FFI

Jack Duvall & Cooper Pierce

Carnegie Mellon University

With C

So You Want To Call C From Rust, Huh?

Conceptually, not too bad, just a few simple steps:

- Declare what C functions are available
- Link against the C library
- Call the function, using `unsafe`

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

a.k.a. how to talk to other people ('s C code)

Review: Calling Conventions

a.k.a. the Application Binary Interface (ABI)

a.k.a. how to talk to other people ('s C code)

- How are arguments passed?
- What registers are clobbered?
- How do you get the return value?

Rust Supported Calling Conventions (via LLVM)

- "Rust"—Rust's own calling convention
- "C"—(default) calling convention used by your C compiler
- "system"—calling convention used by your OS, usually same as "C" except on Win32 where it's "stdcall"
- "cdecl"—x86_32 calling convention
- "stdcall"—Win32 x86_32 ABI
- "win64"—x86_64 Windows ABI
- "sysv64"—x86_64 non-Windows
- "aapcs"—ARM
- "fastcall"
- "vectorcall"

See https://doc.rust-lang.org/reference/items/external-blocks.html for details.

External Linkage With extern

```rust
// Function we can link against
extern "C" {
    fn my_other_c_function(x: i32, y: i32) -> i32;
}

// Function that we export and can be linked to
#[no_mangle]
extern "C" fn my_rust_function(x: i32, y: i32) -> i32 { ... }
```

Review[1]: Linkage

How can we link code?

- Dynamic Linkage: "hey OS i want this library please load it when u launch me"
  - Pros: Smaller binary size, flexible to upgrade library
  - Cons: Code can't handle upgrades in a significant number of cases
- Static Linkage: "hey Compiler i want this library please put it next to my code"
  - Pros: You always get the library version you want
  - Cons: Upgrading requires re-compilation

[0] okay, probably not

Specifying Linkage for extern C Functions

```
#[link(name = "foo")] // kind = "dylib"
extern {
    fn cool_foo() -> *const u8;
}

#[link(name = "bar", kind = "static")]
extern {
    fn cool_bar() -> *const u8;
}
```

Things To Watch Out For

A couple of potential linking pitfalls:

- Your compiler can find the library you're linking against
  - For dynamic libraries, OS needs to find too!
  - For very fancy libraries, needs to be built by the same compiler!
- Your definitions in Rust exactly match the definitions in C

`bindgen`

Idea: Computer, Write Rust FFI For Me

Steps:

- Tell `bindgen` to make bindings at compile time
- Use `include!` macro to textually include generated bindings
- Link against C library
- Call the functions using `unsafe`

How Do We Do Stuff At Compile Time?

`build.rs` scripts!

- Placed at root of package next to `Cargo.toml`
- Run before Rust code compiled, can do arbitrary configuration since it's a binary itself
- Special output used to control behavior of Cargo

Small `build.rs` Example

```rust
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");
}
```

bindgen build.rs Example

```rust
fn main() {
    println!("cargo:rustc-link-lib=bz2");
    println!("cargo:rerun-if-changed=wrapper.h");
    let bindings = bindgen::Builder::default()
        .header("wrapper.h")
        .parse_callbacks(Box::new(bindgen::CargoCallbacks))
        .generate()
        .expect("Unable to generate bindings");
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

`build.rs` Handles Linkage For Us!

```rust
println!("cargo:rustc-link-lib=bz2");
```

does dynamic linking, looking for libbz2.so, and

```rust
println!("cargo:rustc-link-lib=static=bz2");
```

does static linking, looking for libbz2.a

See https://doc.rust-lang.org/cargo/reference/build-scripts.html for all options

Including Generated Bindings

This step needs to be done because Cargo only looks at the source tree for files to compile, and `build.rs` scripts should not be modifying that directly:

```
// Contents of src/lib/ffi.rs

#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

Example C Header To Parse

```c
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c, CoolStruct* cs);
```

Example bindgen Generated Bindings

```rust
#[repr(C)]
pub struct CoolStruct {
    pub x: ::std::os::raw::c_int,
    pub y: ::std::os::raw::c_int,
}

extern "C" {
    pub fn cool_function(i: ::std::os::raw::c_int,
                         c: ::std::os::raw::c_char,
                         cs: *mut CoolStruct);
}
```

With C++

One Option: “C++ is just C”

One Option: “C++ is just C”

- Make a C interface to your C++ library

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust
- ???

One Option: “C++ is just C”

- Make a C interface to your C++ library
- Use the same techniques as before to use that interface in Rust
- ???
- Profit?

The Issue: C++ Is Not Just C

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
  - `std::string` → `char *`

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
  - `std::string` → `char *`
  - `std::vector<int>` → `int *`

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
  - `std::string` → `char *`
  - `std::vector<int>` → `int *`
- We lose safety guarantees if we just use pointers

The Issue: C++ Is Not Just C

- Lots of common types are painful to convert back to C representations
  - `std::string` → `char *`
  - `std::vector<int>` → `int *`
- We lose safety guarantees if we just use pointers
- Hey, wait a minute, doesn't Rust solve those same problems?

