Solution

```rust
mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomP
    }

    // Layout according to the Linux man page for readdir(3), where ino_t a
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(target_os = "macos")]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the sectior
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" ref
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerP
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

```

```rust
use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        let path =
            CString::new(path).map_err(|err| format!("Invalid path: {err}")
        // SAFETY: path.as_ptr() cannot be NULL.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("Could not open {path:?}"))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        // SAFETY: self.dir is never NULL.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // We have reached the end of the directory.
            return None;
        }
        // SAFETY: dirent is not NULL and dirent.d_name is NUL
        // terminated.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        // SAFETY: self.dir is never NULL.
        if unsafe { ffi::closedir(self.dir) } != 0 {
            panic!("Could not close {:?}", self.path);
        }
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
```

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use std::error::Error;

    #[test]
    fn test_nonexisting_directory() {
        let iter = DirectoryIterator::new("no-such-directory");
        assert!(iter.is_err());
    }

    #[test]
    fn test_empty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", ".."]);
        Ok(())
    }

    #[test]
    fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        std::fs::write(tmp.path().join("foo.txt"), "The Foo Diaries\n")?;
        std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
        std::fs::write(tmp.path().join("crab.rs"), "//! Crab\n")?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
        Ok(())
    }
}
```

Welcome to Rust in Android

Rust is supported for system software on Android. This means that you can write new services, libraries, drivers or even firmware in Rust (or improve existing code as needed).

▼ *Speaker Notes*

The speaker may mention any of the following given the increased use of Rust in Android:

- Service example: DNS over HTTP.
- Libraries: Rutabaga Virtual Graphics Interface.
- Kernel Drivers: Binder.
- Firmware: pKVM firmware.

Setup

We will be using a Cuttlefish Android Virtual Device to test our code. Make sure you have access to one or create a new one with:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-userdebug
acloud create
```

Please see the Android Developer Codelab for details.

The code on the following pages can be found in the `src/android/` directory of the course material. Please `git clone` the repository to follow along.

▼ *Speaker Notes*

Key points:

- Cuttlefish is a reference Android device designed to work on generic Linux desktops. MacOS support is also planned.
- The Cuttlefish system image maintains high fidelity to real devices, and is the ideal emulator to run many Rust use cases.

Build Rules

The Android build system (Soong) supports Rust through several modules:

Module Type	Description
`rust_binary`	Produces a Rust binary.
`rust_library`	Produces a Rust library, and provides both `rlib` and `dylib` variants.
`rust_ffi`	Produces a Rust C library usable by `cc` modules, and provides both static and shared variants.
`rust_proc_macro`	Produces a `proc-macro` Rust library. These are analogous to compiler plugins.
`rust_test`	Produces a Rust test binary that uses the standard Rust test harness.
`rust_fuzz`	Produces a Rust fuzz binary leveraging `libfuzzer`.
`rust_protobuf`	Generates source and produces a Rust library that provides an interface for a particular protobuf.
`rust_bindgen`	Generates source and produces a Rust library containing Rust bindings to C libraries.

We will look at `rust_binary` and `rust_library` next.

▼ *Speaker Notes*

Additional items the speaker may mention:

- Cargo is not optimized for multi-language repositories, and also downloads packages from the internet.
- For compliance and performance, Android must have crates in-tree. It must also interoperate with C/C++/Java code. Soong fills that gap.
- Soong has many similarities to Bazel, which is the open-source variant of Blaze (used in google3).
- Fun fact: Data from Star Trek is a Soong-type Android.

Rust Binaries

Let's start with a simple application. At the root of an AOSP checkout, create the following files:

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust demo.

/// Prints a greeting to standard output.
fn main() {
    println!("Hello from Rust!");
}
```

You can now build, push, and run the binary:

```
m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust" /data/local/tmp
adb shell /data/local/tmp/hello_rust
```

```
Hello from Rust!
```

▼ *Speaker Notes*

- Go through the build steps and demonstrate them running in your emulator.
- Notice the extensive documentation comments? The Android build rules enforce that all modules have documentation. Try removing it and see what error you get.
- Stress that the Rust build rules look like the other Soong rules. This is by design, to make using Rust as easy as C++ or Java.

Rust Libraries

You use `rust_library` to create a new Rust library for Android.

Here we declare a dependency on two libraries:

- `libgreeting`, which we define below,
- `libtextwrap`, which is a crate already vendored in `external/rust/android-crates-io/crates/`.

hello_rust/Android.bp:

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
    ],
    prefer_rlib: true, // Need this to avoid dynamic link error.
}

rust_library {
    name: "libgreetings",
    crate_name: "greetings",
    srcs: ["src/lib.rs"],
}
```

hello_rust/src/main.rs:

```
//! Rust demo.

use greetings::greeting;
use textwrap::fill;

/// Prints a greeting to standard output.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

hello_rust/src/lib.rs:

```
//! Greeting library.

/// Greet `name`.
pub fn greeting(name: &str) -> String {
```

You build, push, and run the binary like before:

```
m hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep" /data/local/tmp
adb shell /data/local/tmp/hello_rust_with_dep
```

```
Hello Bob, it is very
nice to meet you!
```

▼ *Speaker Notes*

- Go through the build steps and demonstrate them running in your emulator.
- A Rust crate named `greetings` must be built by a rule called `libgreetings`. Note how the Rust code uses the crate name, as is normal in Rust.
- Again, the build rules enforce that we add documentation comments to all public items.

AIDL

Rust supports the Android Interface Definition Language (AIDL):

- Rust code can call existing AIDL servers.
- You can create new AIDL servers in Rust.

▼ *Speaker Notes*

- AIDL enables Android apps to interact with each other.
- Since Rust is a first-class citizen in this ecosystem, other processes on the device can call Rust services.

Birthday Service Tutorial

To illustrate using Rust with Binder, we will create a Binder interface. Then, we'll implement the service and write a client that talks to it.

AIDL Interfaces

You declare the API of your service using an AIDL interface:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

birthday_service/aidl/Android.bp:

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
        rust: { // Rust is not enabled by default
            enabled: true,
        },
    },
}
```

▼ *Speaker Notes*

- Note that the directory structure under the `aidl/` directory needs to match the package name used in the AIDL file, i.e. the package is `com.example.birthdayservice` and the file is at `aidl/com/example/IBirthdayService.aidl`.

Generated Service API

Binder generates a trait for each interface definition.

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

out/soong/.intermediates/.../com_example_birthdayservice.rs:

```
trait IBirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) ->
binder::Result<String>;
}
```

Your service will need to implement this trait, and your client will use this trait to talk to the service.

▼ *Speaker Notes*

- Point out how the generated function signature, specifically the argument and return types, correspond to the interface definition.
  - `String` for an argument results in a different Rust type than `String` as a return type.

Service Implementation

We can now implement the AIDL service:

birthday_service/src/lib.rs:

```
//! Implementation of the `IBirthdayService` AIDL interface.
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::IBirthdayService;
use com_example_birthdayservice::binder;

/// The `IBirthdayService` implementation.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) ->
binder::Result<String> {
        Ok(format!("Happy Birthday {name}, congratulations with the {years}
years!"))
    }
}
```

birthday_service/Android.bp:

```
rust_library {
    name: "libbirthdayservice",
    crate_name: "birthdayservice",
    srcs: ["src/lib.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
    ],
}
```

▼ *Speaker Notes*

- Point out the path to the generated `IBirthdayService` trait, and explain why each of the segments is necessary.
- Note that `wishHappyBirthday` and other AIDL IPC methods take `&self` (instead of `&mut self`).
    - This is necessary because Binder responds to incoming requests on a thread pool, allowing for multiple requests to be processed in parallel. This requires that the service methods only get a shared reference to `self`.

  - The correct approach for managing service state depends heavily on the details of your service.
- TODO: What does the `binder::Interface` trait do? Are there methods to override? Where is the source?

AIDL Server

Finally, we can create a server which exposes the service:

birthday_service/src/server.rs:

```rust
//! Birthday service.
use birthdayservice::BirthdayService;
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::BnBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Entry point for birthday service.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER,
birthday_service_binder.as_binder())
        .expect("Failed to register service");
    binder::ProcessState::join_thread_pool();
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbirthdayservice",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

▼ *Speaker Notes*

The process for taking a user-defined service implementation (in this case, the `BirthdayService` type, which implements the `IBirthdayService`) and starting it as a Binder service has multiple steps. This may appear more complicated than students are used to if they've used Binder from C++ or another language. Explain to students why each

1. Create an instance of your service type (`BirthdayService`).
2. Wrap the service object in the corresponding `Bn*` type (`BnBirthdayService` in this case). This type is generated by Binder and provides common Binder functionality, similar to the `BnBinder` base class in C++. Since Rust doesn't have inheritance, we use composition, putting our `BirthdayService` within the generated `BnBinderService` .
3. Call `add_service` , giving it a service identifier and your service object (the `BnBirthdayService` object in the example).
4. Call `join_thread_pool` to add the current thread to Binder's thread pool and start listening for connections.

Deploy

We can now build, push, and start the service:

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server" /data/local/tmp
adb root
adb shell /data/local/tmp/birthday_server
```

In another terminal, check that the service runs:

```
adb shell service check birthdayservice
```

```
Service birthdayservice: found
```

You can also call the service with `service call`:

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

```
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....        ')
```

AIDL Client

Finally, we can create a Rust client for our new service.

birthday_service/src/client.rs:

```
use
com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayServ
ice::IBirthdayService;
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Call the birthday service.
fn main() -> Result<(), Box<dyn Error>> {
    let name = std::env::args().nth(1).unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = binder::get_interface::<dyn IBirthdayService>
(SERVICE_IDENTIFIER)
        .map_err(|_| "Failed to connect to BirthdayService")?;

    // Call the service.
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
}
```

birthday_service/Android.bp:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

Notice that the client does not depend on `libbirthdayservice`.

