Generated C++

```
#[cxx::bridge]
mod ffi {
    // Rust types and signatures exposed to C++.
    extern "Rust" {
        type MultiBuf;

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

Results in (roughly) the following C++:

```
struct MultiBuf final : public ::rust::Opaque {
  ~MultiBuf() = delete;

private:
  friend ::rust::layout;
  struct layout {
    static ::std::size_t size() noexcept;
    static ::std::size_t align() noexcept;
  };
};

::rust::Slice<::std::uint8_t const> next_chunk(::org::blobstore::MultiBuf &buf)
noexcept;
```

C++ Bridge Declarations

```
#[cxx::bridge]
mod ffi {
    // C++ types and signatures exposed to Rust.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

Results in (roughly) the following Rust:

```
#[repr(C)]
pub struct BlobstoreClient {
    _private: ::cxx::private::Opaque,
}

pub fn new_blobstore_client() -> ::cxx::UniquePtr<BlobstoreClient> {
    extern "C" {
        #[link_name = "org$blobstore$cxxbridge1$new_blobstore_client"]
        fn __new_blobstore_client() -> *mut BlobstoreClient;
    }
    unsafe { ::cxx::UniquePtr::from_raw(__new_blobstore_client()) }
}

impl BlobstoreClient {
    pub fn put(&self, parts: &mut MultiBuf) -> u64 {
        extern "C" {
            #[link_name = "org$blobstore$cxxbridge1$BlobstoreClient$put"]
            fn __put(
                _: &BlobstoreClient,
                parts: *mut ::cxx::core::ffi::c_void,
            ) -> u64;
        }
        unsafe {
            __put(self, parts as *mut MultiBuf as *mut ::cxx::core::ffi::c_void)
        }
    }
}

// ...
```

▼ *Speaker Notes*

- The programmer does not need to promise that the signatures they have typed in are accurate. CXX performs static assertions that the signatures exactly correspond with what is declared in C++.
- `unsafe extern` blocks allow you to declare C++ functions that are safe to call from Rust.

Shared Types

```
#[cxx::bridge]
mod ffi {
    #[derive(Clone, Debug, Hash)]
    struct PlayingCard {
        suit: Suit,
        value: u8,  // A=1, J=11, Q=12, K=13
    }

    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

▼ *Speaker Notes*

- Only C-like (unit) enums are supported.
- A limited number of traits are supported for `#[derive()]` on shared types. Corresponding functionality is also generated for the C++ code, e.g. if you derive `Hash` also generates an implementation of `std::hash` for the corresponding C++ type.

Shared Enums

```
#[cxx::bridge]
mod ffi {
    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

Generated Rust:

```
#[derive(Copy, Clone, PartialEq, Eq)]
#[repr(transparent)]
pub struct Suit {
    pub repr: u8,
}

#[allow(non_upper_case_globals)]
impl Suit {
    pub const Clubs: Self = Suit { repr: 0 };
    pub const Diamonds: Self = Suit { repr: 1 };
    pub const Hearts: Self = Suit { repr: 2 };
    pub const Spades: Self = Suit { repr: 3 };
}
```

Generated C++:

```
enum class Suit : uint8_t {
  Clubs = 0,
  Diamonds = 1,
  Hearts = 2,
  Spades = 3,
};
```

▼ *Speaker Notes*

- On the Rust side, the code generated for shared enums is actually a struct wrapping a numeric value. This is because it is not UB in C++ for an enum class to hold a value different from all of the listed variants, and our Rust representation needs to have the same behavior.

Rust Error Handling

```rust
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn fallible(depth: usize) -> anyhow::Result<String> {
    if depth == 0 {
        return Err(anyhow::Error::msg("fallible1 requires depth > 0"));
    }

    Ok("Success!".into())
}
```

▼ *Speaker Notes*

- Rust functions that return `Result` are translated to exceptions on the C++ side.
- The exception thrown will always be of type `rust::Error`, which primarily exposes a way to get the error message string. The error message will come from the error type's `Display` impl.
- A panic unwinding from Rust to C++ will always cause the process to immediately terminate.

C++ Error Handling

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("example/include/example.h");
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn main() {
    if let Err(err) = ffi::fallible(99) {
        eprintln!("Error: {}", err);
        process::exit(1);
    }
}
```

▼ *Speaker Notes*

- C++ functions declared to return a `Result` will catch any thrown exception on the C++ side and return it as an `Err` value to the calling Rust function.
- If an exception is thrown from an extern "C++" function that is not declared by the CXX bridge to return `Result`, the program calls C++'s `std::terminate`. The behavior is equivalent to the same exception being thrown through a `noexcept` C++ function.

Additional Types

Rust Type	C++ Type
`String`	`rust::String`
`&str`	`rust::Str`
`CxxString`	`std::string`
`&[T]` / `&mut [T]`	`rust::Slice`
`Box<T>`	`rust::Box<T>`
`UniquePtr<T>`	`std::unique_ptr<T>`
`Vec<T>`	`rust::Vec<T>`
`CxxVector<T>`	`std::vector<T>`

▼ *Speaker Notes*

- These types can be used in the fields of shared structs and the arguments and returns of extern functions.
- Note that Rust's `String` does not map directly to `std::string`. There are a few reasons for this:
  - `std::string` does not uphold the UTF-8 invariant that `String` requires.
  - The two types have different layouts in memory and so can't be passed directly between languages.
  - `std::string` requires move constructors that don't match Rust's move semantics, so a `std::string` can't be passed by value to Rust.

Building in Android

Create two genrules: One to generate the CXX header, and one to generate the CXX source file. These are then used as inputs to the `cc_library_static`.

```
// Generate a C++ header containing the C++ bindings
// to the Rust exported functions in lib.rs.
genrule {
    name: "libcxx_test_bridge_header",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) --header > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.h"],
}

// Generate the C++ code that Rust calls into.
genrule {
    name: "libcxx_test_bridge_code",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.cc"],
}
```

▼ *Speaker Notes*

- The `cxxbridge` tool is a standalone tool that generates the C++ side of the bridge module. It is included in Android and available as a Soong tool.
- By convention, if your Rust source file is `lib.rs` your header file will be named `lib.rs.h` and your source file will be named `lib.rs.cc`. This naming convention isn't enforced, though.

Building in Android

Create a `cc_library_static` to build the C++ library, including the CXX generated header and source file.

```
cc_library_static {
    name: "libcxx_test_cpp",
    srcs: ["cxx_test.cpp"],
    generated_headers: [
        "cxx-bridge-header",
        "libcxx_test_bridge_header"
    ],
    generated_sources: ["libcxx_test_bridge_code"],
}
```

▼ *Speaker Notes*

- Point out that `libcxx_test_bridge_header` and `libcxx_test_bridge_code` are the dependencies for the CXX-generated C++ bindings. We'll show how these are setup on the next slide.
- Note that you also need to depend on the `cxx-bridge-header` library in order to pull in common CXX definitions.
- Full docs for using CXX in Android can be found in the Android docs. You may want to share that link with the class so that students know where they can find these instructions again in the future.

Building in Android

Create a `rust_binary` that depends on `libcxx` and your `cc_library_static`.

