Rust + Nix

Zach Mitchell - flox

Me

- Rustacean since 2018
- Nix user for ~9 months
- Nix documentation team member for ~6 months
- Engineer at flox for ~4 months
- Irredeemable nerd for about 32 years

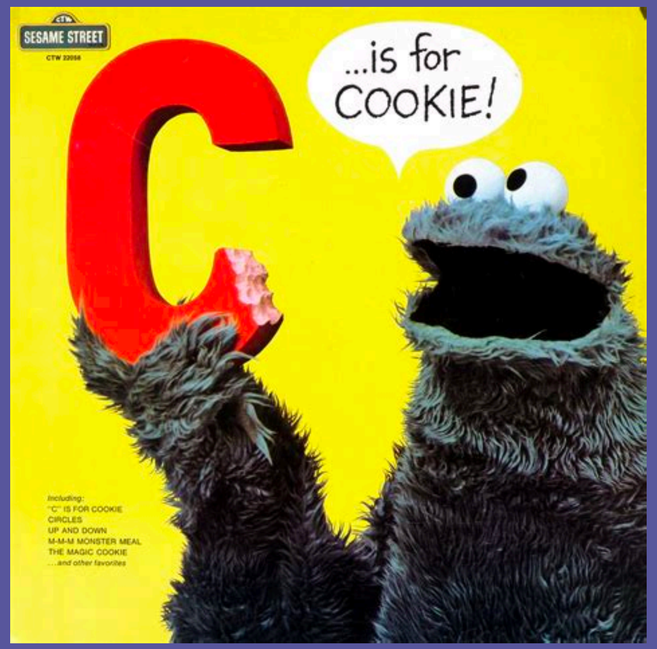
Language bindings

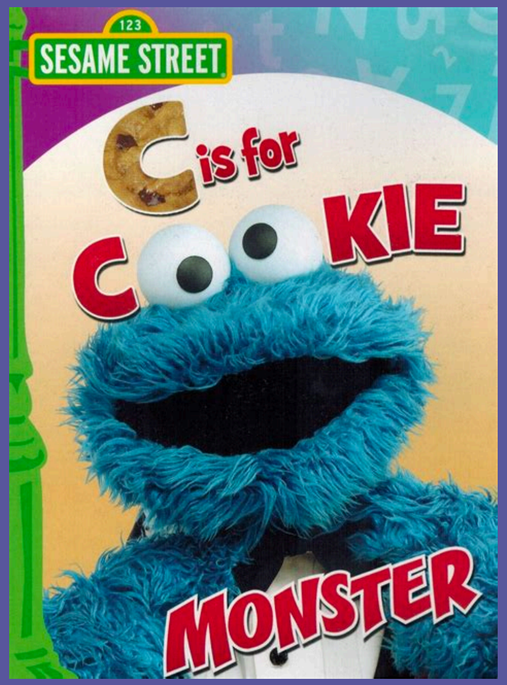
Why have I done this to myself?

- flox uses Rust
- flox uses Nix
- ???
- Profit?

Language bindings

- Language A
  - Language you're writing in
- Language B
  - Language you're binding to
- Mnemonic
  - B is for binding


SESAME STREET
...is for COOKIE!
Including:
"C" IS FOR COOKIE
CIRCLES
UP AND DOWN
M-M-M MONSTER MEAL
THE MAGIC COOKIE
...and other favorites


123
SESAME STREET
C is for COOKIE
MONSTER


LIFE
"C" IS FOR COOKIE
And that is quite satisfactory for my own personal obsessions


C IS FOR COOKIE
(that's good enough for me)


"C" is for COOKIE

Why write language bindings?

- The only implementation is written in Language B
- Language B is faster (e.g. Python -> C, Fortran, C++, Rust, etc)
- Compatibility
- Avoiding reinventing the wheel
- You're on a vision quest
- Language A good, Language B bad
- Pure, unadulterated hubris