Build, push, and run the client on your device:

```
m birthday_client
```

```
Happy Birthday Charlie, congratulations with the 60 years!
```

▼ *Speaker Notes*

- `Strong<dyn IBirthdayService>` is the trait object representing the service that the client has connected to.
  - `Strong` is a custom smart pointer type for Binder. It handles both an in-process ref count for the service trait object, and the global Binder ref count that tracks how many processes have a reference to the object.
  - Note that the trait object that the client uses to talk to the service uses the exact same trait that the server implements. For a given Binder interface, there is a single Rust trait generated that both client and server use.
- Use the same service identifier used when registering the service. This should ideally be defined in a common crate that both the client and server can depend on.

Changing API

Let's extend the API: we'll let clients specify a list of lines for the birthday card:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

This results in an updated trait definition for `IBirthdayService`:

```
trait IBirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String>;
}
```

▼ *Speaker Notes*

- Note how the `String[]` in the AIDL definition is translated as a `&[String]` in Rust, i.e. that idiomatic Rust types are used in the generated bindings wherever possible:
  - `in` array arguments are translated to slices.
  - `out` and `inout` args are translated to `&mut Vec<T>`.
  - Return values are translated to returning a `Vec<T>`.

Updating Client and Service

Update the client and server code to account for the new API.

birthday_service/src/lib.rs:

```rust
impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String> {
        let mut msg = format!(
            "Happy Birthday {name}, congratulations with the {years} years!",
        );

        for line in text {
            msg.push('\n');
            msg.push_str(line);
        }

        Ok(msg)
    }
}
```

birthday_service/src/client.rs:

```rust
let msg = service.wishHappyBirthday(
    &name,
    years,
    &[
        String::from("Habby birfday to yuuuuu"),
        String::from("And also: many more"),
    ],
)?;
```

▼ *Speaker Notes*

- TODO: Move code snippets into project files where they'll actually be built?

Working With AIDL Types

AIDL types translate into the appropriate idiomatic Rust type:

- Primitive types map (mostly) to idiomatic Rust types.
- Collection types like slices, `Vec` s and string types are supported.
- References to AIDL objects and file handles can be sent between clients and services.
- File handles and parcelables are fully supported.

Primitive Types

Primitive types map (mostly) idiomatically:

AIDL Type	Rust Type	Note
`boolean`	`bool`	
`byte`	`i8`	Note that bytes are signed.
`char`	`u16`	Note the usage of `u16`, NOT `u32`.
`int`	`i32`	
`long`	`i64`	
`float`	`f32`	
`double`	`f64`	
`String`	`String`	

Array Types

The array types (`T[]` , `byte[]` , and `List<T>`) are translated to the appropriate Rust array type depending on how they are used in the function signature:

Position	Rust Type
`in` argument	`&[T]`
`out` / `inout` argument	`&mut Vec<T>`
Return	`Vec<T>`

▼ *Speaker Notes*

- In Android 13 or higher, fixed-size arrays are supported, i.e. `T[N]` becomes `[T; N]` . Fixed-size arrays can have multiple dimensions (e.g. `int[3][4]`). In the Java backend, fixed-size arrays are represented as array types.
- Arrays in parcelable fields always get translated to `Vec<T>` .

Sending Objects

AIDL objects can be sent either as a concrete AIDL type or as the type-erased `IBinder` interface:

birthday_service/aidl/com/example/birthdayservice/IBirthdayInfoProvider.aidl:

```
package com.example.birthdayservice;

interface IBirthdayInfoProvider {
    String name();
    int years();
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.IBirthdayInfoProvider;

interface IBirthdayService {
    /** The same thing, but using a binder object. */
    String wishWithProvider(IBirthdayInfoProvider provider);

    /** The same thing, but using `IBinder`. */
    String wishWithErasedProvider(IBinder provider);
}
```

birthday_service/src/client.rs:

```rust
/// Rust struct implementing the `IBirthdayInfoProvider` interface.
struct InfoProvider {
    name: String,
    age: u8,
}

impl binder::Interface for InfoProvider {}

impl IBirthdayInfoProvider for InfoProvider {
    fn name(&self) -> binder::Result<String> {
        Ok(self.name.clone())
    }

    fn years(&self) -> binder::Result<i32> {
        Ok(self.age as i32)
    }
}

fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    // Create a binder object for the `IBirthdayInfoProvider` interface.
    let provider = BnBirthdayInfoProvider::new_binder(
        InfoProvider { name: name.clone(), age: years as u8 },
        BinderFeatures::default(),
    );

    // Send the binder object to the service.
    service.wishWithProvider(&provider)?;

    // Perform the same operation but passing the provider as an `SpIBinder`.
    service.wishWithErasedProvider(&provider.as_binder())?;
}
```

▼ *Speaker Notes*

- Note the usage of `BnBirthdayInfoProvider`. This serves the same purpose as `BnBirthdayService` that we saw previously.

Parcelables

Binder for Rust supports sending parcelables directly:

birthday_service/aidl/com/example/birthdayservice/BirthdayInfo.aidl:

```
package com.example.birthdayservice;

parcelable BirthdayInfo {
    String name;
    int years;
}
```

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
import com.example.birthdayservice.BirthdayInfo;

interface IBirthdayService {
    /** The same thing, but with a parcelable. */
    String wishWithInfo(in BirthdayInfo info);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    let info = BirthdayInfo { name: "Alice".into(), years: 123 };
    service.wishWithInfo(&info)?;
}
```

Sending Files

Files can be sent between Binder clients/servers using the `ParcelFileDescriptor` type:

birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl:

```
interface IBirthdayService {
    /** The same thing, but loads info from a file. */
    String wishFromFile(in ParcelFileDescriptor infoFile);
}
```

birthday_service/src/client.rs:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    // Open a file and put the birthday info in it.
    let mut file = File::create("/data/local/tmp/birthday.info").unwrap();
    writeln!(file, "{name}")?;
    writeln!(file, "{years}")?;

    // Create a `ParcelFileDescriptor` from the file and send it.
    let file = ParcelFileDescriptor::new(file);
    service.wishFromFile(&file)?;
}
```

birthday_service/src/lib.rs:

```
impl IBirthdayService for BirthdayService {
    fn wishFromFile(
        &self,
        info_file: &ParcelFileDescriptor,
    ) -> binder::Result<String> {
        // Convert the file descriptor to a `File`. `ParcelFileDescriptor`
wraps
        // an `OwnedFd`, which can be cloned and then used to create a `File`
        // object.
        let mut info_file = info_file
            .as_ref()
            .try_clone()
            .map(File::from)
            .expect("Invalid file handle");

        let mut contents = String::new();
        info_file.read_to_string(&mut contents).unwrap();

        let mut lines = contents.lines();
        let name = lines.next().unwrap();
        let years: i32 = lines.next().unwrap().parse().unwrap();

        Ok(format!("Happy Birthday {name}, congratulations with the {years}
years!"))
    }
}
```

▼ *Speaker Notes*

- `ParcelFileDescriptor` wraps an `OwnedFd`, and so can be created from a `File` (or any other type that wraps an `OwnedFd`), and can be used to create a new `File` handle on the other side.
- Other types of file descriptors can be wrapped and sent, e.g. TCP, UDP, and UNIX sockets.

Testing in Android

Building on Testing, we will now look at how unit tests work in AOSP. Use the `rust_test` module for your unit tests:

testing/Android.bp:

```
rust_library {
    name: "libleftpad",
    crate_name: "leftpad",
    srcs: ["src/lib.rs"],
}

rust_test {
    name: "libleftpad_test",
    crate_name: "leftpad_test",
    srcs: ["src/lib.rs"],
    host_supported: true,
    test_suites: ["general-tests"],
}

rust_test {
    name: "libgoogletest_example",
    crate_name: "googletest_example",
    srcs: ["googletest.rs"],
    rustlibs: ["libgoogletest_rust"],
    host_supported: true,
}

rust_test {
    name: "libmockall_example",
    crate_name: "mockall_example",
    srcs: ["mockall.rs"],
    rustlibs: ["libmockall"],
    host_supported: true,
}
```

testing/src/lib.rs:

```rust
//! Left-padding library.

/// Left-pad `s` to `width`.
pub fn leftpad(s: &str, width: usize) -> String {
    format!("{s:>width$}")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn short_string() {
        assert_eq!(leftpad("foo", 5), "  foo");
    }

    #[test]
    fn long_string() {
        assert_eq!(leftpad("foobar", 6), "foobar");
    }
}
```

You can now run the test with

```
atest --host libleftpad_test
```

The output looks like this:

```
INFO: Elapsed time: 2.666s, Critical Path: 2.40s
INFO: 3 processes: 2 internal, 1 linux-sandbox.
INFO: Build completed successfully, 3 total actions
//comprehensive-rust-android/testing:libleftpad_test_host            PASSED in
2.3s
    PASSED  libleftpad_test.tests::long_string (0.0s)
    PASSED  libleftpad_test.tests::short_string (0.0s)
Test cases: finished with 2 passing and 0 failing out of 2 test cases
```

Notice how you only mention the root of the library crate. Tests are found recursively in nested modules.