```
rust_binary {
    name: "cxx_test",
    srcs: ["lib.rs"],
    rustlibs: ["libcxx"],
    static_libs: ["libcxx_test_cpp"],
}
```

Interoperability with Java

Java can load shared objects via Java Native Interface (JNI). The `jni` crate allows you to create a compatible library.

First, we create a Rust function to export to Java:

interoperability/java/src/lib.rs:

```
//! Rust <-> Java FFI demo.

use jni::JNIEnv;
use jni::objects::{JClass, JString};
use jni::sys::jstring;

/// HelloWorld::hello method implementation.
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
pub extern "system" fn Java_HelloWorld_hello(
    mut env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
    let input: String = env.get_string(&name).unwrap().into();
    let greeting = format!("Hello, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_raw()
}
```

interoperability/java/Android.bp:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

We then call this function from Java:

interoperability/java/HelloWorld.java:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

interoperability/java/Android.bp:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
    jni_libs: ["libhello_jni"],
```

Finally, you can build, sync, and run the binary:

```
m helloworld_jni
adb sync  # requires adb root && adb remount
adb shell /system/bin/helloworld_jni
```

▼ *Speaker Notes*

- The `unsafe(no_mangle)` attribute instructs Rust to emit the `Java_HelloWorld_hello` symbol exactly as written. This is important so that Java can recognize the symbol as a `hello` method on the `HelloWorld` class.
    - By default, Rust will mangle (rename) symbols so that a binary can link in two versions of the same Rust crate.

Welcome to Rust in Chromium

Rust is supported for third-party libraries in Chromium, with first-party glue code to connect between Rust and existing Chromium C++ code.

> Today, we'll call into Rust to do something silly with strings. If you've got a corner of the code where you're displaying a UTF8 string to the user, feel free to follow this recipe in your part of the codebase instead of the exact part we talk about.

Setup

Make sure you can build and run Chromium. Any platform and set of build flags is OK, so long as your code is relatively recent (commit position 1223636 onwards, corresponding to November 2023):

```
gn gen out/Debug
autoninja -C out/Debug chrome
out/Debug/chrome # or on Mac, out/Debug/Chromium.app/Contents/MacOS/Chromium
```

(A component, debug build is recommended for quickest iteration time. This is the default!)

See How to build Chromium if you aren't already at that point. Be warned: setting up to build Chromium takes time.

It's also recommended that you have Visual Studio code installed.

About the exercises

This part of the course has a series of exercises which build on each other. We'll be doing them spread throughout the course instead of just at the end. If you don't have time to complete a certain part, don't worry: you can catch up in the next slot.

Comparing Chromium and Cargo Ecosystems

The Rust community typically uses `cargo` and libraries from crates.io. Chromium is built using `gn` and `ninja` and a curated set of dependencies.

When writing code in Rust, your choices are:

- Use `gn` and `ninja` with the help of the templates from `//build/rust/*.gni` (e.g. `rust_static_library` that we'll meet later). This uses Chromium's audited toolchain and crates.
- Use `cargo`, but restrict yourself to Chromium's audited toolchain and crates
- Use `cargo`, trusting a toolchain and/or crates downloaded from the internet

From here on we'll be focusing on `gn` and `ninja`, because this is how Rust code can be built into the Chromium browser. At the same time, Cargo is an important part of the Rust ecosystem and you should keep it in your toolbox.

Mini exercise

Split into small groups and:

- Brainstorm scenarios where `cargo` may offer an advantage and assess the risk profile of these scenarios.
- Discuss which tools, libraries, and groups of people need to be trusted when using `gn` and `ninja`, offline `cargo`, etc.

▼ *Speaker Notes*

Ask students to avoid peeking at the speaker notes before completing the exercise. Assuming folks taking the course are physically together, ask them to discuss in small groups of 3-4 people.

Notes/hints related to the first part of the exercise ("scenarios where Cargo may offer an advantage"):

- It's fantastic that when writing a tool, or prototyping a part of Chromium, one has access to the rich ecosystem of crates.io libraries. There is a crate for almost anything and they are usually quite pleasant to use. (`clap` for command-line parsing, `serde` for serializing/deserializing to/from various formats, `itertools` for working with iterators, etc.).
  - `cargo` makes it easy to try a library (just add a single line to `Cargo.toml` and start writing code)
  - It may be worth comparing how CPAN helped make `perl` a popular choice. Or comparing with `python` + `pip`.
- Development experience is made really nice not only by core Rust tools (e.g. using `rustup` to switch to a different `rustc` version when testing a crate that needs to work on nightly, current stable, and older stable) but also by an ecosystem of third-party tools (e.g. Mozilla provides `cargo vet` for streamlining and sharing security audits; `criterion` crate gives a streamlined way to run benchmarks).
  - `cargo` makes it easy to add a tool via `cargo install --locked cargo-vet`.
  - It may be worth comparing with Chrome Extensions or VScode extensions.

    - Perhaps surprisingly, Rust is becoming increasingly popular in the industry for writing command line tools. The breadth and ergonomics of libraries is comparable to Python, while being more robust (thanks to the rich typesystem) and running faster (as a compiled, rather than interpreted language).
    - Participating in the Rust ecosystem requires using standard Rust tools like Cargo. Libraries that want to get external contributions, and want to be used outside of Chromium (e.g. in Bazel or Android/Soong build environments) should probably use Cargo.

- Examples of Chromium-related projects that are `cargo`-based:
    - `serde_json_lenient` (experimented with in other parts of Google which resulted in PRs with performance improvements)
    - Fontations libraries like `font-types`
    - `gnrt` tool (we will meet it later in the course) which depends on `clap` for command-line parsing and on `toml` for configuration files.
        - Disclaimer: a unique reason for using `cargo` was unavailability of `gn` when building and bootstrapping Rust standard library when building Rust toolchain.
        - `run_gnrt.py` uses Chromium's copy of `cargo` and `rustc`. `gnrt` depends on third-party libraries downloaded from the internet, but `run_gnrt.py` asks `cargo` that only `--locked` content is allowed via `Cargo.lock`.)

