6-1: FFI (Theory)

Artem Pavlov, TII, Abu Dhabi, 21.04.2025

Application Binary Interface (ABI)

- Interoperability convention between binary modules
- Describes register uses (calle/caller saved, return registers), stack organization, etc.
- The most commonly used is the “C ABI”

ABI-safe types

- Only ABI-safe types can be safely used in ABI interfaces:
  - integral or floating point primitive types
  - #[repr(C)]-annotated struct or union
  - #[repr(C)] or #[repr(Int)]-annotated enum with at least one variant and only fieldless variants
  - Plain references and raw pointers to ABI-safe types (not slices or dyn references!)
- If non-ABI-safe types are used in ABI interfaces compiler will raise improper_ctypes warnings

std::ffi

- std::ffi module provides common type definitions and aliases useful for FFI code
- Aliases for common C types: c_void, c_char, c_float, c_double, c_int, c_long, etc.
- Null-terminated strings: CStr and CString
- OS-specific string types: OsStr and OsString

The libc crate

- The standard Rust wrapper around libc
- Provides type aliases and definitions various C types
- Added as dependency by modifying Cargo.toml:
  ```
  [dependencies]
  libc = "0.2.0"
  ```

Calling foreign functions

```
use libc::size_t;

#[link(name = "snappy")]
extern "C" {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

Multiple functions in extern block

```rust
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern "C" {
    fn snappy_compress(
        input: *const u8,
        input_length: size_t,
        compressed: *mut u8,
        compressed_length: *mut size_t,
    ) -> c_int;
    fn snappy_uncompress(
        compressed: *const u8,
        compressed_length: size_t,
        uncompressed: *mut u8,
        uncompressed_length: *mut size_t,
    ) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(
        compressed: *const u8,
        compressed_length: size_t,
        result: *mut size_t,
    ) -> c_int;
    fn snappy_validate_compressed_buffer(
        compressed: *const u8,
        compressed_length: size_t
    ) -> c_int;
}
```

Wrapping extern functions

```rust
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe { snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0 }
}

pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

Linking using build.rs

- Build script is a piece of code which runs before crate compilation
- Build scripts are defined by creation build.rs file (in the same folder as Cargo.toml)
- Build script inputs are passed as environmental variables
- Build scripts can read and write into package’s directory
- Build scripts can communicate with Cargo by printing
- Library linking can be done by `println!("cargo:rustc-link-lib=snappy");`
- More information: https://doc.rust-lang.org/cargo/reference/build-scripts.html

*-sys crates

- In the Rust ecosystem it’s common to have one sys crate (e.g. libgit2-sys)
- Safe wrappers are built on top of a sys crate

Ownership and FFI

- C APIs often “take ownership” over raw pointers (e.g. for resource destruction)
- With Rust wrappers it’s important to correctly wrap such functions with consuming methods or Drop impls
- When Rust code provides an FFI interface, value can be dropped using std::ptr::drop_in_place

Exporting Rust FFI functions

```rust
// definition crate
#[no_mangle]
unsafe extern "C" fn add(a: u32, b: u32) -> u32 {
    a + b
}

// user crate
mod sys {
    extern "C" {
        fn add(a: u32, b: u32) -> u32;
    }
}

// safe wrapper
fn add(a: u32, b: u32) -> u32 {
    unsafe { sys::add(a, b) }
}
```

Opaque types

- For non-ABI-safe types a common pattern is to define “opaque” types
- “Opaque” types should have the same size and alignment as the wrapped type

```
#[repr(C, align(16))]
pub struct Foo {
    _private: [u8; 160],
}
extern "C" {
    fn foo(arg: *mut Foo);
}
```

Creating a shared library

- If crate exposes FFI interface, it can be compiled to a shared library and an object file using the following options in Cargo.toml:

```
[lib]
name = "add"
crate-type = ["staticlib", "cdylib"]
```

Panics and foreign code

- Stack unwinding from Rust code into foreign code resulted in undefined behavior with old version of Rust compiler.
- In modern Rust versions panics are stopped at an ABI boundary
- But it’s still worth to compile libraries with panic=”abort”
- Linking tricks can be used to prove that compiled library does not contain any panics

Catching panics at ABI boundary

```rust
use std::panic::catch_unwind;

fn may_panic() {
    if rand::random() {
        panic!("panic happens");
    }
}

#[no_mangle]
pub unsafe extern "C" fn no_panic() -> i32 {
    let result = catch_unwind(may_panic);
    match result {
        Ok(_) => 0,
        Err(_) => -1,
    }
}
```

rust-bindgen

- A helper utility and library for generating bindings from header files
- Usually used as a utility with generated bindings being published as sys crates
- Repository: https://github.com/rust-lang/rust-bindgen
- User guide: https://rust-lang.github.io/rust-bindgen

cbindgen

- An utility for generating C/C++ header files for Rust libraries
- Rust library must expose a public C API
- User guide: https://github.com/mozilla/cbindgen/blob/master/docs.md

Questions?