GoogleTest

The GoogleTest crate allows for flexible test assertions using *matchers*:

```
use googletest::prelude::*;

#[googletest::test]
fn test_elements_are() {
    let value = vec!["foo", "bar", "baz"];
    expect_that!(value, elements_are!(eq(&"foo"), lt(&"xyz"),
starts_with("b")));
}
```

If we change the last element to `"!"`, the test fails with a structured error message pin-pointing the error:

```
---- test_elements_are stdout ----
Value of: value
Expected: has elements:
  0. is equal to "foo"
  1. is less than "xyz"
  2. starts with prefix "!"
Actual: ["foo", "bar", "baz"],
  where element #2 is "baz", which does not start with "!"
  at src/testing/googletest.rs:6:5
Error: See failure output above
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- GoogleTest is not part of the Rust Playground, so you need to run this example in a local environment. Use `cargo add googletest` to quickly add it to an existing Cargo project.
- The `use googletest::prelude::*;` line imports a number of commonly used macros and types.
- This just scratches the surface, there are many builtin matchers. Consider going through the first chapter of "Advanced testing for Rust applications", a self-guided Rust course: it provides a guided introduction to the library, with exercises to help you get comfortable with `googletest` macros, its matchers and its overall philosophy.
- A particularly nice feature is that mismatches in multi-line strings are shown as a diff:

```rust
#[test]
fn test_multiline_string_diff() {
    let haiku = "Memory safety found,\n\
                 Rust's strong typing guides the way,\n\
                 Secure code you'll write.";
    assert_that!(
        haiku,
        eq("Memory safety found,\n\
            Rust's silly humor guides the way,\n\
            Secure code you'll write.")
    );
}
```

shows a color-coded diff (colors not shown here):

```
    Value of: haiku
Expected: is equal to "Memory safety found,\nRust's silly humor guides the
way,\nSecure code you'll write."
Actual: "Memory safety found,\nRust's strong typing guides the way,\nSecure
code you'll write.",
  which isn't equal to "Memory safety found,\nRust's silly humor guides the
way,\nSecure code you'll write."
Difference(-actual / +expected):
 Memory safety found,
-Rust's strong typing guides the way,
+Rust's silly humor guides the way,
 Secure code you'll write.
  at src/testing/googletest.rs:17:5
```

- The crate is a Rust port of GoogleTest for C++.

Mocking

For mocking, Mockall is a widely used library. You need to refactor your code to use traits, which you can then quickly mock:

```rust
use std::time::Duration;

#[mockall::automock]
pub trait Pet {
    fn is_hungry(&self, since_last_meal: Duration) -> bool;
}

#[test]
fn test_robot_dog() {
    let mut mock_dog = MockPet::new();
    mock_dog.expect_is_hungry().return_const(true);
    assert!(mock_dog.is_hungry(Duration::from_secs(10)));
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- Mockall is the recommended mocking library in Android (AOSP). There are other mocking libraries available on crates.io, in particular in the area of mocking HTTP services. The other mocking libraries work in a similar fashion as Mockall, meaning that they make it easy to get a mock implementation of a given trait.

- Note that mocking is somewhat *controversial*: mocks allow you to completely isolate a test from its dependencies. The immediate result is faster and more stable test execution. On the other hand, the mocks can be configured wrongly and return output different from what the real dependencies would do.

  If at all possible, it is recommended that you use the real dependencies. As an example, many databases allow you to configure an in-memory backend. This means that you get the correct behavior in your tests, plus they are fast and will automatically clean up after themselves.

  Similarly, many web frameworks allow you to start an in-process server which binds to a random port on `localhost`. Always prefer this over mocking away the framework since it helps you test your code in the real environment.

- Mockall is not part of the Rust Playground, so you need to run this example in a local environment. Use `cargo add mockall` to quickly add Mockall to an existing Cargo project.

hungry 3 hours after the last time it was fed:

```
#[test]
fn test_robot_cat() {
    let mut mock_cat = MockPet::new();
    mock_cat
        .expect_is_hungry()
        .with(mockall::predicate::gt(Duration::from_secs(3 * 3600)))
        .return_const(true);
    mock_cat.expect_is_hungry().return_const(false);
    assert!(mock_cat.is_hungry(Duration::from_secs(5 * 3600)));
    assert!(!mock_cat.is_hungry(Duration::from_secs(5)));
}
```

- You can use `.times(n)` to limit the number of times a mock method can be called to `n` — the mock will automatically panic when dropped if this isn't satisfied.

Logging

You should use the `log` crate to automatically log to `logcat` (on-device) or `stdout` (on-host):

hello_rust_logs/Android.bp:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    host_supported: true,
}
```

hello_rust_logs/src/main.rs:

```
//! Rust logging demo.

use log::{debug, error, info};

/// Logs a greeting.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_max_level(log::LevelFilter::Trace),
    );
    debug!("Starting program.");
    info!("Things are going fine.");
    error!("Something went wrong!");
}
```

Build, push, and run the binary on your device:

```
m hello_rust_logs
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs" /data/local/tmp
adb shell /data/local/tmp/hello_rust_logs
```

The logs show up in `adb logcat`:

```
adb logcat -s rust
```

▼ *Speaker Notes*

- The logger implementation in `liblogger` is only needed in the final binary, if you're logging from a library you only need the `log` facade crate.

Interoperability

Rust has excellent support for interoperability with other languages. This means that you can:

- Call Rust functions from other languages.
- Call functions written in other languages from Rust.

When you call functions in a foreign language, you're using a *foreign function interface*, also known as FFI.

▼ *Speaker Notes*

- This is a key ability of Rust: compiled code becomes indistinguishable from compiled C or C++ code.

- Technically, we say that Rust can be compiled to the same ABI (application binary interface) as C code.

Interoperability with C

Rust has full support for linking object files with a C calling convention. Similarly, you can export Rust functions and call them from C.

You can do it by hand if you want:

```rust
unsafe extern "C" {
    safe fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = abs(x);
    println!("{x}, {abs_x}");
}
```

We already saw this in the Safe FFI Wrapper exercise.

This assumes full knowledge of the target platform. Not recommended for production.

We will look at better options next.

▼ *Speaker Notes*

- The `"C"` part of the `extern` block tells Rust that `abs` can be called using the C ABI (application binary interface).
- The `safe fn abs` part tells Rust that `abs` is a safe function. By default, extern functions are unsafe, but since `abs(x)` can't trigger undefined behavior with any `x`, we can declare it safe.

A Simple C Library

Let's first create a small C library:

interoperability/bindgen/libbirthday.h:

```c
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

interoperability/bindgen/libbirthday.c:

```c
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| Happy Birthday %s!\n", card->name);
  printf("| Congratulations with the %i years!\n", card->years);
  printf("+--------------\n");
}
```

Add this to your `Android.bp` file:

interoperability/bindgen/Android.bp:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

Using Bindgen

The bindgen tool can auto-generate bindings from a C header file.

Create a wrapper header file for the library (not strictly needed in this example):

interoperability/bindgen/libbirthday_wrapper.h:

```
#include "libbirthday.h"
```

interoperability/bindgen/Android.bp:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "bindings",
    static_libs: ["libbirthday"],
}
```

Finally, we can use the bindings in our Rust program:

interoperability/bindgen/Android.bp:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
    static_libs: ["libbirthday"],
}
```

interoperability/bindgen/main.rs:

```
//! Bindgen demo.

use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("Peter").unwrap();
    let card = card { name: name.as_ptr(), years: 42 };
    // SAFETY: The pointer we pass is valid because it came from a Rust
    // reference, and the `name` it contains refers to `name` above which also
    // remains valid. `print_card` doesn't store either pointer to use later
    // after it returns.
    unsafe {
        print_card(&card);
    }
```

▼ *Speaker Notes*

- The Android build rules will automatically call `bindgen` for you behind the scenes.
- Notice that the Rust code in `main` is still hard to write. It is good practice to encapsulate the output of `bindgen` in a Rust library which exposes a safe interface to caller.