`cxx`

Main Features

- Shared Structs/Enums
- Opaque Types (on either side)
- Functions (on either side)
    - Not type-generic ones though!

Canonical Example

```
#[cxx::bridge]
mod ffi {

    extern "Rust" {
        // Rust stuff
    }

    unsafe extern "C++" {
        // C++ stuff
    }

}
```

Rust Stuff: All The Stuff You Love!

```rust
type MultiBuf;

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

- Can also use `String`, `&str`, `Vec<T>`, `&[T]`, `Box<T>`!
- Converted to `rust::String`, `rust::Str`, `rust::Slice<T>`, `rust::Box<T>`, `rust::Vec<T>` in C++ code
- These are C++-native types, with the utilities you expect, much easier to work with than raw pointers

C++ Stuff: All The Stuff You Can Tolerate!

- `std::unique_ptr<T>`, `std::shared_ptr<T>`, `std::string`, `std::vector<T>`
- Converted to `UniquePtr<T>`, `SharedPtr<T>`, `CxxString`, `CxxVector` in Rust code
- `Result<T>` from Rust will be `rust::Error` in C++ and a C++ function throwing an exception will be `Result<T, cxx:Exception>` in Rust

C++ Stuff: Code Example

```
include!("example/include/blobstore.h");

type BlobstoreClient;

fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;

fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
```

Not Quite Complete

There are a couple missing features:

- C++ function pointers → Rust

Not Quite Complete

There are a couple missing features:

- C++ function pointers → Rust

... and that's basically it!

With Python

Everyone[1] loves Python!

C++ has PyBind11, do we have anything like that for Rust?

[1] Well, maybe not *everyone*...

Everyone[1] loves Python!

C++ has PyBind11, do we have anything like that for Rust?

Of course we do! PyO3

[1] Well, maybe not *everyone*...

Everyone[1] loves Python!

C++ has PyBind11, do we have anything like that for Rust?

Of course we do! PyO3

- Limited to Python-understandable types, but a lot of conversions

[1] Well, maybe not *everyone*...

pyo3 FFI: Rust Side

```rust
use pyo3::prelude::*;

/// Formats the sum of two numbers as string.
#[pyfunction]
fn sum_as_string(a: usize, b: usize) -> PyResult<String> {
    Ok((a + b).to_string())
}

/// A Python module implemented in Rust.
#[pymodule]
fn pyo3_example(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum_as_string, m)?)?;
    Ok(())
}
```

pyo3 FFI: Python Side

```python
import pyo3_example
print(pyo3_example.sum_as_string(123432, 432432)[0])
```

pyo3: Going The Other Way

```rust
use pyo3::prelude::*;
use pyo3::types::IntoPyDict;

fn main() -> PyResult<()> {
    Python::with_gil(|py| {
        let sys = py.import("sys")?;
        let version: String = sys.getattr("version")?.extract()?;
        let locals = [("os", py.import("os")?)].into_py_dict(py);
        let code = "os.getenv('USER')";
        let user: String = py.eval(code, None, Some(&locals))?
                             .extract()?;
        println!("Hello {}, I'm Python {}", user, version);
        Ok(())
    })
}
```

With Javascript

What's Another Compilation Target Among Friends?

Rust can target WebAssembly!

[2] Better than Java!

What's Another Compilation Target Among Friends?

Rust can target WebAssembly!
What is WebAssembly?

[2] Better than Java!

What's Another Compilation Target Among Friends?

Rust can target WebAssembly!
What is WebAssembly?

- A "low-level assembly-like language with a compact binary format that runs with near-native performance" in the browser (MDN)

[2] Better than Java!

What's Another Compilation Target Among Friends?

Rust can target WebAssembly!
What is WebAssembly?

- A "low-level assembly-like language with a compact binary format that runs with near-native performance" in the browser (MDN)
- Speeds up critical parts of web applications

[2] Better than Java!

What's Another Compilation Target Among Friends?

Rust can target WebAssembly!
What is WebAssembly?

- A "low-level assembly-like language with a compact binary format that runs with near-native performance" in the browser (MDN)
- Speeds up critical parts of web applications
- Also used for cross-platform binaries[2]

[2] Better than Java!

`wasm_bindgen` FFI: Rust side

```rust
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet() {
    alert("Hello, wasm-game-of-life!");
}
```

`wasm_bindgen` FFI: Rust side

```rust
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet() {
    alert("Hello, wasm-game-of-life!");
}
```

Use the `wasm-pack` tool to compile this code into WebAssembly!

wasm_bindgen FFI: Javascript side

```
import init, { greet } from './pkg/wasm_example.js';

async function run() {
    await init();
    greet();
}
```

wasm_bindgen FFI: Javascript side

```
import init, { greet } from './pkg/wasm_example.js';

async function run() {
    await init();
    greet();
}
```

Not pictured: trying to get this to work with unholier JS build systems