or too slow!
- Slides will be available, list of links on last slide!
- Feel free to ask questions, we will probably not have too much time at the end. I might not be able to answer all of them here, but I am around afterwards and will also provide ways to reach out online!

There is a lot of interest in using languages safer than C++ right now, but there is also a lot of C++ code out there that we can not just leave behind.

About the time I handed is the paper for this presentation Chandler Carruth introduced a new programming language at CppNorth It is supposed to become safer than C++, but at the same time stay fully compatible with it -- so that existing C++ code can be reused.

Since then Herb Sutter introduced cppfront at CppCon, a transpiler that compiles a new language called "cpp2" into C++ and then builds it with a standard C++ compiler. Again a new language fully compatible with C++.

Rust was developed at Mozilla, also with the goal to integrate into their existing C++ code base. Indeed rust code is shipped shipped in combination with a lot of C++ code in the firefox browser today.

Integrating Rust and C++ is possible. This presentation wants to show some options you have to combine Rust and C++ code.

slint

THE FAST AND EASY UI TOOLKIT

ABOUT ME

- Long time C++ Developer
- Now: Rust Developer

Speaker notes

C++ DEV

25 or so years, most of the time around and on UI technologies and tooling.

RUST

…still working on tooling…

SLINT

slint

THE FAST AND EASY UI TOOLKIT

SCALABLE LIGHTWEIGHT INTUITIVE NATIVE UI TOOLKIT

Implemented in 100% Rust

- Provides C++ API
- Optionally uses existing C++ code

Speaker notes

Scalable: Microcontroller to graphics accelerated workstation

Lightweight: Uses few resources

Intuitive: Simple UI description language

Native: 100% Rust -- incl. the UI description parts

We want to provide a nice API for users of other languages as well. Supports JS and C++ in addition to Rust. We want Slint to feel native in all languages.

We optionally use existing code from C++ toolkits for some platform integration tasks.

TOPICS

- Code Level Integration
- Build-System Integration

Speaker notes

TALK

Make Rust call C++ code and the other way around.

BUILD

Build projects that consist of rust code and C++ code

CODE-LEVEL INTEGRATION

Speaker notes

Challenges:

- Rust has no defined ABI
- … neither has C++ (compiler defined!)
- Languages have different concepts like inheritance, lifetimes, templates, …
- Data types do not match up. Even if they contain the "same" data, field names/sequence/types will be different!
- Even if datatypes would match up: Different requirements on the representation of contained data. E.g. utf-8 encoded strings in rust vs. bytes in unknown encoding in C++)

Least common denominator: C foreign function interface

C FFI is the backbone all the options we discuss are built upon.

AUTOMATIC BINDING GENERATION

- `bindgen`: C and simple C++ → Rust
- `cbindgen`: Rust → C or C++11 headers

Speaker notes

Idea: Examine existing code and generate C FFI-based bindings from that!

These tools typically are uni-directional!

BINDGEN

Parses header files and generates a rust code out of it.

Will try to convert whatever it can, skipping constructs it can not handle.

Problems incl. templates, inline functions, exceptions, automatically calling copy/move constructors, cross-language inheritance.

Needs manual configuration to block types that don't work or mark them as "opaque"

CBINDGEN

Parses rust code and exposes types and functions marked as `repr("C")`.

Typically works reliably as the programmer has already put in the hard work by marking up functions and types.

These are just the most commonly used ones, more tools available.

SEMI-AUTOMATIC BINDING GENERATION

cxx CRATE

Speaker notes

Requires custom code/data to generate bindings for C++ and Rust from. A C FFI interface is "hidden" between the two bindings.

CXX

Most widely used crate in this area, other build on top or beside this one.

cxx CRATE

```rust
#[cxx::bridge]
mod ffi {
    struct Metadata {
        size: usize,
        tags: Vec<String>,
    }

    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        include!("demo/include/blobstore.h");
```

Speaker notes

Code taken straight from crates demo!

Aim: Describe interface and generate *safe* and *fast* bindings from and to C++ code.

safe: Safe in the rust sense: Rust compiler enforces its invariants. The C++ isn't safe in this sense: The rust compiler can not reason about the C++ code. The code generated by cxx is safe.

fast: Zero copy, no transformations going between languages => custom types like `CxxString` in rust and `rust::String` in C++!

Interface definition in rust. Macro for the win!