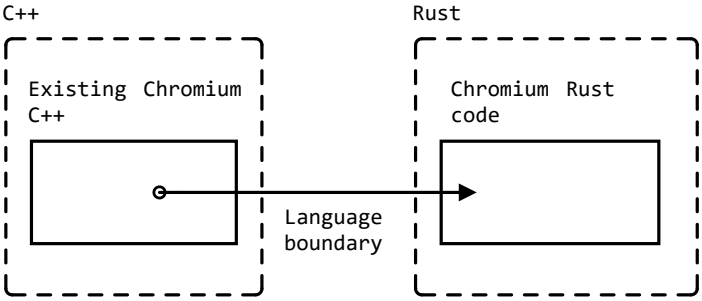
Students may identify the following items as being implicitly or explicitly trusted:

- `rustc` (the Rust compiler) which in turn depends on the LLVM libraries, the Clang compiler, the `rustc` sources (fetched from GitHub, reviewed by Rust compiler team), binary Rust compiler downloaded for bootstrapping
- `rustup` (it may be worth pointing out that `rustup` is developed under the umbrella of the https://github.com/rust-lang/ organization - same as `rustc`)
- `cargo`, `rustfmt`, etc.
- Various internal infrastructure (bots that build `rustc`, system for distributing the prebuilt toolchain to Chromium engineers, etc.)
- Cargo tools like `cargo audit`, `cargo vet`, etc.
- Rust libraries vendored into `//third_party/rust` (audited by security@chromium.org)
- Other Rust libraries (some niche, some quite popular and commonly used)

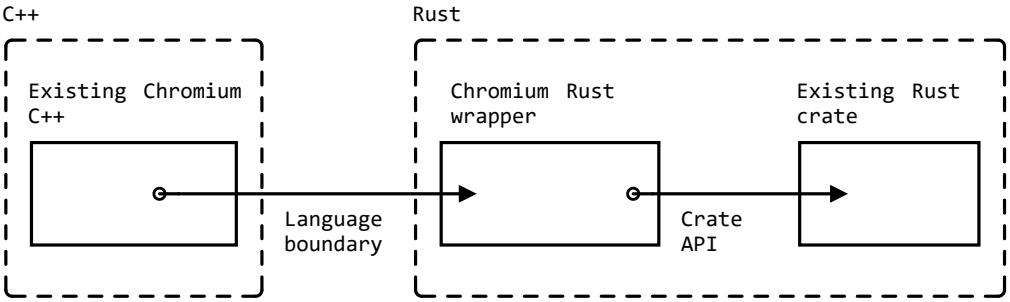
Chromium Rust policy

Chromium's Rust policy can be found here. Rust can be used for both first-party and third-party code.

Using Rust for pure first-party code looks like this:



The third-party case is also common. It's likely that you'll also need a small amount of first-party glue code, because very few Rust libraries directly expose a C/C++ API.



The scenario of using a third-party crate is the more complex one, so today's course will focus on:

- Bringing in third-party Rust libraries ("crates")
- Writing glue code to be able to use those crates from Chromium C++. (The same techniques are used when working with first-party Rust code).

Build rules

Rust code is usually built using `cargo`. Chromium builds with `gn` and `ninja` for efficiency — its static rules allow maximum parallelism. Rust is no exception.

Adding Rust code to Chromium

In some existing Chromium `BUILD.gn` file, declare a `rust_static_library`:

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}
```

You can also add `deps` on other Rust targets. Later we'll use this to depend upon third party code.

▼ *Speaker Notes*

You must specify *both* the crate root, *and* a full list of sources. The `crate_root` is the file given to the Rust compiler representing the root file of the compilation unit — typically `lib.rs`. `sources` is a complete list of all source files which `ninja` needs in order to determine when rebuilds are necessary.

(There's no such thing as a Rust `source_set`, because in Rust, an entire crate is a compilation unit. A `static_library` is the smallest unit.)

Students might be wondering why we need a gn template, rather than using gn's built-in support for Rust static libraries. The answer is that this template provides support for CXX interop, Rust features, and unit tests, some of which we'll use later.

Including unsafe Rust Code

Unsafe Rust code is forbidden in `rust_static_library` by default — it won't compile. If you need unsafe Rust code, add `allow_unsafe = true` to the gn target. (Later in the course we'll see circumstances where this is necessary.)

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [
    "lib.rs",
    "hippopotamus.rs"
  ]
  allow_unsafe = true
}
```

Depending on Rust Code from Chromium C++

Simply add the above target to the `deps` of some Chromium C++ target.

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}

# or source_set, static_library etc.
component("preexisting_cpp") {
  deps = [ ":my_rust_lib" ]
}
```

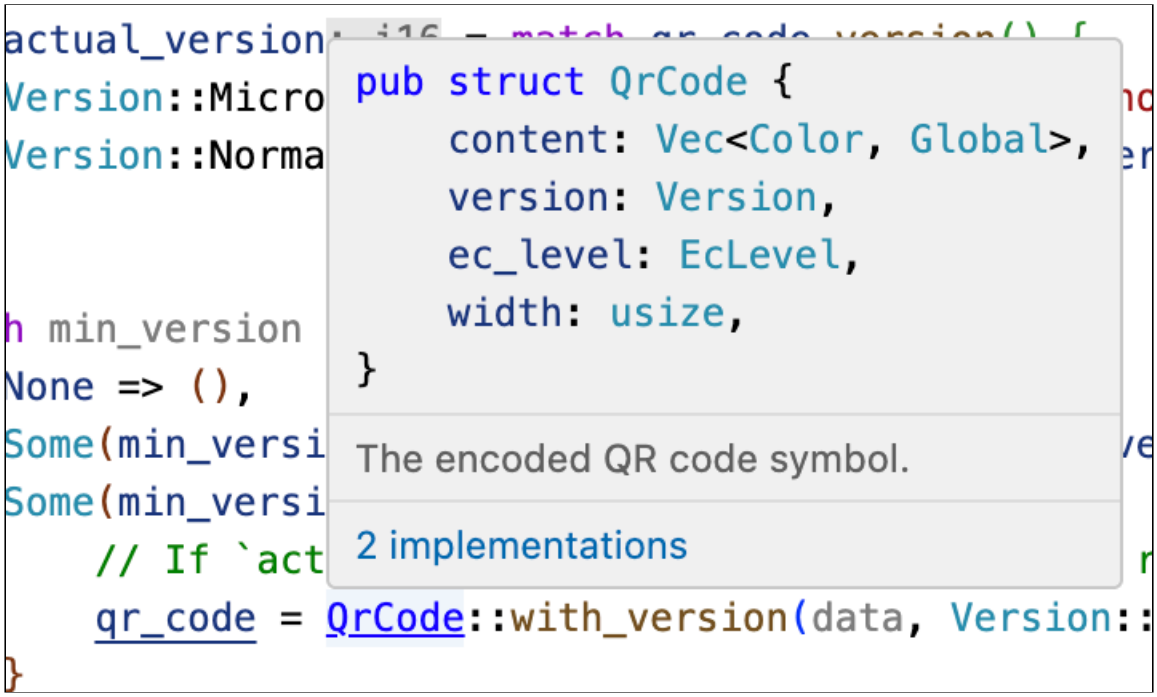
▼ *Speaker Notes*

We'll see that this relationship only works if the Rust code exposes plain C APIs which can be called from C++, or if we use a C++/Rust interop tool.

Visual Studio Code

Types are elided in Rust code, which makes a good IDE even more useful than for C++. Visual Studio code works well for Rust in Chromium. To use it,

- Ensure your VSCode has the `rust-analyzer` extension, not earlier forms of Rust support
- `gn gen out/Debug --export-rust-project` (or equivalent for your output directory)
- `ln -s out/Debug/rust-project.json rust-project.json`



▼ *Speaker Notes*

A demo of some of the code annotation and exploration features of rust-analyzer might be beneficial if the audience are naturally skeptical of IDEs.

The following steps may help with the demo (but feel free to instead use a piece of Chromium-related Rust that you are most familiar with):

- Open `components/qr_code_generator/qr_code_generator_ffi_glue.rs`
- Place the cursor over the `QrCode::new` call (around line 26) in `qr_code_generator_ffi_glue.rs
- Demo **show documentation** (typical bindings: vscode = ctrl k i; vim/CoC = K).
- Demo **go to definition** (typical bindings: vscode = F12; vim/CoC = g d). (This will take you to `//third_party/rust/.../qr_code-.../src/lib.rs`.)
- Demo **outline** and navigate to the `QrCode::with_bits` method (around line 164; the outline is in the file explorer pane in vscode; typical vim/CoC bindings = space o)
- Demo **type annotations** (there are quite a few nice examples in the `QrCode::with_bits` method)

It may be worth pointing out that `gn gen ... --export-rust-project` will need to be rerun after editing `BUILD.gn` files (which we will do a few times throughout the exercises in this session).

Build rules exercise

In your Chromium build, add a new Rust target to `//ui/base/BUILD.gn` containing:

```
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
pub extern "C" fn hello_from_rust() {
    println!("Hello from Rust!")
}
```

Important: note that `no_mangle` here is considered a type of unsafety by the Rust compiler, so you'll need to allow unsafe code in your `gn` target.

Add this new Rust target as a dependency of `//ui/base:base`. Declare this function at the top of `ui/base/resource/resource_bundle.cc` (later, we'll see how this can be automated by bindings generation tools):

```
extern "C" void hello_from_rust();
```

Call this function from somewhere in `ui/base/resource/resource_bundle.cc` - we suggest the top of `ResourceBundle::MaybeMangleLocalizedString`. Build and run Chromium, and ensure that "Hello from Rust!" is printed lots of times.

If you use VSCode, now set up Rust to work well in VSCode. It will be useful in subsequent exercises. If you've succeeded, you will be able to use right-click "Go to definition" on `println!`.

Where to find help

- The options available to the `rust_static_library` gn template
- Information about `#[unsafe(no_mangle)]`
- Information about `extern "C"`
- Information about gn's `--export-rust-project` switch
- How to install rust-analyzer in VSCode

▼ *Speaker Notes*

It's really important that students get this running, because future exercises will build on it.

This example is unusual because it boils down to the lowest-common-denominator interop language, C. Both C++ and Rust can natively declare and call C ABI functions. Later in the course, we'll connect C++ directly to Rust.

`allow_unsafe = true` is required here because `#[unsafe(no_mangle)]` might allow Rust to generate two functions with the same name, and Rust can no longer guarantee that the right one is called.

If you need a pure Rust executable, you can also do that using the `rust_executable` gn template.

Testing

Rust community typically authors unit tests in a module placed in the same source file as the code being tested. This was covered earlier in the course and looks like this:

```
#[cfg(test)]
mod tests {
    #[test]
    fn my_test() {
        todo!()
    }
}
```

In Chromium we place unit tests in a separate source file and we continue to follow this practice for Rust — this makes tests consistently discoverable and helps to avoid rebuilding `.rs` files a second time (in the `test` configuration).

This results in the following options for testing Rust code in Chromium:

- Native Rust tests (i.e. `#[test]`). Discouraged outside of `//third_party/rust`.
- `gtest` tests authored in C++ and exercising Rust via FFI calls. Sufficient when Rust code is just a thin FFI layer and the existing unit tests provide sufficient coverage for the feature.
- `gtest` tests authored in Rust and using the crate under test through its public API (using `pub mod for_testing { ... }` if needed). This is the subject of the next few slides.

▼ *Speaker Notes*

Mention that native Rust tests of third-party crates should eventually be exercised by Chromium bots. (Such testing is needed rarely — only after adding or updating third-party crates.)

Some examples may help illustrate when C++ `gtest` vs Rust `gtest` should be used:

- QR has very little functionality in the first-party Rust layer (it's just a thin FFI glue) and therefore uses the existing C++ unit tests for testing both the C++ and the Rust implementation (parameterizing the tests so they enable or disable Rust using a `ScopedFeatureList`).
- Hypothetical/WIP PNG integration may need to implement memory-safe implementation of pixel transformations that are provided by `libpng` but missing in the `png` crate - e.g. RGBA => BGRA, or gamma correction. Such functionality may benefit from separate tests authored in Rust.

`rust_gtest_interop` Library

The `rust_gtest_interop` library provides a way to:

- Use a Rust function as a `gtest` testcase (using the `#[gtest(...)]` attribute)
- Use `expect_eq!` and similar macros (similar to `assert_eq!` but not panicking and not terminating the test when the assertion fails).

Example:

```
use rust_gtest_interop::prelude::*;

#[gtest(MyRustTestSuite, MyAdditionTest)]
fn test_addition() {
    expect_eq!(2 + 2, 4);
}
```

GN Rules for Rust Tests

The simplest way to build Rust `gtest` tests is to add them to an existing test binary that already contains tests authored in C++. For example:

```
test("ui_base_unittests") {
  ...
  sources += [ "my_rust_lib_unittest.rs" ]
  deps += [ ":my_rust_lib" ]
}
```

Authoring Rust tests in a separate `static_library` also works, but requires manually declaring the dependency on the support libraries:

```
rust_static_library("my_rust_lib_unittests") {
  testonly = true
  is_gtest_unittests = true
  crate_root = "my_rust_lib_unittest.rs"
  sources = [ "my_rust_lib_unittest.rs" ]
  deps = [
    ":my_rust_lib",
    "//testing/rust_gtest_interop",
  ]
}

test("ui_base_unittests") {
  ...
  deps += [ ":my_rust_lib_unittests" ]
}
```

`chromium::import!` Macro

After adding `:my_rust_lib` to GN `deps`, we still need to learn how to import and use `my_rust_lib` from `my_rust_lib_unittest.rs`. We haven't provided an explicit `crate_name` for `my_rust_lib` so its crate name is computed based on the full target path and name. Fortunately we can avoid working with such an unwieldy name by using the `chromium::import!` macro from the automatically-imported `chromium` crate:

```
chromium::import! {
    "//ui/base:my_rust_lib";
}

use my_rust_lib::my_function_under_test;
```

Under the covers the macro expands to something similar to:

```
extern crate ui_sbase_cmy_urust_ulib as my_rust_lib;

use my_rust_lib::my_function_under_test;
```

More information can be found in the doc comment of the `chromium::import` macro.

▼ *Speaker Notes*

`rust_static_library` supports specifying an explicit name via `crate_name` property, but doing this is discouraged. And it is discouraged because the crate name has to be globally unique. crates.io guarantees uniqueness of its crate names so `cargo_crate` GN targets (generated by the `gnrt` tool covered in a later section) use short crate names.

Testing exercise

Time for another exercise!

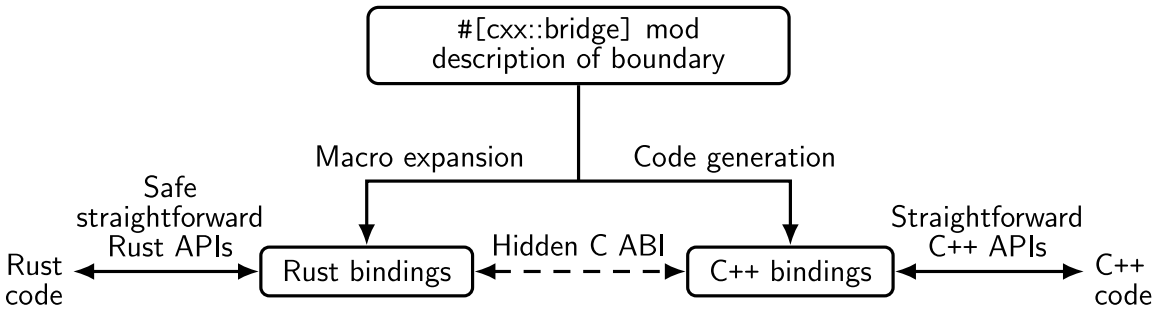
In your Chromium build:

- Add a testable function next to `hello_from_rust`. Some suggestions: adding two integers received as arguments, computing the nth Fibonacci number, summing integers in a slice, etc.
- Add a separate `..._unittest.rs` file with a test for the new function.
- Add the new tests to `BUILD.gn`.
- Build the tests, run them, and verify that the new test works.

Interoperability with C++

The Rust community offers multiple options for C++/Rust interop, with new tools being developed all the time. At the moment, Chromium uses a tool called CXX.

You describe your whole language boundary in an interface definition language (which looks a lot like Rust) and then CXX tools generate declarations for functions and types in both Rust and C++.



See the CXX tutorial for a full example of using this.

▼ *Speaker Notes*

Talk through the diagram. Explain that behind the scenes, this is doing just the same as you previously did. Point out that automating the process has the following benefits:

- The tool guarantees that the C++ and Rust sides match (e.g. you get compile errors if the `#[cxx::bridge]` doesn't match the actual C++ or Rust definitions, but with out-of-sync manual bindings you'd get Undefined Behavior)
- The tool automates generation of FFI thunks (small, C-ABI-compatible, free functions) for non-C features (e.g. enabling FFI calls into Rust or C++ methods; manual bindings would require authoring such top-level, free functions manually)
- The tool and the library can handle a set of core types - for example:
  - `&[T]` can be passed across the FFI boundary, even though it doesn't guarantee any particular ABI or memory layout. With manual bindings `std::span<T>` / `&[T]` have to be manually destructured and rebuilt out of a pointer and length - this is error-prone given that each language represents empty slices slightly differently)
  - Smart pointers like `std::unique_ptr<T>`, `std::shared_ptr<T>`, and/or `Box` are natively supported. With manual bindings, one would have to pass C-ABI-compatible raw pointers, which would increase lifetime and memory-safety risks.
  - `rust::String` and `CxxString` types understand and maintain differences in string representation across the languages (e.g. `rust::String::lossy` can build a Rust string from non-UTF8 input and `rust::String::c_str` can NUL-terminate a string).

Example Bindings

CXX requires that the whole C++/Rust boundary is declared in `cxx::bridge` modules inside `.rs` source code.

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MultiBuf;

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

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

        type BlobstoreClient;

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

// Definitions of Rust types and functions go here
```

▼ *Speaker Notes*

Point out:

- Although this looks like a regular Rust `mod`, the `#[cxx::bridge]` procedural macro does complex things to it. The generated code is quite a bit more sophisticated - though this does still result in a `mod` called `ffi` in your code.
- Native support for C++'s `std::unique_ptr` in Rust
- Native support for Rust slices in C++
- Calls from C++ to Rust, and Rust types (in the top part)
- Calls from Rust to C++, and C++ types (in the bottom part)

Common misconception: It *looks* like a C++ header is being parsed by Rust, but this is misleading. This header is never interpreted by Rust, but simply `#include`d in the generated C++ code for the benefit of C++ compilers.

Limitations of CXX

By far the most useful page when using CXX is the type reference.

CXX fundamentally suits cases where:

- Your Rust-C++ interface is sufficiently simple that you can declare all of it.
- You're using only the types natively supported by CXX already, for example `std::unique_ptr`, `std::string`, `&[u8]` etc.

It has many limitations — for example lack of support for Rust's `Option` type.

These limitations constrain us to using Rust in Chromium only for well isolated "leaf nodes" rather than for arbitrary Rust-C++ interop. When considering a use-case for Rust in Chromium, a good starting point is to draft the CXX bindings for the language boundary to see if it appears simple enough.

▼ *Speaker Notes*

In addition, right now, Rust code in one component cannot depend on Rust code in another, due to linking details in our component build. That's another reason to restrict Rust to use in leaf nodes.

You should also discuss some of the other sticky points with CXX, for example:

- Its error handling is based around C++ exceptions (given on the next slide)
- Function pointers are awkward to use.

CXX Error Handling

CXX's support for `Result<T,E>` relies on C++ exceptions, so we can't use that in Chromium. Alternatives:

- The `T` part of `Result<T, E>` can be:
  - Returned via out parameters (e.g. via `&mut T`). This requires that `T` can be passed across the FFI boundary - for example `T` has to be:
    - A primitive type (like `u32` or `usize`)
    - A type natively supported by `cxx` (like `UniquePtr<T>`) that has a suitable default value to use in a failure case (*unlike* `Box<T>`).
  - Retained on the Rust side, and exposed via reference. This may be needed when `T` is a Rust type, which cannot be passed across the FFI boundary, and cannot be stored in `UniquePtr<T>`.
- The `E` part of `Result<T, E>` can be:
  - Returned as a boolean (e.g. `true` representing success, and `false` representing failure)
  - Preserving error details is in theory possible, but so far hasn't been needed in practice.

