

Rust programming

Module 6: Rust for Systems Programming

Unit 1

Foreign Function Interface

Learning objectives

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 language 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 eightbytes⁹. 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**.

⁹Therefore 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

```
1  extern "C" {  
2      fn my_c_function(x: i32) -> bool;  
3  }  
4  
5  pub fn main () {  
6      unsafe { my_c_function(42) };  
7  }
```

generates this code for ``main`` :

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

Space vs Speed

We can compile this code in two ways

```
1  fn foo(vec: Vec<u8>) -> usize { vec.len() }  
2  
3  fn main() { foo(vec![]); }
```

Using 3 registers:

```
1  fn foo(ptr: *const u8, len: usize, cap: usize) -> usize {  
2      len  
3  }
```

or using one register and indirection:

```
1  fn foo(vec: *const (usize, usize, usize)) -> usize {  
2      vec.1  
3  }
```

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

```
1  extern "C" {
2      // integers
3      fn is_even(x: i32) -> bool;
4
5      // pointers
6      fn is_null(ptr: *const u32) -> bool;
7  }
8
9
10 #[repr(u8)]
11 enum Color { R, G, B }
12
13 extern "C" {
14     // tag-only enums
15     fn circle_with_me(c: Color) -> Color;
16 }
```

C types != Rust types

- for others, we must explicitly pick the representation

```
1  #[repr(C)]
2  struct Point { x: f32, y: f32 }
3
4  extern "C" {
5      // repr(C) structs
6      fn h(p: Point) -> bool;
7  }
8
9  #[repr(transparent)]
10 struct Wrapper<T>(T);
11
12 extern "C" {
13     // repr(transparent) structs, if the inner type is repr(C)
14     fn h(w: Wrapper<u64>) -> bool;
15 }
```

C types != Rust types

- for others, we must explicitly pick the representation

```
1  #[repr(C)]
2  union U { int: i64, float: f64 }
3
4  extern "C" {
5      // repr(C) unions
6      fn i(u: U) -> bool;
7  }
```

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

```
1  extern "C" {
2      pub fn crypto_stream_salsa20_tweet_xor(
3          arg1: *mut ::std::os::raw::c_uchar,
4          arg2: *const ::std::os::raw::c_uchar,
5          arg3: ::std::os::raw::c_ulonglong,
6          arg4: *const ::std::os::raw::c_uchar,
7          arg5: *const ::std::os::raw::c_uchar,
8      ) -> ::std::os::raw::c_int;
9  }
10 extern "C" {
11     pub fn crypto_verify_16_tweet(
12         arg1: *const ::std::os::raw::c_uchar,
13         arg2: *const ::std::os::raw::c_uchar,
14     ) -> ::std::os::raw::c_int;
15 }
16 extern "C" {
17     pub fn crypto_verify_32_tweet(
18         arg1: *const ::std::os::raw::c_uchar,
19         arg2: *const ::std::os::raw::c_uchar,
20     ) -> ::std::os::raw::c_int;
21 }
```


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

```
1  #[no_mangle]
2  extern "C" fn sum(ptr: *const u64, len: usize) -> u64 {
3      let slice = unsafe { std::slice::from_raw_parts(ptr, len) };
4
5      slice.iter().sum()
6  }
```

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

Summary