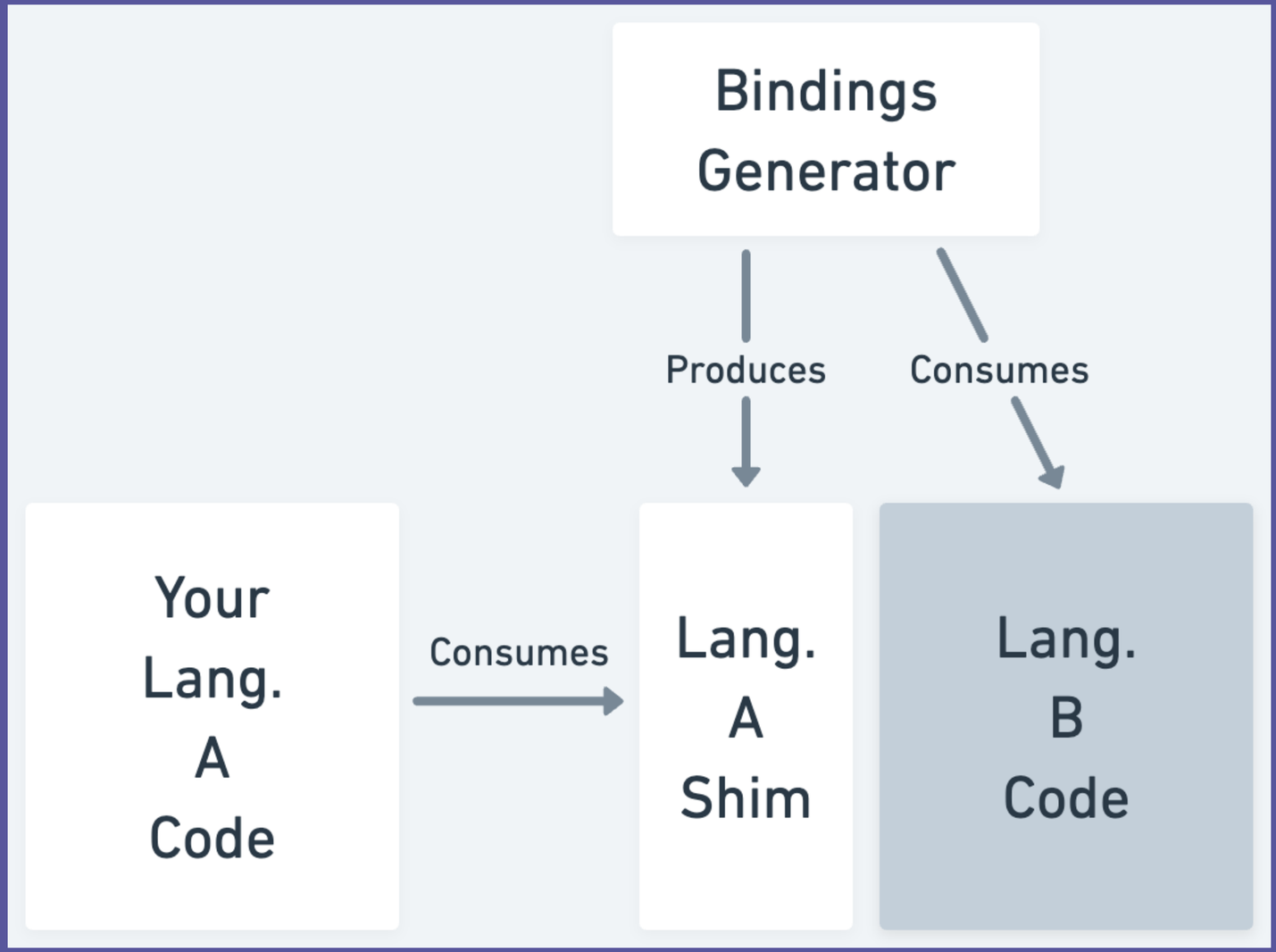
What makes this difficult?

Rust	C++
Traits	Inheritance
Generics	Templates
No defined ABI	Compiler defined ABI
Different type layouts in memory	
Different invariants that must be upheld	
Different ownership semantics	
Different vtables, inlining, moving, copying, etc	

Is it impossible?

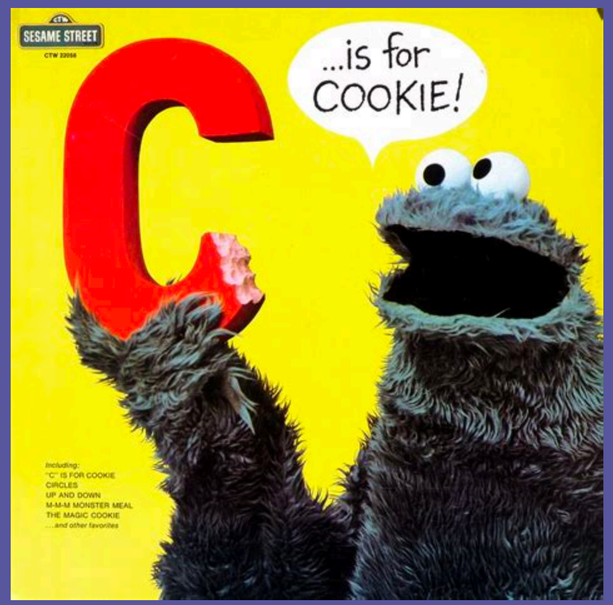
No!