Running Our Binary

Build, push, and run the binary on your device:

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card" /data/local/tmp
adb shell /data/local/tmp/print_birthday_card
```

Finally, we can run auto-generated tests to ensure the bindings work:

interoperability/bindgen/Android.bp:

```
rust_test {
    name: "libbirthday_bindgen_test",
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // Generated file, skip linting
    lints: "none",
}
```

```
atest libbirthday_bindgen_test
```

A Simple Rust Library

Exporting Rust functions and types to C is easy. Here's a simple Rust library:

interoperability/rust/libanalyze/analyze.rs

```rust
//! Rust FFI demo.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// Analyze the numbers.
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x ({x}) is smallest!");
    } else {
        println!("y ({y}) is probably larger than x ({x})");
    }
}
```

interoperability/rust/libanalyze/Android.bp

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

▼ *Speaker Notes*

`#[unsafe(no_mangle)]` disables Rust's usual name mangling, so the exported symbol will just be the name of the function. You can also use `#[unsafe(export_name = "some_name")]` to specify whatever name you want.

Calling Rust

We can now call this from a C binary:

interoperability/rust/libanalyze/analyze.h

```c
#ifndef ANALYZE_H
#define ANALYZE_H

void analyze_numbers(int x, int y);

#endif
```

interoperability/rust/analyze/main.c

```c
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

interoperability/rust/analyze/Android.bp

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

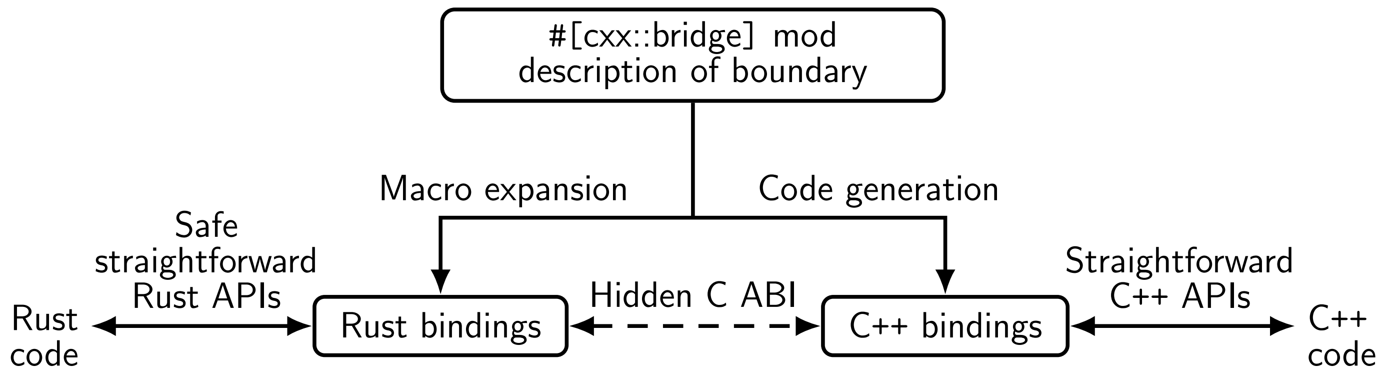
Build, push, and run the binary on your device:

```
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers" /data/local/tmp
adb shell /data/local/tmp/analyze_numbers
```

With C++

The CXX crate enables safe interoperability between Rust and C++.

The overall approach looks like this:



The Bridge Module

CXX relies on a description of the function signatures that will be exposed from each language to the other. You provide this description using extern blocks in a Rust module annotated with the `#[cxx::bridge]` attribute macro.

```rust
#[allow(unsafe_op_in_unsafe_fn)]
#[cxx::bridge(namespace = "org::blobstore")]
mod ffi {
    // Shared structs with fields visible to both languages.
    struct BlobMetadata {
        size: usize,
        tags: Vec<String>,
    }

    // Rust types and signatures exposed to C++.
    extern "Rust" {
        type MultiBuf;

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

    // 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;
    }
}
```

▼ *Speaker Notes*

- The bridge is generally declared in an `ffi` module within your crate.
- From the declarations made in the bridge module, CXX will generate matching Rust and C++ type/function definitions in order to expose those items to both languages.
- To view the generated Rust code, use cargo-expand to view the expanded proc macro. For most of the examples you would use `cargo expand ::ffi` to expand just the `ffi` module (though this doesn't apply for Android projects).
- To view the generated C++ code, look in `target/cxxbridge`.

Rust Bridge Declarations

```rust
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MyType; // Opaque type
        fn foo(&self); // Method on `MyType`
        fn bar() -> Box<MyType>; // Free function
    }
}

struct MyType(i32);

impl MyType {
    fn foo(&self) {
        println!("{}", self.0);
    }
}

fn bar() -> Box<MyType> {
    Box::new(MyType(123))
}
```

▼ *Speaker Notes*

- Items declared in the `extern "Rust"` reference items that are in scope in the parent module.
- The CXX code generator uses your `extern "Rust"` section(s) to produce a C++ header file containing the corresponding C++ declarations. The generated header has the same path as the Rust source file containing the bridge, except with a .rs.h file extension.

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)
        }
    }
}

// ...
```

- 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 UTF-8 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 that 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`

- `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.

- Broad, generic examples of projects where `cargo` may be the right choice:

  - 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 type system) 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:

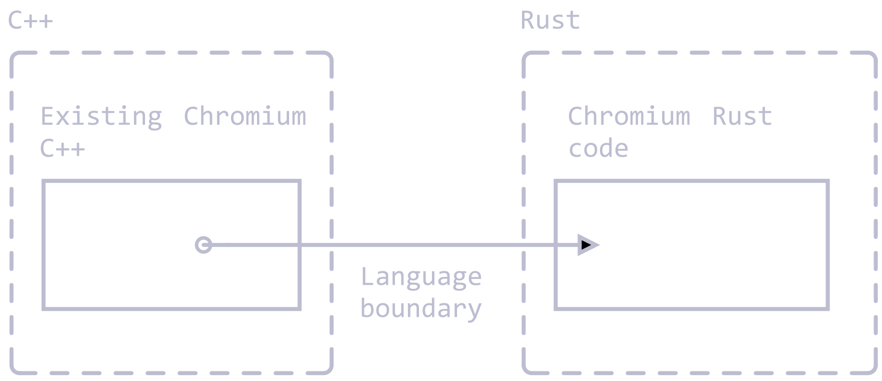
- `rustc` (the Rust compiler) which in turn depends on the LLVM libraries, the Clang compiler, the `rustc` sources (fetched from GitHub, reviewed by Rust compiler team), binary Rust compiler downloaded for bootstrapping

- `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)

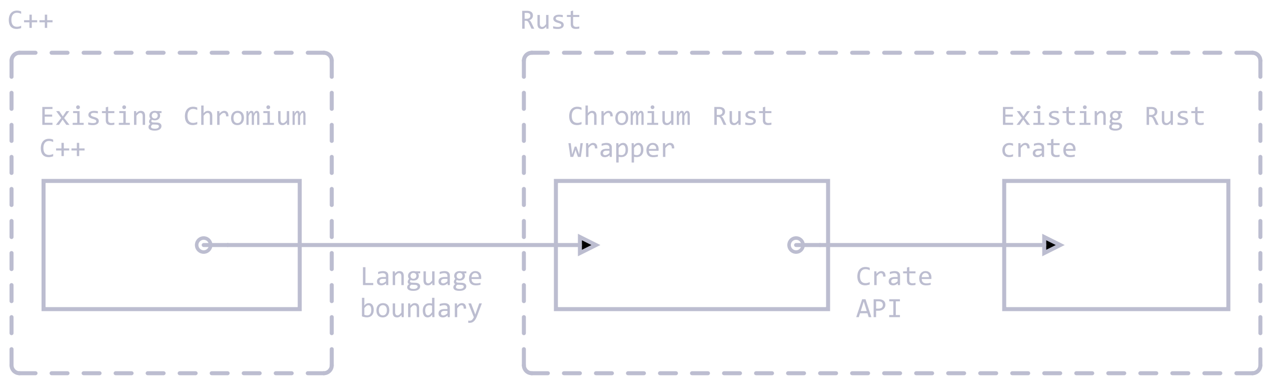
Chromium Rust policy

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" ]
}
```