CXX Error Handling: QR Example

The QR code generator is an example where a boolean is used to communicate success vs failure, and where the successful result can be passed across the FFI boundary:

```
#[cxx::bridge(namespace = "qr_code_generator")]
mod ffi {
    extern "Rust" {
        fn generate_qr_code_using_rust(
            data: &[u8],
            min_version: i16,
            out_pixels: Pin<&mut CxxVector<u8>>,
            out_qr_size: &mut usize,
        ) -> bool;
    }
}
```

▼ *Speaker Notes*

Students may be curious about the semantics of the `out_qr_size` output. This is not the size of the vector, but the size of the QR code (and admittedly it is a bit redundant - this is the square root of the size of the vector).

It may be worth pointing out the importance of initializing `out_qr_size` before calling into the Rust function. Creation of a Rust reference that points to uninitialized memory results in Undefined Behavior (unlike in C++, when only the act of dereferencing such memory results in UB).

If students ask about `Pin`, then explain why CXX needs it for mutable references to C++ data: the answer is that C++ data can't be moved around like Rust data, because it may contain self-referential pointers.

CXX Error Handling: PNG Example

A prototype of a PNG decoder illustrates what can be done when the successful result cannot be passed across the FFI boundary:

```
#[cxx::bridge(namespace = "gfx::rust_bindings")]
mod ffi {
    extern "Rust" {
        /// This returns an FFI-friendly equivalent of `Result<PngReader<'a>,
        /// ()>`.
        fn new_png_reader<'a>(input: &'a [u8]) -> Box<ResultOfPngReader<'a>>;

        /// C++ bindings for the `crate::png::ResultOfPngReader` type.
        type ResultOfPngReader<'a>;
        fn is_err(self: &ResultOfPngReader) -> bool;
        fn unwrap_as_mut<'a, 'b>(
            self: &'b mut ResultOfPngReader<'a>,
        ) -> &'b mut PngReader<'a>;

        /// C++ bindings for the `crate::png::PngReader` type.
        type PngReader<'a>;
        fn height(self: &PngReader) -> u32;
        fn width(self: &PngReader) -> u32;
        fn read_rgba8(self: &mut PngReader, output: &mut [u8]) -> bool;
    }
}
```

▼ *Speaker Notes*

`PngReader` and `ResultOfPngReader` are Rust types — objects of these types cannot cross the FFI boundary without indirection of a `Box<T>`. We can't have an `out_parameter: &mut PngReader`, because CXX doesn't allow C++ to store Rust objects by value.

This example illustrates that even though CXX doesn't support arbitrary generics nor templates, we can still pass them across the FFI boundary by manually specializing / monomorphizing them into a non-generic type. In the example `ResultOfPngReader` is a non-generic type that forwards into appropriate methods of `Result<T, E>` (e.g. into `is_err`, `unwrap`, and/or `as_mut`).

Using cxx in Chromium

In Chromium, we define an independent `#[cxx::bridge] mod` for each leaf-node where we want to use Rust. You'd typically have one for each `rust_static_library`. Just add

```
cxx_bindings = [ "my_rust_file.rs" ]
   # list of files containing #[cxx::bridge], not all source files
allow_unsafe = true
```

to your existing `rust_static_library` target alongside `crate_root` and `sources`.

C++ headers will be generated at a sensible location, so you can just

```
#include "ui/base/my_rust_file.rs.h"
```

You will find some utility functions in `//base` to convert to/from Chromium C++ types to CXX Rust types — for example `SpanToRustSlice`.

▼ *Speaker Notes*

Students may ask — why do we still need `allow_unsafe = true`?

The broad answer is that no C/C++ code is "safe" by the normal Rust standards. Calling back and forth to C/C++ from Rust may do arbitrary things to memory, and compromise the safety of Rust's own data layouts. Presence of *too many* `unsafe` keywords in C/C++ interop can harm the signal-to-noise ratio of such a keyword, and is controversial, but strictly, bringing any foreign code into a Rust binary can cause unexpected behavior from Rust's perspective.

The narrow answer lies in the diagram at the top of this page — behind the scenes, CXX generates Rust `unsafe` and `extern "C"` functions just like we did manually in the previous section.

Exercise: Interoperability with C++

Part one

- In the Rust file you previously created, add a `#[cxx::bridge]` which specifies a single function, to be called from C++, called `hello_from_rust`, taking no parameters and returning no value.
- Modify your previous `hello_from_rust` function to remove `extern "C"` and `#[unsafe(no_mangle)]`. This is now just a standard Rust function.
- Modify your `gn` target to build these bindings.
- In your C++ code, remove the forward-declaration of `hello_from_rust`. Instead, include the generated header file.
- Build and run!

Part two

It's a good idea to play with CXX a little. It helps you think about how flexible Rust in Chromium actually is.

Some things to try:

- Call back into C++ from Rust. You will need:
  - An additional header file which you can `include!` from your `cxx::bridge`. You'll need to declare your C++ function in that new header file.
  - An `unsafe` block to call such a function, or alternatively specify the `unsafe` keyword in your `#[cxx::bridge]` as described here.
  - You may also need to `#include "third_party/rust/cxx/v1/crate/include/cxx.h"`
- Pass a C++ string from C++ into Rust.
- Pass a reference to a C++ object into Rust.
- Intentionally get the Rust function signatures mismatched from the `#[cxx::bridge]`, and get used to the errors you see.
- Intentionally get the C++ function signatures mismatched from the `#[cxx::bridge]`, and get used to the errors you see.
- Pass a `std::unique_ptr` of some type from C++ into Rust, so that Rust can own some C++ object.
- Create a Rust object and pass it into C++, so that C++ owns it. (Hint: you need a `Box`).
- Declare some methods on a C++ type. Call them from Rust.
- Declare some methods on a Rust type. Call them from C++.

Part three

Now you understand the strengths and limitations of CXX interop, think of a couple of use-cases for Rust in Chromium where the interface would be sufficiently simple. Sketch how you might define that interface.

Where to find help

- The `rust_static_library` gn template

▼ *Speaker Notes*

As students explore Part Two, they're bound to have lots of questions about how to achieve these things, and also how CXX works behind the scenes.

Some of the questions you may encounter:

- I'm seeing a problem initializing a variable of type X with type Y, where X and Y are both function types. This is because your C++ function doesn't quite match the declaration in your `cxx::bridge`.
- I seem to be able to freely convert C++ references into Rust references. Doesn't that risk UB? For CXX's *opaque* types, no, because they are zero-sized. For CXX trivial types yes, it's *possible* to cause UB, although CXX's design makes it quite difficult to craft such an example.

Adding Third Party Crates

Rust libraries are called “crates” and are found at crates.io. It’s *very easy* for Rust crates to depend upon one another. So they do!