Bindings generators

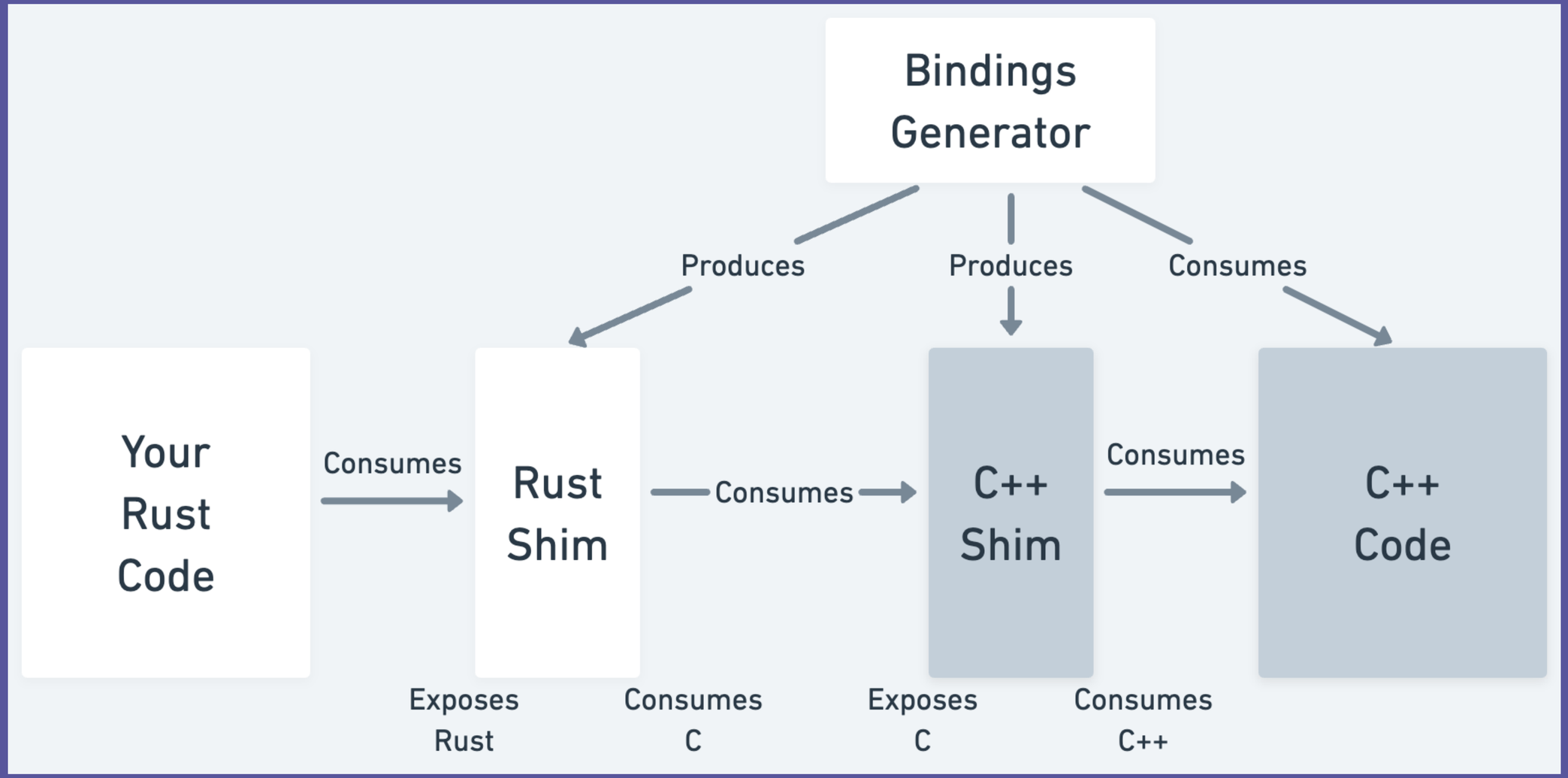


Bindings generators

- cxx
  - Generates bindings for manually specified types/functions
- autocxx
  - Tries to automatically generate bindings
- bindgen
  - Mostly for C code
- cbindgen
  - Generate a C interface for Rust code