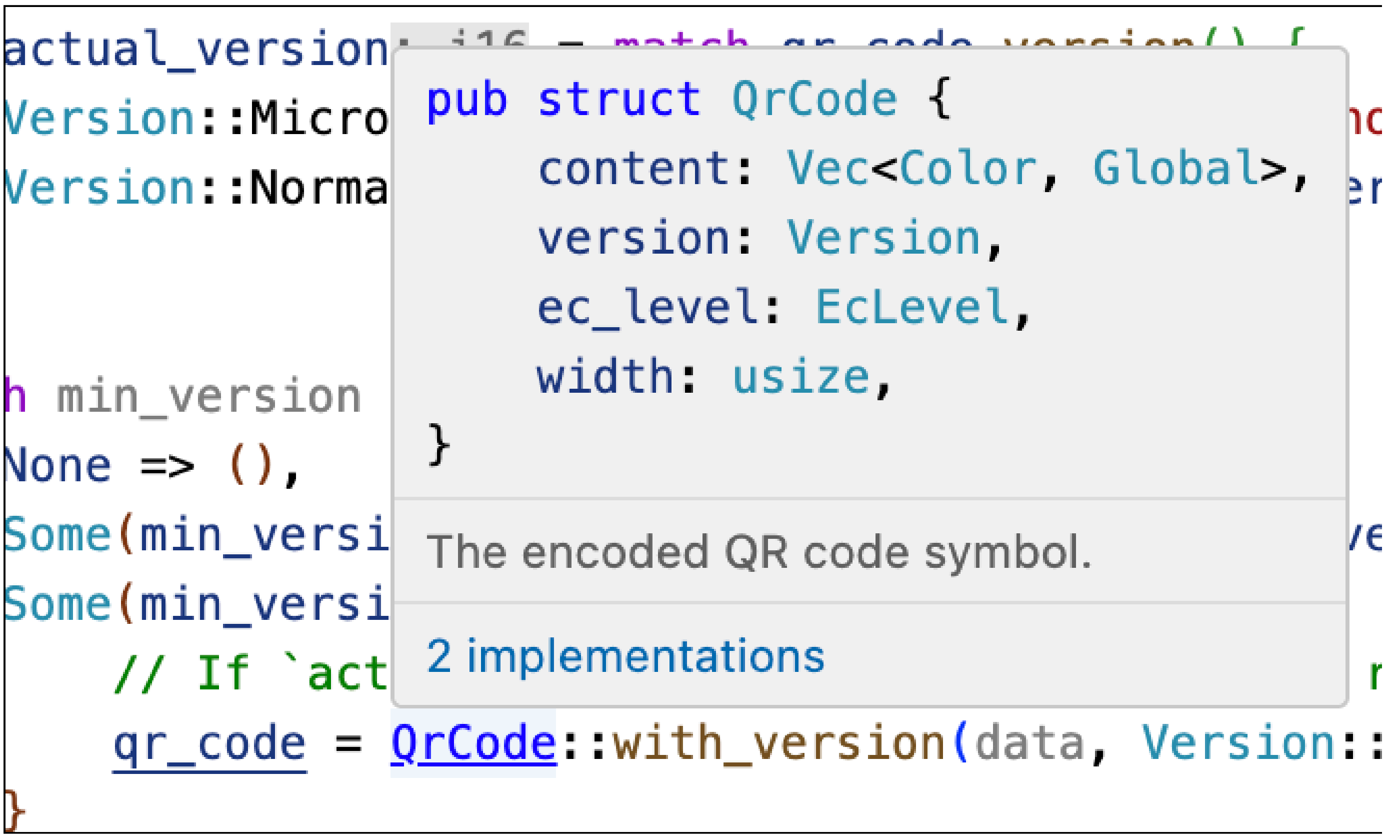
▼ *Speaker Notes*

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`



```
actual_version: i16 = match qr_code.version() {
Version::Micro                                    nc
Version::Norma                                    er


h min_version
None => (),
Some(min_versi                                    ve
Some(min_versi
    // If `act                                    r
    qr_code = QrCode::with_version(data, Version::
}
```

```
pub struct QrCode {
    content: Vec<Color, Global>,
    version: Version,
    ec_level: EcLevel,
    width: usize,
}
```

The encoded QR code symbol.

2 implementations

▼ *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

- 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 `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 memory-safe implementations 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

`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:

```rust
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:

```rust
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:

```rust
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!

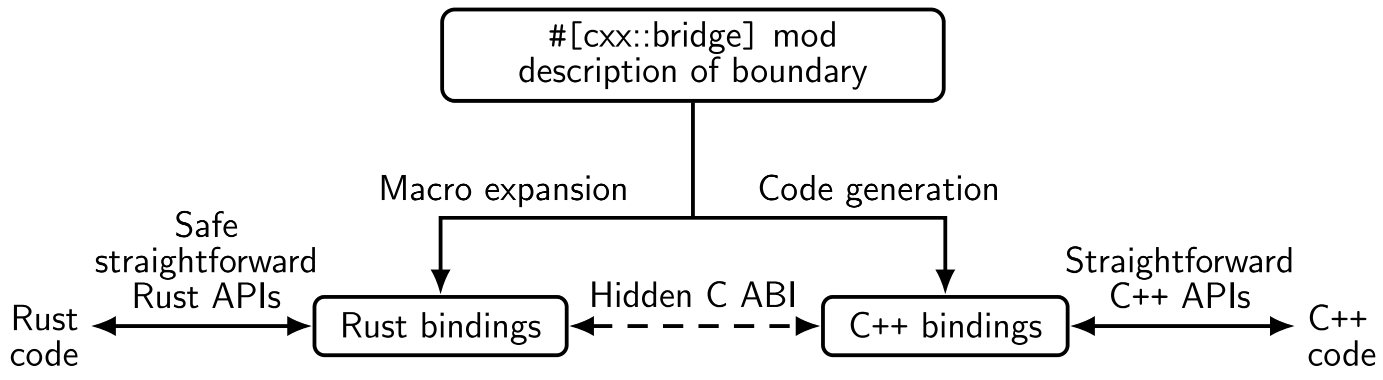
In your Chromium build:

- 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.

a Rust string from non-UTF-8 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.

```rust
#[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:

```rust
#[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:

```rust
#[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`)

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 `cxx` binding reference
- 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`:

```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:

```toml
[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.

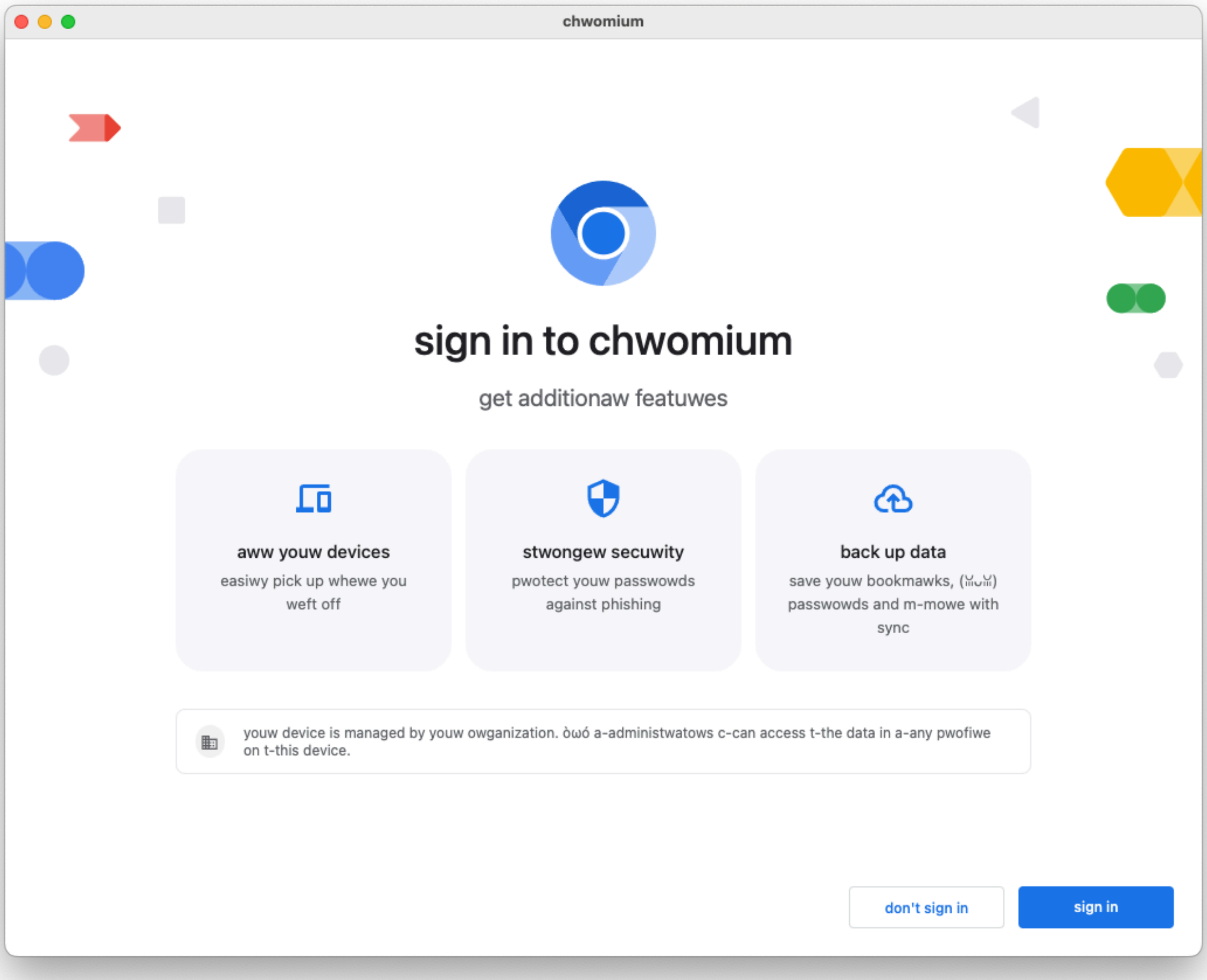
In fact, you already imported that crate in the previous exercise.

(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!


chwomium

sign in to chwomium

get additionaw featuwes

aww youw devices

easiwy pick up whewe you weft off

stwongew secuwity

pwotect youw passwowds against phishing

back up data

save youw bookmawks, (ᐡᴗᐡ) passwowds and m-mowe with sync

youw device is managed by youw owganization. òwó a-administwatows c-can access t-the data in a-any pwofiwe on t-this device.