Property	C++ library	Rust crate
Build system	Lots	Consistent: `Cargo.toml`
Typical library size	Large-ish	Small
Transitive dependencies	Few	Lots

For a Chromium engineer, this has pros and cons:

- All crates use a common build system so we can automate their inclusion into Chromium...
- ... but, crates typically have transitive dependencies, so you will likely have to bring in multiple libraries.

We’ll discuss:

- How to put a crate in the Chromium source code tree
- How to make `gn` build rules for it
- How to audit its source code for sufficient safety.

▼ *Speaker Notes*

All of the things in the table on this slide are generalizations, and counter-examples can be found. But in general it's important for students to understand that most Rust code depends on other Rust libraries, because it's easy to do so, and that this has both benefits and costs.

Configuring the `Cargo.toml` file to add crates

Chromium has a single set of centrally-managed direct crate dependencies. These are managed through a single `Cargo.toml`:

```
[dependencies]
bitflags = "1"
cfg-if = "1"
cxx = "1"
# lots more...
```

As with any other `Cargo.toml`, you can specify more details about the dependencies — most commonly, you'll want to specify the `features` that you wish to enable in the crate.

When adding a crate to Chromium, you'll often need to provide some extra information in an additional file, `gnrt_config.toml`, which we'll meet next.

Configuring `gnrt_config.toml`

Alongside `Cargo.toml` is `gnrt_config.toml`. This contains Chromium-specific extensions to crate handling.

If you add a new crate, you should specify at least the `group`. This is one of:

```
#   'safe': The library satisfies the rule-of-2 and can be used in any process.
#   'sandbox': The library does not satisfy the rule-of-2 and must be used in
#              a sandboxed process such as the renderer or a utility process.
#   'test': The library is only used in tests.
```

For instance,

```
[crate.my-new-crate]
group = 'test' # only used in test code
```

Depending on the crate source code layout, you may also need to use this file to specify where its `LICENSE` file(s) can be found.

Later, we'll see some other things you will need to configure in this file to resolve problems.

Downloading Crates

A tool called `gnrt` knows how to download crates and how to generate `BUILD.gn` rules.

To start, download the crate you want like this:

```
cd chromium/src
vpython3 tools/crates/run_gnrt.py -- vendor
```

> Although the `gnrt` tool is part of the Chromium source code, by running this command you will be downloading and running its dependencies from `crates.io`. See the earlier section discussing this security decision.

This `vendor` command may download:

- Your crate
- Direct and transitive dependencies
- New versions of other crates, as required by `cargo` to resolve the complete set of crates required by Chromium.

Chromium maintains patches for some crates, kept in `//third_party/rust/chromium_crates_io/patches`. These will be reapplied automatically, but if patching fails you may need to take manual action.

Generating gn Build Rules

Once you've downloaded the crate, generate the `BUILD.gn` files like this:

```
vpython3 tools/crates/run_gnrt.py -- gen
```

Now run `git status`. You should find:

- At least one new crate source code in `third_party/rust/chromium_crates_io/vendor`
- At least one new `BUILD.gn` in `third_party/rust/<crate name>/v<major semver version>`
- An appropriate `README.chromium`

The "major semver version" is a Rust "semver" version number.

Take a close look, especially at the things generated in `third_party/rust`.

▼ *Speaker Notes*

Talk a little about semver — and specifically the way that in Chromium it's to allow multiple incompatible versions of a crate, which is discouraged but sometimes necessary in the Cargo ecosystem.

Resolving Problems

If your build fails, it may be because of a `build.rs`: programs which do arbitrary things at build time. This is fundamentally at odds with the design of `gn` and `ninja` which aim for static, deterministic, build rules to maximize parallelism and repeatability of builds.

Some `build.rs` actions are automatically supported; others require action:

build script effect	Supported by our gn templates	Work required by you
Checking rustc version to configure features on and off	Yes	None
Checking platform or CPU to configure features on and off	Yes	None
Generating code	Yes	Yes - specify in `gnrt_config.toml`
Building C/C++	No	Patch around it
Arbitrary other actions	No	Patch around it

Fortunately, most crates don't contain a build script, and fortunately, most build scripts only do the top two actions.

Build Scripts Which Generate Code

If `ninja` complains about missing files, check the `build.rs` to see if it writes source code files.

If so, modify `gnrt_config.toml` to add `build-script-outputs` to the crate. If this is a transitive dependency, that is, one on which Chromium code should not directly depend, also add `allow-first-party-usage=false`. There are several examples already in that file:

```
[crate.unicode-linebreak]
allow-first-party-usage = false
build-script-outputs = ["tables.rs"]
```

Now rerun `gnrt.py -- gen` to regenerate `BUILD.gn` files to inform ninja that this particular output file is input to subsequent build steps.

Build Scripts Which Build C++ or Take Arbitrary Actions

Some crates use the `cc` crate to build and link C/C++ libraries. Other crates parse C/C++ using `bindgen` within their build scripts. These actions can't be supported in a Chromium context — our gn, ninja and LLVM build system is very specific in expressing relationships between build actions.

So, your options are:

- Avoid these crates
- Apply a patch to the crate.

Patches should be kept in `third_party/rust/chromium_crates_io/patches/<crate>` - see for example the patches against the `cxx` crate - and will be applied automatically by `gnrt` each time it upgrades the crate.

Depending on a Crate

Once you've added a third-party crate and generated build rules, depending on a crate is simple. Find your `rust_static_library` target, and add a `dep` on the `:lib` target within your crate.

Specifically,

```
//third_party/rust    crate name    /v    major semver version    :lib
```

For instance,

```
rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
  deps = [ "//third_party/rust/example_rust_crate/v1:lib" ]
}
```

Auditing Third Party Crates

Adding new libraries is subject to Chromium's standard policies, but of course also subject to security review. As you may be bringing in not just a single crate but also transitive dependencies, there may be a lot of code to review. On the other hand, safe Rust code can have limited negative side effects. How should you review it?

Over time Chromium aims to move to a process based around cargo vet.

Meanwhile, for each new crate addition, we are checking for the following:

- Understand why each crate is used. What's the relationship between crates? If the build system for each crate contains a `build.rs` or procedural macros, work out what they're for. Are they compatible with the way Chromium is normally built?
- Check each crate seems to be reasonably well maintained
- Use `cd third-party/rust/chromium_crates_io; cargo audit` to check for known vulnerabilities (first you'll need to `cargo install cargo-audit`, which ironically involves downloading lots of dependencies from the internet2)
- Ensure any `unsafe` code is good enough for the Rule of Two
- Check for any use of `fs` or `net` APIs
- Read all the code at a sufficient level to look for anything out of place that might have been maliciously inserted. (You can't realistically aim for 100% perfection here: there's often just too much code.)

These are just guidelines — work with reviewers from `security@chromium.org` to work out the right way to become confident of the crate.