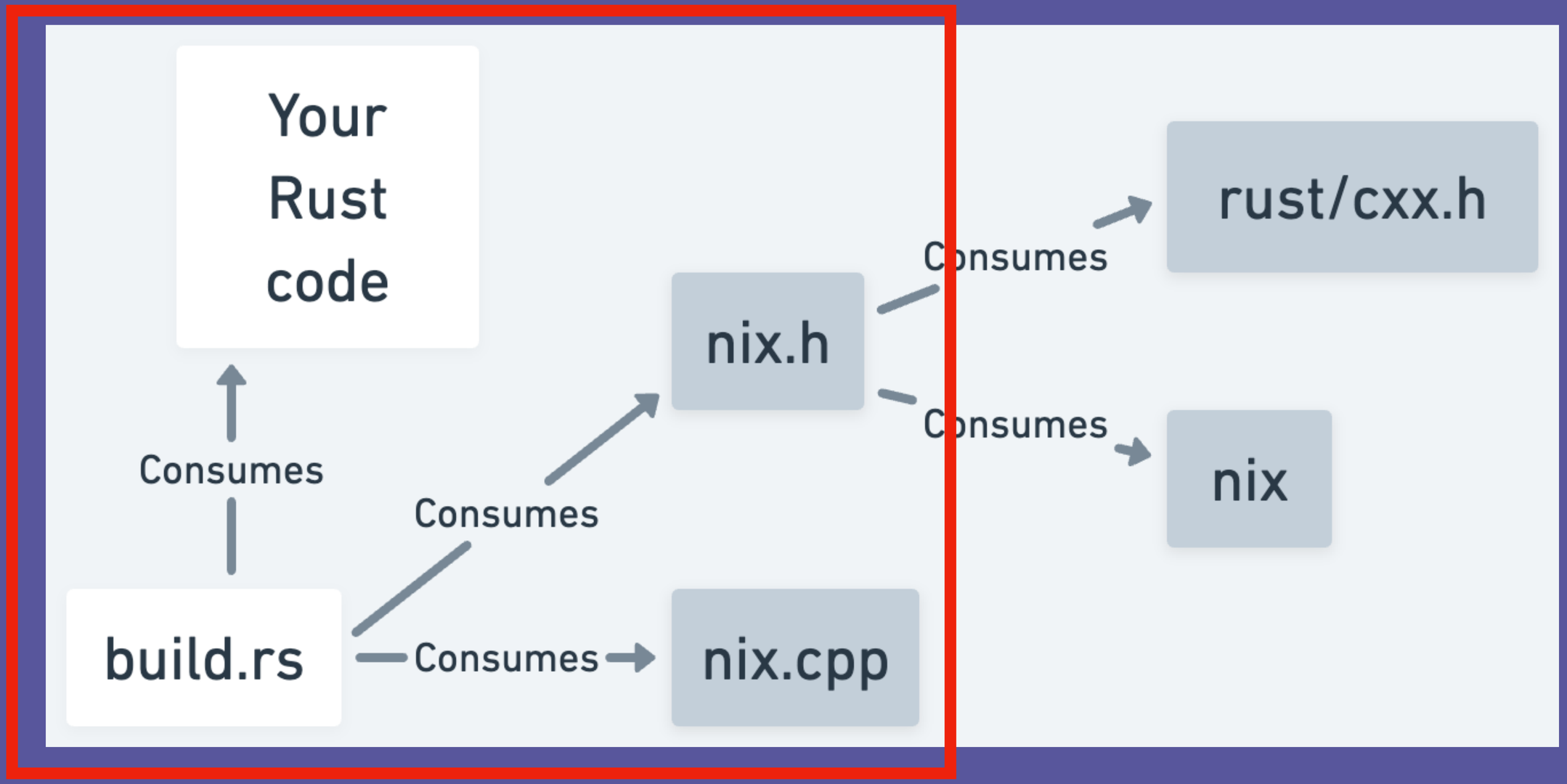
cxx


Bindings Generator
Produces
Produces
Consumes
Your Rust Code
Consumes
Rust Shim
Consumes
C++ Shim
Consumes
C++ Code
Exposes Rust
Consumes C
Exposes C
Consumes C++


So anyway, I started
binding
FX

Using cxx

You write this part



Let's parse a flake reference

```
FlakeRef parsed = parseFlakeRef(...);
cout << parsed.to_string() << endl;
```