▼ *Speaker Notes*

Students will likely need some hints here. Hints include:

- UTF-16 vs UTF-8. Students should be aware that Rust strings are always UTF-8, 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.

Exercise Solutions

Solutions to the Chromium exercises can be found in this series of CLs.

Or, if you'd prefer "standalone" solutions that don't require applying patchsets or integration with core Chromium code, you can find them in the `//chromium/src/codelabs/rust` subdirectory in Chromium.

Welcome to Bare Metal Rust

This is a standalone one-day course about bare-metal Rust, aimed at people who are familiar with the basics of Rust (perhaps from completing the Comprehensive Rust course), and ideally also have some experience with bare-metal programming in some other language such as C.

Today we will talk about 'bare-metal' Rust: running Rust code without an OS underneath us. This will be divided into several parts:

- What is `no_std` Rust?
- Writing firmware for microcontrollers.
- Writing bootloader / kernel code for application processors.
- Some useful crates for bare-metal Rust development.

For the microcontroller part of the course we will use the BBC micro:bit v2 as an example. It's a development board based on the Nordic nRF52833 microcontroller with some LEDs and buttons, an I2C-connected accelerometer and compass, and an on-board SWD debugger.

To get started, install some tools we'll need later. On gLinux or Debian:

```
sudo apt install gdb-multiarch libudev-dev picocom pkg-config qemu-system-arm
build-essential
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-
rs/releases/latest/download/probe-rs-tools-installer.sh | sh
```

And give users in the `plugdev` group access to the micro:bit programmer:

```
echo 'SUBSYSTEM=="hidraw", ATTRS{idVendor}=="0d28", MODE="0660",
GROUP="logindev", TAG+="uaccess"' |\
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

You should see "NXP ARM mbed" in the output of `lsusb` if the device is available. If you are using a Linux environment on a Chromebook, you will need to share the USB device with Linux, via `chrome://os-settings/crostini/sharedUsbDevices`.

On MacOS:

```
xcode-select --install
brew install gdb picocom qemu
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-
rs/releases/latest/download/probe-rs-tools-installer.sh | sh
```

no_std

`core`	`alloc`	`std`
• Slices, `&str`, `CStr` • `NonZeroU8`... • `Option`, `Result` • `Display`, `Debug`, `write!`... • `Iterator` • `Error` • `panic!`, `assert_eq!`... • `NonNull` and all the usual pointer-related functions • `Future` and `async`/`await` • `fence`, `AtomicBool`, `AtomicPtr`, `AtomicU32`... • `Duration`	• `Box`, `Cow`, `Arc`, `Rc` • `Vec`, `BinaryHeap`, `BtreeMap`, `LinkedList`, `VecDeque` • `String`, `CString`, `format!`	• `HashMap` • `Mutex`, `Condvar`, `Barrier`, `Once`, `RwLock`, `mpsc` • `File` and the rest of `fs` • `println!`, `Read`, `Write`, `Stdin`, `Stdout` and the rest of `io` • `Path`, `OsString` • `net` • `Command`, `Child`, `ExitCode` • `spawn`, `sleep` and the rest of `thread` • `SystemTime`, `Instant`

▼ *Speaker Notes*

- `HashMap` depends on RNG.
- `std` re-exports the contents of both `core` and `alloc`.

A minimal no_std program

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

▼ *Speaker Notes*

- This will compile to an empty binary.
- `std` provides a panic handler; without it we must provide our own.
- It can also be provided by another crate, such as `panic-halt`.
- Depending on the target, you may need to compile with `panic = "abort"` to avoid an error about `eh_personality`.
- Note that there is no `main` or any other entry point; it's up to you to define your own entry point. This will typically involve a linker script and some assembly code to set things up ready for Rust code to run.

alloc

To use `alloc` you must implement a global (heap) allocator.

```rust
#![no_main]
#![no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

const HEAP_SIZE: usize = 65536;
static mut HEAP: [u8; HEAP_SIZE] = [0; HEAP_SIZE];

pub fn entry() {
    // SAFETY: `HEAP` is only used here and `entry` is only called once.
    unsafe {
        // Give the allocator some memory to allocate.
        HEAP_ALLOCATOR.lock().init(&raw mut HEAP as usize, HEAP_SIZE);
    }

    // Now we can do things that require heap allocation.
    let mut v = Vec::new();
    v.push("A string".to_string());
}
```

▼ *Speaker Notes*

- `buddy_system_allocator` is a crate implementing a basic buddy system allocator. Other crates are available, or you can write your own or hook into your existing allocator.
- The const parameter of `LockedHeap` is the max order of the allocator; i.e. in this case it can allocate regions of up to 2**32 bytes.
- If any crate in your dependency tree depends on `alloc` then you must have exactly one global allocator defined in your binary. Usually this is done in the top-level binary crate.
- `extern crate panic_halt as _` is necessary to ensure that the `panic_halt` crate is linked in so we get its panic handler.
- This example will build but not run, as it doesn't have an entry point.

Microcontrollers

The `cortex_m_rt` crate provides (among other things) a reset handler for Cortex M microcontrollers.

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

Next we'll look at how to access peripherals, with increasing levels of abstraction.

▼ *Speaker Notes*

- The `cortex_m_rt::entry` macro requires that the function have type `fn() -> !`, because returning to the reset handler doesn't make sense.
- Run the example with `cargo embed --bin minimal`

Raw MMIO

Most microcontrollers access peripherals via memory-mapped IO. Let’s try turning on an LED on our micro:bit:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use core::mem::size_of;
use cortex_m_rt::entry;

/// GPIO port 0 peripheral address
const GPIO_P0: usize = 0x5000_0000;

// GPIO peripheral offsets
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// PIN_CNF fields
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u3
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u3
    // SAFETY: The pointers are to valid peripheral control registers, and
    // aliases exist.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
    }

    // Set pin 28 low and pin 21 high to turn the LED on.
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // SAFETY: The pointers are to valid peripheral control registers, and
    // aliases exist.

    loop {}
}
```

▼ *Speaker Notes*

- GPIO 0 pin 21 is connected to the first column of the LED matrix, and pin 28 to the first row.

Run the example with:

```
cargo embed --bin mmio
```

Peripheral Access Crates

`svd2rust` generates mostly-safe Rust wrappers for memory-mapped peripherals from CMSIS-SVD files.

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // Set pin 28 low and pin 21 high to turn the LED on.
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());

    loop {}
}
```

▼ *Speaker Notes*

- SVD (System View Description) files are XML files typically provided by silicon vendors that describe the memory map of the device.
  - They are organized by peripheral, register, field and value, with names, descriptions, addresses and so on.

- `cortex-m-rt` provides the vector table, among other things.
- If you `cargo install cargo-binutils` then you can run `cargo objdump --bin pac -- -d --no-show-raw-insn` to see the resulting binary.

Run the example with:

```
cargo embed --bin pac
```

HAL crates

HAL crates for many microcontrollers provide wrappers around various peripherals. These generally implement traits from `embedded-hal`.

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::gpio::{Level, p0};
use nrf52833_hal::pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // Create HAL wrapper for GPIO port 0.
    let gpio0 = p0::Parts::new(p.P0);

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // Set pin 28 low and pin 21 high to turn the LED on.
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

▼ *Speaker Notes*

- `set_low` and `set_high` are methods on the `embedded_hal` `OutputPin` trait.
- HAL crates exist for many Cortex-M and RISC-V devices, including various STM32, GD32, nRF, NXP, MSP430, AVR and PIC microcontrollers.

Run the example with:

```
cargo embed --bin hal
```

Board support crates

Board support crates provide a further level of wrapping for a specific board for convenience.

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use microbit::Board;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

▼ *Speaker Notes*

- In this case the board support crate is just providing more useful names, and a bit of initialization.
- The crate may also include drivers for some on-board devices outside of the microcontroller itself.
  - `microbit-v2` includes a simple driver for the LED matrix.

Run the example with:

```
cargo embed --bin board_support
```

The type state pattern

```rust
#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // Error, moved.
    let mut pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Leve
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // Error, moved.

