Rust programming

Module G: FFI

Learning objectives

- Working with C from Rust and vice versa
- be familiar with the C representation
- be familiar with the C calling convention
- Work with `` `cargo bindgen` ``
- Make nice rust APIs around C libraries
- Create python extensions

Content overview

- Calling convention and ABI
- using C from Rust
- using Rust from C
- `` `cargo bindgen` ``
- PyO3

Why do languages talk with each other?

- You get an ecosystem for free
- The other language has capabilities (performance, hardware access) that you don’t

Tight langugage coupling

Many languages can use code written in other languages

- JVM: Java, Scala, and Kotlin
- BEAM VM: Erlang and Elixir
- Bare Metal: Zig, D and Nim can import C code

The compiler checks names and types.

Rust cannot "just" import C code

- Idiomatic C is not idiomatic Rust
- C code cannot provide the guarantees that Rust expects
- maintaining half of a C compiler is not fun

Hence, a much looser coupling:

- generate assembly that is similar to what C generates
- have the linker stitch everything together

what if we kissed

3.2.3 Parameter Passing

After the argument values have been computed, they are placed either in registers or pushed on the stack. The way how values are passed is described in the following sections.

Definitions We first define a number of classes to classify arguments. The classes are corresponding to AMD64 register classes and defined as:

INTEGER This class consists of integral types that fit into one of the general purpose registers.

SSE The class consists of types that fit into a vector register.

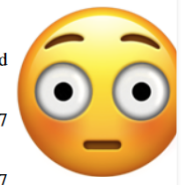
SSEUP The class consists of types that fit into a vector register and can be passed and returned in the upper bytes of it.

X87, X87UP These classes consists of types that will be returned via the x87 FPU.

COMPLEX_X87 This class consists of types that will be returned via the x87 FPU.

NO_CLASS This class is used as initializer in the algorithms. It will be used for padding and empty structures and unions.

MEMORY This class consists of types that will be passed and returned in memory via the stack.



Classification The size of each argument gets rounded up to eightbytes.[9]
The basic types are assigned their natural classes:

- Arguments of types (signed and unsigned) `_Bool`, `char`, `short`, `int`, `long`, `long long`, and pointers are in the INTEGER class.
- Arguments of types `float`, `double`, `_Decimal32`, `_Decimal64` and `__m64` are in class SSE.

[9] Therefore the stack will always be eightbyte aligned.

and we were both C

Rust & C disagree

- different calling conventions
- different memory layout

Idea: forward-declare the signature

In rust, this function can now be used like any other

```
1   extern "C" {
2       fn my_c_function(x: i32) -> bool;
3   }
```

The linker will stitch this declaration together with the definition

How to call a function

```
extern "C" {
    fn my_c_function(x: i32) -> bool;
}

pub fn main () {
    unsafe { my_c_function(42) };
}
```

generates this code for `main`:

```
example::main:
 push   rax                            # free up rax
 mov    edi,0x2a                       # put the argument into the edi register
 call   80b0 <example::my_c_function>  # call `my_c_function`
 pop    rax                            # restore rax
 ret                                   # return
```

Space vs Speed

We can compile this code in two ways

```
fn foo(vec: Vec<u8>) -> usize { vec.len() }

fn main() { foo(vec![]); }
```

Using 3 registers:

```
fn foo(ptr: *const u8, len: usize, cap: usize) -> usize {
    len
}
```

or using one register and indirection:

```
fn foo(vec: *const (usize, usize, usize)) -> usize {
    vec.1
}
```

Calling convention

- Rust and C make different choices on by-value vs. by-reference
- `` `extern "C"` `` forces rust to use the C calling convention
- The C calling convention is the lingua franca of calling between languages

C types != Rust types

- for some types, Rust and C agree on the representation

```
extern "C" {
    // integers
    fn is_even(x: i32) -> bool;

    // pointers
    fn is_null(ptr: *const u32) -> bool;
}


#[repr(u8)]
enum Color { R, G, B }

extern "C" {
    // tag-only enums
    fn circle_with_me(c: Color) -> Color;
}
```

C types != Rust types

- for others, we must explicitly pick the representation

```
#[repr(C)]
struct Point { x: f32, y: f32 }

extern "C" {
    // repr(C) structs
    fn h(p: Point) -> bool;
}

#[repr(transparent)]
struct Wrapper<T>(T);

extern "C" {
    // repr(transparent) structs, if the inner type is repr(C)
    fn h(w: Wrapper<u64>) -> bool;
}
```

C types != Rust types

- for others, we must explicitly pick the representation

```
#[repr(C)]
union U { int: i64, float: f64 }

extern "C" {
    // repr(C) unions
    fn i(u: U) -> bool;
}
```

C types != Rust types

- many types just don't work:
- enums like `Result` or `Option`
- owned collections like `String` and `Vec<T>`
- fat pointers like `&str` or `&[T]`

these need special, manual treatment

`cargo-bindgen`

Generates rust API bindings based on C header files

```
extern "C" {
    pub fn crypto_stream_salsa20_tweet_xor(
        arg1: *mut ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
        arg3: ::std::os::raw::c_ulonglong,
        arg4: *const ::std::os::raw::c_uchar,
        arg5: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn crypto_verify_16_tweet(
        arg1: *const ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn crypto_verify_32_tweet(
        arg1: *const ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
```

So far

C and Rust don’t just work together, we must

- tell rust the name and type of extern functions
- force rust to use the C calling convention
- use only types that have a C-compatible representation
- `cargo-bindgen` automates parts of this process

Using Rust from C

exposed functions look like this

```
#[no_mangle]
extern "C" fn sum(ptr: *const u64, len: usize) -> u64 {
    let slice = unsafe { std::slice::from_raw_parts(ptr, len) };

    slice.iter().sum()
}
```

Compiling rust into a static library requires some extra setup in the `Cargo.toml`.

Using Rust from Python

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

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

/// A Python module implemented in Rust. The name of this function must match
/// the `lib.name` setting in the `Cargo.toml`, else Python will not be able to
/// import the module.
#[pymodule]
fn string_sum(_py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum_as_string, m)?)?;
    Ok(())
}
```

```
$ python
>>> import string_sum
>>> string_sum.sum_as_string(5, 20)
'25'
```

Demo

Optional demo: `roc glue`

END

Code example

```
fn main() {
  println!("Hello world!");
}
```

RUN ▶ ••• DEBUG ⌄ STABLE ⌄ ••• SHARE TOOLS ⌄ CONFIG ⌄ ?

```
1 fn main() {
2   println!("Hello world!");
3 }
```