Problem: opaque types

- Anything not "plain old data" gets turned into an opaque type
  - Even basic types like std::string are not "plain old data"
- You can't access member functions/attributes on opaque types
- Opaque types can't be returned directly
  - Must be wrapped in a pointer/reference

Consequence - so many wrappers

- You have to write wrappers to...
  - Return values behind pointers
  - Access member attributes
  - Call member functions

nix.h

```
#pragma once

#include "rust/cxx.h"
#include <nix/flake/flake.hh>
#include <nix/fetchers.hh>
#include <nix/error.hh>

namespace nix_cxx
{
  using nix::FlakeRef;

  std::unique_ptr<nix::FlakeRef> parse_flakeref(rust::Str url);
  rust::String flakeref_to_string(std::unique_ptr<nix::FlakeRef> flakeref);
}
```

nix.cpp

```cpp
#include "nix-cxx/include/nix.h"
#include <memory>

using namespace std;

namespace nix_cxx
{
  std::unique_ptr<nix::FlakeRef> parse_flakeref(rust::Str url)
  {
    nix::FlakeRef originalRef = nix::parseFlakeRef(string(url), nix::absPath("."));
    return std::make_unique<nix::FlakeRef>(originalRef);
  }

  rust::String flakeref_to_string(std::unique_ptr<nix::FlakeRef> flakeref)
  {
    auto s = flakeref->to_string();
    return rust::String(s);
  }
}
```

nix.rs

```rust
#[cxx::bridge(namespace = "nix_cxx")]
pub(crate) mod ffi {
    unsafe extern "C++" {
        include!("nix-cxx/include/nix.h");

        type FlakeRef;

        fn parse_flakeref(url: &str) -> UniquePtr<FlakeRef>;
        fn flakeref_to_string(flakeref: UniquePtr<FlakeRef>) -> String;
    }
}
```

main.rs

```rust
mod nix;

fn main() {
    let url = "github:NixOS/nixpkgs";
    let flake_ref = nix::ffi::parse_flakeref(url);
    let flake_ref_str = nix::ffi::flakeref_to_string(flake_ref);
    println!("{}", flake_ref_str);
}
```

nix-community/harmonia - libnixstore

- Binary cache implementation written in Rust
- Binds to Nix store APIs

nix-community/harmonia - libnixstore

```rust
fn query_raw_realisation(output_id: &str) -> Result<String>;
fn query_path_from_hash_part(hash_part: &str) -> Result<String>;
fn compute_fs_closure(
    flip_direction: bool,
    include_outputs: bool,
    paths: Vec<&str>,
) -> Result<Vec<String>>;
fn topo_sort_paths(paths: Vec<&str>) -> Result<Vec<String>>;
fn follow_links_to_store_path(path: &str) -> Result<String>;
fn export_paths(fd: i32, paths: Vec<&str>) -> Result<()>;
fn import_paths(fd: i32, dont_check_signs: bool) -> Result<()>;
fn hash_path(algo: &str, base32: bool, path: &str) -> Result<String>;
fn hash_file(algo: &str, base32: bool, path: &str) -> Result<String>;
fn hash_string(algo: &str, base32: bool, s: &str) -> Result<String>;
fn convert_hash(algo: &str, s: &str, to_base_32: bool) -> Result<String>;
fn sign_string(secret_key: &str, msg: &str) -> Result<String>;
fn check_signature(public_key: &str, sig: &str, msg: &str) -> Result<bool>;
fn add_to_store(src_path: &str, recursive: i32, algo: &str) -> Result<String>;
```

Bad or Rad?

"brad"

- Midjourney

Operation Cookie Monster

- Nix issue #7271 by @roberth
  - Language bindings and/or C interface for the Nix language
- Nix PR #8699 by @yorickvP
  - (Towards) stable C bindings for libutil, libexpr
- nix-python -> Python bindings
- Operation Cookie Monster -> Rust bindings

Expectations

- Quality?
  - Weekend project
- Stable?
  - lol no
  - Built against a PR, rebase could break things
- Rust-y-ness?
  - Mutability, mutability everywhere