    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Leve
    let _pin3: P0_03<Output<PushPull>> =
        gpio0.p0_03.into_push_pull_output(Level::Low);

    loop {}
}
```

▼ *Speaker Notes*

- Pins don't implement `Copy` or `Clone`, so only one instance of each can exist. Once a pin is moved out of the port struct, nobody else can take it.
- Changing the configuration of a pin consumes the old pin instance, so you can't use the old instance afterwards.
- The type of a value indicates the state it is in: e.g., in this case, the configuration state of a GPIO pin. This encodes the state machine into the type system and ensures that you don't try to use a pin in a certain way without properly configuring it first. Illegal state transitions are caught at compile time.
- You can call `is_high` on an input pin and `set_high` on an output pin, but not vice-versa.
- Many HAL crates follow this pattern.

embedded-hal

The `embedded-hal` crate provides a number of traits covering common microcontroller peripherals:

- GPIO
- PWM
- Delay timers
- I2C and SPI buses and devices

Similar traits for byte streams (e.g. UARTs), CAN buses and RNGs are broken out into `embedded-io`, `embedded-can` and `rand_core` respectively.

Other crates then implement drivers in terms of these traits, e.g. an accelerometer driver might need an I2C or SPI device instance.

▼ *Speaker Notes*

- The traits cover using the peripherals but not initializing or configuring them, as initialization and configuration is usually highly platform-specific.
- There are implementations for many microcontrollers, as well as other platforms such as Linux on Raspberry Pi.
- `embedded-hal-async` provides async versions of the traits.
- `embedded-hal-nb` provides another approach to non-blocking I/O, based on the `nb` crate.

`probe-rs` and `cargo-embed`

probe-rs is a handy toolset for embedded debugging, like OpenOCD but better integrated.

- SWD (Serial Wire Debug) and JTAG via CMSIS-DAP, ST-Link and J-Link probes
- GDB stub and Microsoft DAP (Debug Adapter Protocol) server
- Cargo integration

`cargo-embed` is a cargo subcommand to build and flash binaries, log RTT (Real Time Transfers) output and connect GDB. It's configured by an `Embed.toml` file in your project directory.

▼ *Speaker Notes*

- CMSIS-DAP is an Arm standard protocol over USB for an in-circuit debugger to access the CoreSight Debug Access Port of various Arm Cortex processors. It's what the on-board debugger on the BBC micro:bit uses.
- ST-Link is a range of in-circuit debuggers from ST Microelectronics, J-Link is a range from SEGGER.
- The Debug Access Port is usually either a 5-pin JTAG interface or 2-pin Serial Wire Debug.
- probe-rs is a library that you can integrate into your own tools if you want to.
- The Microsoft Debug Adapter Protocol lets VSCode and other IDEs debug code running on any supported microcontroller.
- cargo-embed is a binary built using the probe-rs library.
- RTT (Real Time Transfers) is a mechanism to transfer data between the debug host and the target through a number of ring buffers.

Debugging

Embed.toml:

```toml
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

In one terminal under `src/bare-metal/microcontrollers/examples/`:

```
cargo embed --bin board_support debug
```

In another terminal in the same directory:

On gLinux or Debian:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-
command="target remote :1338"
```

On MacOS:

```
arm-none-eabi-gdb target/thumbv7em-none-eabihf/debug/board_support --eval-
command="target remote :1338"
```

▼ *Speaker Notes*

In GDB, try running:

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

Other projects

- RTIC
  - "Real-Time Interrupt-driven Concurrency".
  - Shared resource management, message passing, task scheduling, timer queue.
- Embassy
  - `async` executors with priorities, timers, networking, USB.
- TockOS
  - Security-focused RTOS with preemptive scheduling and Memory Protection Unit support.
- Hubris
  - Microkernel RTOS from Oxide Computer Company with memory protection, unprivileged drivers, IPC.
- Bindings for FreeRTOS.

Some platforms have `std` implementations, e.g. esp-idf.

▼ *Speaker Notes*

- RTIC can be considered either an RTOS or a concurrency framework.
  - It doesn't include any HALs.
  - It uses the Cortex-M NVIC (Nested Virtual Interrupt Controller) for scheduling rather than a proper kernel.
  - Cortex-M only.
- Google uses TockOS on the Haven microcontroller for Titan security keys.
- FreeRTOS is mostly written in C, but there are Rust bindings for writing applications.

Exercises

We will read the direction from an I2C compass, and log the readings to a serial port.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

Compass

We will read the direction from an I2C compass, and log the readings to a serial port. If you have time, try displaying it on the LEDs somehow too, or use the buttons somehow.

Hints:

- Check the documentation for the `lsm303agr` and `microbit-v2` crates, as well as the micro:bit hardware.
- The LSM303AGR Inertial Measurement Unit is connected to the internal I2C bus.
- TWI is another name for I2C, so the I2C master peripheral is called TWIM.
- The LSM303AGR driver needs something implementing the `embedded_hal::i2c::I2c` trait. The `microbit::hal::Twim` struct implements this.
- You have a `microbit::Board` struct with fields for the various pins and peripherals.
- You can also look at the nRF52833 datasheet if you want, but it shouldn't be necessary for this exercise.

Download the exercise template and look in the `compass` directory for the following files.

src/main.rs:

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::{Delay, uarte::{Baudrate, Parity, Uarte}}, Board};

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Use the system timer as a delay provider.
    let mut delay = Delay::new(board.SYST);

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

Cargo.toml (you shouldn't need to change this):

```toml
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
cortex-m-rt = "0.7.5"
embedded-hal = "1.0.0"
lsm303agr = "1.1.0"
microbit-v2 = "0.16.0"
panic-halt = "1.0.0"
```

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

.cargo/config.toml (you shouldn't need to change this):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

See the serial output on Linux with:

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

Or on Mac OS something like (the device name may be slightly different):

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Use Ctrl+A Ctrl+Q to quit picocom.

Bare Metal Rust Morning Exercise

Compass

(back to exercise)

```rust
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use embedded_hal::digital::InputPin;
use lsm303agr::{
    AccelMode, AccelOutputDataRate, Lsm303agr, MagMode, MagOutputDataRate,
};
use microbit::Board;
use microbit::display::blocking::Display;
use microbit::hal::twim::Twim;
use microbit::hal::uarte::{Baudrate, Parity, Uarte};
use microbit::hal::{Delay, Timer};
use microbit::pac::twim0::frequency::FREQUENCY_A;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Use the system timer as a delay provider.
    let mut delay = Delay::new(board.SYST);

    // Set up the I2C controller and Inertial Measurement Unit.
    writeln!(serial, "Setting up IMU...").unwrap();
    let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(),
FREQUENCY_A::K100);
    let mut imu = Lsm303agr::new_with_i2c(i2c);
    imu.init().unwrap();
    imu.set_mag_mode_and_odr(
        &mut delay,
        MagMode::HighResolution,
        MagOutputDataRate::Hz50,
    )
    .unwrap();
    imu.set_accel_mode_and_odr(
        &mut delay,
        AccelMode::Normal,
        AccelOutputDataRate::Hz50,
    )
    .unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    let mut mode = Mode::Compass;
    let mut button_pressed = false;

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        while !(imu.mag_status().unwrap().xyz_new_data()
            && imu.accel_status().unwrap().xyz_new_data())
        {}
        let compass_reading = imu.magnetic_field().unwrap();
        let accelerometer_reading = imu.acceleration().unwrap();
        writeln!(
            serial,
            "{},{},{}\t{},{},{}",
            compass_reading.x_nt(),
            compass_reading.y_nt(),
            compass_reading.z_nt(),
            accelerometer_reading.x_mg(),
            accelerometer_reading.y_mg(),
            accelerometer_reading.z_mg(),
        )
        .unwrap();

        let mut image = [[0; 5]; 5];
        let (x, y) = match mode {
            Mode::Compass => (
                scale(-compass_reading.x_nt(), -COMPASS_SCALE, COMPASS_SCALE,
0, 4)
                    as usize,
                scale(compass_reading.y_nt(), -COMPASS_SCALE, COMPASS_SCALE,
0, 4)
                    as usize,
            ),
            Mode::Accelerometer => (
                scale(
                    accelerometer_reading.x_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
                scale(
                    -accelerometer_reading.y_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
            ),
        };
all LEDs
        // on.
        if board.buttons.button_a.is_low().unwrap() {
            if !button_pressed {
                mode = mode.next();
                display.show(&mut timer, [[255; 5]; 5], 200);
            }
            button_pressed = true;
        } else {
            button_pressed = false;
        }
    }
}

#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum Mode {
    Compass,
    Accelerometer,
}

impl Mode {
    fn next(self) -> Self {
        match self {
            Self::Compass => Self::Accelerometer,
            Self::Accelerometer => Self::Compass,
        }
    }
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) ->
i32 {
    let range_in = max_in - min_in;
    let range_out = max_out - min_out;
    let scaled = min_out + range_out * (value - min_in) / range_in;
    scaled.clamp(min_out, max_out)
}
```

Application processors

So far we've talked about microcontrollers, such as the Arm Cortex-M series. These are typically small systems with very limited resources.

Larger systems with more resources are typically called application processors, built around processors such as the ARM Cortex-A or Intel Atom.

For simplicity we'll just work with QEMU's aarch64 'virt' board.

▼ *Speaker Notes*

- Broadly speaking, microcontrollers don't have an MMU or multiple levels of privilege (exception levels on Arm CPUs, rings on x86).
- Application processors have more resources, and often run an operating system, instead of directly executing the target application on startup.
- QEMU supports emulating various different machines or board models for each architecture. The 'virt' board doesn't correspond to any particular real hardware, but is designed purely for virtual machines.
- We will still address this board as bare-metal, as if we were writing an operating system.

Getting Ready to Rust

Before we can start running Rust code, we need to do some initialization.