Shared data structures go straight into the `ffi` module. cxx maps data types.

structs and functions/methods exposed from Rust. Must be in parent scope!

structs and functions/methods exposed from C++. Important! Add unsafe/lifetimes here, which will be ignored in C++ but leads to better bindings. cxx will assert that signatures match!

How does this work?

Code is generated from this definition that implements C FFI + nicer facade in front of that:

- Rust side: Macro-based, so during rust build
- C++: As part of build.rs/via command line tool. Needs build system integration!

NO BINDING GENERATION

- cpp crate

Speaker notes

Use C++ code directly! Of course you still need to do bindings for data types and such ;-)

CPP

Some macros that enable embedding of C++ code right into the middle of your rust code!

Works by extracting the C++ bits, wrapping them into function definitions and calling them in the right places.

What does that look like?

```
fn notify(&self) {
  let obj = self.obj;









}
```

Speaker notes

Just write normal rust code…

```rust
fn notify(&self) {
  let obj = self.obj;
  cpp!(unsafe [obj as "Object*"] {
    auto data = queryInterface(obj)->data();



    updateA18y(Event(obj));
  });
}
```

Speaker notes

… then use the `cpp` macro and continue to write C++.

```rust
fn notify(&self) {
  let obj = self.obj;
  cpp!(unsafe [obj as "Object*"] {
    auto data = queryInterface(obj)->data();
    rust!(rearm [data: Pin<&A18yItemData> as "void*"] {
      data.arm_state_tracker();
    });
    updateA18y(Event(obj));
  });
}
```

Speaker notes

You can even use a `rust!` pseudo-macro to switch back into rust.

You might still need some data types you can pass between languages. cpp has some more macros to help defining those. You need that when both sides need to access the same data. Otherwise you can use void*/*c_void to pass objects through the "other" language.

Slint uses cpp crate to integrate with existing C++ toolkits.

SUMMARY

- A whole range of options for Rust to talk to C++ code
  - … none is fully automatic!
- No C++ compatibility built into Rust
  - External code generator
  - Macros!

Speaker notes

Rust can develop independent of strength and weaknesses of C++ type system. That's a good thing for a language ;-)

Integration of C++ into Rust via standard mechanisms of the language.

Macros in rust are so extremely powerful!

Integration of rust into C++ limited, but possible

BUILD-SYSTEM INTEGRATION

Speaker notes

Now that we know how to make C++ and Rust talk to each other, how can we build the combined codebase? It depends on which language implements the final binary.

You will typically have one build system for your C++ project and another build system for your rust project. The task is to integrate one build system into the other: It's often not practical to redo the build system of a project!

Replacing the Rust build system is tricky: Cargo is the official interface to build rust code. The rust compiler's command line interface is an internal interface according to the rust community.

slint

CARGO

A little C++ in a Rust project.

`build.rs`

- `cc` crate
- `cmake` crate

Speaker notes

First step: Check crates.io! Maybe somebody has already done bindings?

BUILD.RS

If you want to build a rust binary linking to C/C++ code, then you need to integrate generating that into cargo

You extend Cargo build adding a build.rs file. That's built during the build and then executed to extend Cargos functionality!

CC AND CMAKE

To make C/C++ integration easier you can use crates as dev-dependency (build time dependency!).

E.g. cc to drive the C/C++ compiler directly or cmake to build a cmake project.

Of course you can also go the other direction: Let cmake drive the build and delegate to cargo to build some rust code into a library.

slint

CMAKE

A little Rust in a C++ project.

CORROSION

```
cmake_minimum_required(VERSION 3.15)
project(MyCoolProject LANGUAGES CXX)

find_package(Corrosion REQUIRED)

corrosion_import_crate(MANIFEST_PATH rust-lib/Cargo.toml)

add_executable(cpp-exe main.cpp)
target_link_libraries(cpp-exe PUBLIC rust-lib)
```

Speaker notes

The corrosion project on Github tries to make it simple to have a cargo project as a subproject in CMake.

It will parse your cargo build system and expose all build targets to cmake.

Slint uses corrosion to allow C++ people to work with slint.

THANK YOU!

Mail: tobias.hunger@slint-ui.com

Twitter: @t_hunger

slint

LINKS I

- Carbon announcement
- Cppfront announcement
- Slint website
- bindgen crate
- cbindgen crate
- Challenges converting C++ to Rust
- KDE rust binding generator

LINKS II

- KDE rust binding generator
- cxx crate
- cpp crate
- Olivier Goffart talking about the cpp crate
- cc crate
- cmake crate
- corrosion-rs