Checking Crates into Chromium Source Code

`git status` should reveal:

- Crate code in `//third_party/rust/chromium_crates_io`
- Metadata (`BUILD.gn` and `README.chromium`) in `//third_party/rust/<crate>/<version>`

Please also add an `OWNERS` file in the latter location.

You should land all this, along with your `Cargo.toml` and `gnrt_config.toml` changes, into the Chromium repo.

Important: you need to use `git add -f` because otherwise `.gitignore` files may result in some files being skipped.

As you do so, you might find presubmit checks fail because of non-inclusive language. This is because Rust crate data tends to include names of git branches, and many projects still use non-inclusive terminology there. So you may need to run:

```
infra/update_inclusive_language_presubmit_exempt_dirs.sh >
infra/inclusive_language_presubmit_exempt_dirs.txt
git add -p infra/inclusive_language_presubmit_exempt_dirs.txt # add whatever changes
are yours
```

Keeping Crates Up to Date

As the OWNER of any third party Chromium dependency, you are expected to keep it up to date with any security fixes. It is hoped that we will soon automate this for Rust crates, but for now, it's still your responsibility just as it is for any other third party dependency.

Exercise

Add uwuify to Chromium, turning off the crate's default features. Assume that the crate will be used in shipping Chromium, but won't be used to handle untrustworthy input.

(In the next exercise we'll use uwuify from Chromium, but feel free to skip ahead and do that now if you like. Or, you could create a new `rust_executable` target which uses `uwuify`).

▼ *Speaker Notes*

Students will need to download lots of transitive dependencies.

The total crates needed are:

- `instant`,
- `lock_api`,
- `parking_lot`,
- `parking_lot_core`,
- `redox_syscall`,
- `scopeguard`,
- `smallvec`, and
- `uwuify`.

If students are downloading even more than that, they probably forgot to turn off the default features.

Thanks to Daniel Liu for this crate!

Bringing It Together — Exercise

In this exercise, you're going to add a whole new Chromium feature, bringing together everything you already learned.

The Brief from Product Management

A community of pixies has been discovered living in a remote rainforest. It's important that we get Chromium for Pixies delivered to them as soon as possible.

The requirement is to translate all Chromium's UI strings into Pixie language.

There's not time to wait for proper translations, but fortunately pixie language is very close to English, and it turns out there's a Rust crate which does the translation.

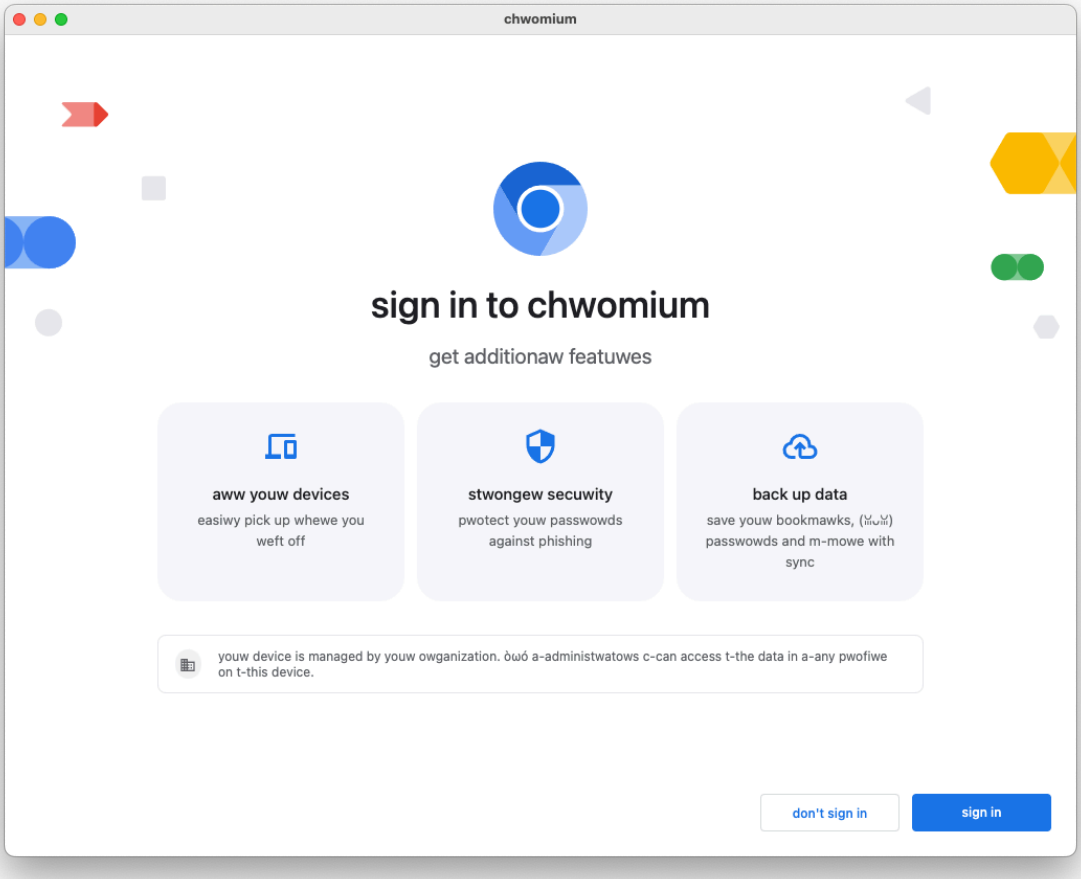
In fact, you already imported that crate in the previous exercise.

(Obviously, real translations of Chrome require incredible care and diligence. Don't ship this!)

Steps

Modify `ResourceBundle::MaybeMangleLocalizedString` so that it uwuifies all strings before display. In this special build of Chromium, it should always do this irrespective of the setting of `mangle_localized_strings_`.

If you've done everything right across all these exercises, congratulations, you should have created Chrome for pixies!



▼ *Speaker Notes*

Students will likely need some hints here. Hints include:

- UTF16 vs UTF8. Students should be aware that Rust strings are always UTF8, and will probably decide that it's better to do the conversion on the C++ side using `base::UTF16ToUTF8` and back again.
- If students decide to do the conversion on the Rust side, they'll need to consider `String::from_utf16`, consider error handling, and consider which CXX supported types can transfer a lot of u16s.
- Students may design the C++/Rust boundary in several different ways, e.g. taking and returning strings by value, or taking a mutable reference to a string. If a mutable reference is used, CXX will likely tell the student that they need to use `Pin`. You may need to explain what `Pin` does, and then explain why CXX needs it for mutable references to C++ data: the answer is that C++ data can't be moved around like Rust data, because it may contain self-referential pointers.
- The C++ target containing `ResourceBundle::MaybeMangleLocalizedString` will need to depend on a `rust_static_library` target. The student probably already did this.
- The `rust_static_library` target will need to depend on `//third_party/rust/uwuify/v0_2:lib`.