```
/**
 * This is a generic entry point for an image. It carries out the
 * operations required to prepare the loaded image to be run.
 * Specifically, it
 *
 * - sets up the MMU with an identity map of virtual to physical
 *   addresses, and enables caching
 * - enables floating point
 * - zeroes the bss section using registers x25 and above
 * - prepares the stack, pointing to a section within the image
 * - sets up the exception vector
 * - branches to the Rust `main` function
 *
 * It preserves x0-x3 for the Rust entry point, as these may contain
 * boot parameters.
 */
.section .init.entry, "ax"
.global entry
entry:
    /*
     * Load and apply the memory management configuration, ready to
     * enable MMU and caches.
     */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then
     * invalidate any potentially stale local TLB entries before they
     * start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed
     * until this has completed.
     */
    msr sctlr_el1, x30
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:  wfi
    b 2b
```

▼ *Speaker Notes*

This code is in `src/bare-metal/aps/examples/src/entry.S`. It's not necessary to understand this in detail – the takeaway is that typically some low-level setup is needed to meet Rust's expectations of the system.

- This is the same as it would be for C: initializing the processor state, zeroing the BSS, and setting up the stack pointer.
  - The BSS (block starting symbol, for historical reasons) is the part of the object file that contains statically allocated variables that are initialized to zero. They are omitted from the image, to avoid wasting space on zeroes. The compiler assumes that the loader will take care of zeroing them.
- The BSS may already be zeroed, depending on how memory is initialized and the image is loaded, but we zero it to be sure.
- We need to enable the MMU and cache before reading or writing any memory. If we don't:
  - Unaligned accesses will fault. We build the Rust code for the `aarch64-unknown-none` target that sets `+strict-align` to prevent the compiler from generating unaligned accesses, so it should be fine in this case, but this is not necessarily the case in general.

host has cacheable aliases to the same memory. Even if the host doesn't explicitly access the memory, speculative accesses can lead to cache fills, and then changes from one or the other will get lost when the cache is cleaned or the VM enables the cache. (Cache is keyed by physical address, not VA or IPA.)

- For simplicity, we just use a hardcoded pagetable (see `idmap.S`) that identity maps the first 1 GiB of address space for devices, the next 1 GiB for DRAM, and another 1 GiB higher up for more devices. This matches the memory layout that QEMU uses.
- We also set up the exception vector (`vbar_el1`), which we'll see more about later.
- All examples this afternoon assume we will be running at exception level 1 (EL1). If you need to run at a different exception level, you'll need to modify `entry.S` accordingly.

Inline assembly

Sometimes we need to use assembly to do things that aren't possible with Rust code. For example, to make an HVC (hypervisor call) to tell the firmware to power off the system:

```rust
#![no_main]
#![no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod asm;
mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // SAFETY: this only uses the declared registers and doesn't do anythir
    // with memory.
    unsafe {
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
            options(nomem, nostack)
        );
    }

    loop {}
}
```

(If you actually want to do this, use the `smccc` crate which has wrappers for all these functions.)

▼ *Speaker Notes*

- PSCI is the Arm Power State Coordination Interface, a standard set of functions to manage system and CPU power states, among other things. It is implemented by EL3 firmware and hypervisors on many systems.
- The `0 => _` syntax means initialize the register to 0 before running the inline assembly code, and ignore its contents afterwards. We need to use `inout` rather

- This `main` function needs to be `#[unsafe(no_mangle)]` and `extern "C"` because it is called from our entry point in `entry.S`.
  - Just `#[no_mangle]` would be sufficient but RFC3325 uses this notation to draw reviewer attention to attributes that might cause undefined behavior if used incorrectly.
- `_x0` – `_x3` are the values of registers `x0` – `x3`, which are conventionally used by the bootloader to pass things like a pointer to the device tree. According to the standard aarch64 calling convention (which is what `extern "C"` specifies to use), registers `x0` – `x7` are used for the first 8 arguments passed to a function, so `entry.S` doesn't need to do anything special except make sure it doesn't change these registers.
- Run the example in QEMU with `make qemu_psci` under `src/bare-metal/aps/examples`.

Volatile memory access for MMIO

- Use `pointer::read_volatile` and `pointer::write_volatile`.
- Never hold a reference to a location being accessed with these methods. Rust may read from (or write to, for `&mut`) a reference at any time.
- Use `&raw` to get fields of structs without creating an intermediate reference.

```rust
const SOME_DEVICE_REGISTER: *mut u64 = 0x800_0000 as _;
// SAFETY: Some device is mapped at this address.
unsafe {
    SOME_DEVICE_REGISTER.write_volatile(0xff);
    SOME_DEVICE_REGISTER.write_volatile(0x80);
    assert_eq!(SOME_DEVICE_REGISTER.read_volatile(), 0xaa);
}
```

▼ *Speaker Notes*

- Volatile access: read or write operations may have side-effects, so prevent the compiler or hardware from reordering, duplicating or eliding them.
  - Usually if you write and then read, e.g. via a mutable reference, the compiler may assume that the value read is the same as the value just written, and not bother actually reading memory.
- Some existing crates for volatile access to hardware do hold references, but this is unsound. Whenever a reference exists, the compiler may choose to dereference it.
- Use `&raw` to get struct field pointers from a pointer to the struct.
- For compatibility with old versions of Rust you can use the `addr_of!` macro instead.

Let’s write a UART driver

The QEMU ‘virt’ machine has a PL011 UART, so let’s write a driver for that.

```rust
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Minimal driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    base_address: *mut u8,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at
    /// given base address.
    ///
    /// # Safety
    ///
    /// The given base address must point to the 8 MMIO control registers c
    /// PL011 device, which must be mapped into the address space of the pr
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // SAFETY: We know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            self.base_address.write_volatile(byte);
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }

    fn read_flag_register(&self) -> u8 {
        // SAFETY: We know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile(
    }
}
```

▼ Speaker Notes

- Note that `Uart::new` is unsafe while the other methods are safe. This is because as long as the caller of `Uart::new` guarantees that its safety requirements are met (i.e. that there is only ever one instance of the driver for a given UART, and nothing else aliasing its address space), then it is always safe to call `write_byte` later because we can assume the necessary preconditions.
- We could have done it the other way around (making `new` safe but `write_byte` unsafe), but that would be much less convenient to use as every place that calls `write_byte` would need to reason about the safety
- This is a common pattern for writing safe wrappers of unsafe code: moving the burden of proof for soundness from a large number of places to a smaller number of places.

More traits

We derived the `Debug` trait. It would be useful to implement a few more traits too.

```rust
use core::fmt::{self, Write};

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
        }
        Ok(())
    }
}

// SAFETY: `Uart` just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

▼ *Speaker Notes*

- Implementing `Write` lets us use the `write!` and `writeln!` macros with our `Uart` type.
- `Send` is an auto-trait, but not implemented automatically because it is not implemented for pointers.

Using it

Let's write a small program using our driver to write to the serial console.

```rust
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod pl011_minimal;

use crate::pl011_minimal::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u8 = 0x900_0000 as _;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device,
    // nothing else accesses that address range.
    let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    system_off::<Hvc>().unwrap();
}
```

▼ *Speaker Notes*

- As in the inline assembly example, this `main` function is called from our entry point code in `entry.S`. See the speaker notes there for details.
- Run the example in QEMU with `make qemu_minimal` under `src/bare-metal/aps/examples`.

A better UART driver

The PL011 actually has more registers, and adding offsets to construct pointers to access them is error-prone and hard to read. Additionally, some of them are bit fields, which would be nice to access in a structured way.

Offset	Register name	Width
0x00	DR	12
0x04	RSR	4
0x18	FR	9
0x20	ILPR	8
0x24	IBRD	16
0x28	FBRD	6
0x2c	LCR_H	8
0x30	CR	16
0x34	IFLS	6
0x38	IMSC	11
0x3c	RIS	11
0x40	MIS	11
0x44	ICR	11
0x48	DMACR	3

▼ *Speaker Notes*

- There are also some ID registers that have been omitted for brevity.

Bitflags

The `bitflags` crate is useful for working with bitflags.

```rust
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
        const TXFE = 1 << 7;
        /// Ring indicator.
        const RI = 1 << 8;
    }
}
```

▼ *Speaker Notes*

- The `bitflags!` macro creates a newtype something like `struct Flags(u16)`, along with a bunch of method implementations to get and set flags.

Multiple registers

We can use a struct to represent the memory layout of the UART's registers.

```rust
#[repr(C, align(4))]
pub struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
```

▼ *Speaker Notes*

- `#[repr(C)]` tells the compiler to lay the struct fields out in order, following the same rules as C. This is necessary for our struct to have a predictable layout, as default Rust representation allows the compiler to (among other things) reorder fields however it sees fit.

Driver

Now let’s use the new `Registers` struct in our driver.

```rust
/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device wit
    /// given set of registers.
    ///
    /// # Safety
    ///
    /// The given pointer must point to the 8 MMIO control registers of a F
    /// device, which must be mapped into the address space of the process
    /// device memory and not have any other aliases.
    pub unsafe fn new(registers: *mut Registers) -> Self {
        Self { registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&mut self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // SAFETY: We know that self.registers points to the control regist
        // of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            (&raw mut (*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&mut self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            // SAFETY: We know that self.registers points to the control
            // registers of a PL011 device which is appropriately mapped.
            let data = unsafe { (&raw const (*self.registers).dr).read_vola
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        // SAFETY: We know that self.registers points to the control regist
        // of a PL011 device which is appropriately mapped.
        unsafe { (&raw const (*self.registers).fr).read_volatile() }
    }
```