Phase 1: Make it work

- Goal: eval an expression
- Roughly ~70 lines not including the build script
- One big "unsafe" block
- Go-style error handling
- Only the middle 1/3 is actually evaluating the expression

```rust
/// Evaluate a Nix expression
pub fn eval_string_raw_ffi(input: impl AsRef<str>) -> Result<String> {
    let expr_result = unsafe {
        // Initialize all the things
        let ctx = ffi::nix_c_context_create();
        ffi::nix_libexpr_init(ctx);
        let store_name = c_str_ptr("auto")?;
        let store = ffi::nix_store_open(ctx, store_name, null_mut::<*mut *const i8>());
        // Are we writing Go now?
        let err = ffi::nix_libexpr_init(ctx);
        if err != ffi::NIX_OK.try_into().unwrap() {
            return Err(anyhow!("couldn't initialize libexpr"))
                .with_context(|| format!("{}", err_msg(ctx)))?;
        }
        let state = ffi::nix_state_create(ctx, null_mut::<*const c_char>(), store);
        // You can now do stuff
        let expr_val = ffi::nix_alloc_value(ctx, state);
        let err = ffi::nix_expr_eval_from_string(
            ctx,
            state,
            c_str_ptr(input)?,
            c_str_ptr(".")?,
            expr_val,
        );
        if err != ffi::NIX_OK.try_into().unwrap() {
            return Err(anyhow!("{}", err_msg(ctx)));
        }
        let err = ffi::nix_value_force(ctx, state, expr_val);
        if err != ffi::NIX_OK.try_into().unwrap() {
            return Err(anyhow!("{}", err_msg(ctx)));
        }
        let expr_result = raw_str_to_string(ffi::nix_get_string(ctx, expr_val));
        // Clean up
        let err = ffi::nix_gc_decref(ctx, expr_val as *const c_void);
        if err != ffi::NIX_OK.try_into().unwrap() {
            return Err(anyhow!("{}", err_msg(ctx)));
        }
        ffi::nix_state_free(state);
        ffi::nix_store_unref(store);
        ffi::nix_c_context_free(ctx);
        expr_result
    };
    Ok(expr_result)
}
```

Phase 2: Make it good

```rust
/// A Nix value
#[derive(Debug)]
pub struct NixValue {
    inner: NonNull<ffi::Value>,
    ctx: Ctx,
}
```

Phase 2: Make it good

```rust
/// Create a new, unassigned value
pub fn new(ctx: Ctx, state: &mut NixState) -> Result<Self> {
    let val = unsafe { ffi::nix_alloc_value(ctx.borrow_mut().ptr(), state.ptr()) };
    if val.is_null() {
        bail!("nix_alloc_value returned a null pointer");
    }
    let val = NixValue {
        inner: NonNull::new(val).unwrap(),
        ctx: ctx.clone(),
    };
    Ok(val)
}
```

Phase 2: Make it good

```rust
/// Evaluate a Nix expression
pub fn eval_string(input: impl AsRef<str>) -> Result<String> {
    let ctx = ManuallyDrop::new(Rc::new(RefCell::new(NixContext::new()?)));
    let guard = NixInitGuard::new(ctx.clone())?;
    let mut store = NixStore::new(ctx.clone(), NixStoreType::Auto)?;
    let mut state = NixState::new(ctx.clone(), &mut store, guard)?;
    let mut value = NixValue::new_with_expr(ctx.clone(), &mut state, input)?;
    value.to_json(&mut state)
}
```

```
$ nix-bindgen -e "builtins.currentSystem"
"aarch64-darwin"
$ nix-bindgen default.nix
"/nix/store/xaca64zy77ck12qyvx8aj8ds9ynxjv5f-hello"
```

Problem: Rust destructors aren't fallible

- Rust destructors are implemented in the Drop trait
- Drop::drop has no return value, much less a Result/Error
- Decrementing the reference count on a Value can produce an error
- Solution:
  - 🤷🤷🤷🤷
  - Right now I'm just eating the error

Problem: carrying state around everywhere

- C++ can raise an exception at any time
- C++ exception in Rust = bad time
- Exceptions caught by the C bindings, stored in "nix_c_context"
  - Now everything needs "nix_c_context"

```
type Ctx = ManuallyDrop<Rc<RefCell<NixContext>>>;
```

Operation Cookie Monster - Takeaways

- Rust bindings to Nix are easy with a C API
- Mutable state needed by most calls
  - "nix_c_context" and "State" needed everywhere
- Having a C API is 👍👍👍

FIN

Resources

- Me
  - tinkering.xyz
  - https://hachyderm.io/@zmitchell
  - https://github.com/zmitchell
- Code
  - https://github.com/zmitchell/nix-ffi-rs
- Slides
  - https://github.com/zmitchell/talks/tree/master/2023-09-09%20-%20nixcon