# Rust Inside Other Languages

For our third project, we're going to choose something that shows off one of Rust's greatest strengths: a lack of a substantial runtime.

As organizations grow, they increasingly rely on a multitude of programming languages. Different programming languages have different strengths and weaknesses, and a polyglot stack lets you use a particular language where its strengths make sense and a different one where it's weak.

A very common area where many programming languages are weak is in runtime performance of programs. Often, using a language that is slower, but offers greater programmer productivity, is a worthwhile trade-off. To help mitigate this, they provide a way to write some of your system in C and then call that C code as though it were written in the higher-level language. This is called a 'foreign function interface', often shortened to 'FFI'.

Rust has support for FFI in both directions: it can call into C code easily, but crucially, it can also be called *into* as easily as C. Combined with Rust's lack of a garbage collector and low runtime requirements, this makes Rust a great candidate to embed inside of other languages when you need that extra oomph.

There is a whole chapter devoted to FFI (ffi.html) and its specifics elsewhere in the book, but in this chapter, we'll examine this particular use-case of FFI, with examples in Ruby, Python, and JavaScript.

## The problem

There are many different projects we could choose here, but we're going to pick an example where Rust has a clear advantage over many other languages: numeric computing and threading.

Many languages, for the sake of consistency, place numbers on the heap, rather than on the stack. Especially in languages that focus on object-oriented programming and use garbage collection, heap allocation is the default. Sometimes optimizations can stack allocate particular numbers, but rather than relying on an optimizer to do its job, we may want to ensure that we're always using primitive number types rather than some sort of object type.

Second, many languages have a 'global interpreter lock' (GIL), which limits concurrency in many situations. This is done in the name of safety, which is a positive effect, but it limits the amount of work that can be done at the same time, which is a big negative.

To emphasize these two aspects, we're going to create a little project that uses these two aspects heavily. Since the focus of the example is to embed Rust into other languages, rather than the problem itself, we'll just use a toy example:

> Start ten threads. Inside each thread, count from one to five million. After all ten threads are finished, print out 'done!'.

I chose five million based on my particular computer. Here's an example of this code in Ruby:

```
threads = []

10.times do
  threads << Thread.new do
    count = 0

    5_000_000.times do
      count += 1
    end

    count
  end
end

threads.each do |t|
  puts "Thread finished with count=#{t.value}"
end
puts "done!"
```

Try running this example, and choose a number that runs for a few seconds. Depending on your computer's hardware, you may have to increase or decrease the number.

On my system, running this program takes `2.156` seconds. And, if I use some sort of process monitoring tool, like `top`, I can see that it only uses one core on my machine. That's the GIL kicking in.

While it's true that this is a synthetic program, one can imagine many problems that are similar to this in the real world. For our purposes, spinning up a few busy threads represents some sort of parallel, expensive computation.

## A Rust library

Let's rewrite this problem in Rust. First, let's make a new project with Cargo:

```
$ cargo new embed
$ cd embed
```

This program is fairly easy to write in Rust:

```
use std::thread;

fn process() {
    let handles: Vec<_> = (0..10).map(|_| {
        thread::spawn(|| {
            let mut x = 0;
            for _ in 0..5_000_000 {
                x += 1
            }
            x
        })
    }).collect();

    for h in handles {
        println!("Thread finished with count={}",
        h.join().map_err(|_| "Could not join a thread!").unwrap());
    }
}
```

Some of this should look familiar from previous examples. We spin up ten threads, collecting them into a `handles` vector. Inside of each thread, we loop five million times, and add one to `x` each time. Finally, we join on each thread.

Right now, however, this is a Rust library, and it doesn't expose anything that's callable from C. If we tried to hook this up to another language right now, it wouldn't work. We only need to make two small changes to fix this, though. The first is to modify the beginning of our code:

```
#[no_mangle]
pub extern fn process() {
```

We have to add a new attribute, `no_mangle`. When you create a Rust library, it changes the name of the function in the compiled output. The reasons for this are outside the scope of this tutorial, but in order for other languages to know how to call the function, we can't do that. This attribute turns that behavior off.

The other change is the `pub extern`. The `pub` means that this function should be callable from outside of this module, and the `extern` says that it should be able to be called from C. That's it! Not a whole lot of change.

The second thing we need to do is to change a setting in our `Cargo.toml`. Add this at the bottom:

```
[lib]
name = "embed"
crate-type = ["dylib"]
```

This tells Rust that we want to compile our library into a standard dynamic library. By default, Rust compiles an 'rlib', a Rust-specific format.

Let's build the project now:

```
$ cargo build --release
   Compiling embed v0.1.0 (file:///home/steve/src/embed)
```

We've chosen `cargo build --release`, which builds with optimizations on. We want this to be as fast as possible! You can find the output of the library in `target/release`:

```
$ ls target/release/
build  deps  examples  libembed.so  native
```

That `libembed.so` is our 'shared object' library. We can use this file just like any shared object library written in C! As an aside, this may be `embed.dll` or `libembed.dylib`, depending on the platform.

Now that we've got our Rust library built, let's use it from our Ruby.

## Ruby

Open up an `embed.rb` file inside of our project, and do this:

```
require 'ffi'

module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
  attach_function :process, [], :void
end

Hello.process

puts 'done!'
```

Before we can run this, we need to install the `ffi` gem:

```
$ gem install ffi # this may need sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

And finally, we can try running it:

```
$ ruby embed.rb
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
Thread finished with count=5000000
done!
done!
$
```

Whoa, that was fast! On my system, this took `0.086` seconds, rather than the two seconds the pure Ruby version took. Let's break down this Ruby code:

```
require 'ffi'
```

We first need to require the `ffi` gem. This lets us interface with our Rust library like a C library.

```
module Hello
  extend FFI::Library
  ffi_lib 'target/release/libembed.so'
```

The `Hello` module is used to attach the native functions from the shared library. Inside, we `extend` the necessary `FFI::Library` module and then call `ffi_lib` to load up our shared object library. We just pass it the path that our library is stored, which, as we saw before, is

`target/release/libembed.so`.

```
attach_function :process, [], :void
```

The `attach_function` method is provided by the FFI gem. It's what connects our `process()` function in Rust to a Ruby function of the same name. Since `process()` takes no arguments, the second parameter is an empty array, and since it returns nothing, we pass `:void` as the final argument.

```
Hello.process
```

This is the actual call into Rust. The combination of our `module` and the call to `attach_function` sets this all up. It looks like a Ruby function but is actually Rust!

```
puts 'done!'
```

Finally, as per our project's requirements, we print out `done!`.

That's it! As we've seen, bridging between the two languages is really easy, and buys us a lot of performance.

Next, let's try Python!

## Python

Create an `embed.py` file in this directory, and put this in it:

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembed.so")

lib.process()

print("done!")
```

Even easier! We use `cdll` from the `ctypes` module. A quick call to `LoadLibrary` later, and we can call `process()`.

On my system, this takes `0.017` seconds. Speedy!

# Node.js

Node isn't a language, but it's currently the dominant implementation of server-side JavaScript.

In order to do FFI with Node, we first need to install the library:

```
$ npm install ffi
```

After that installs, we can use it:

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembed', {
  'process': ['void', []]
});

lib.process();

console.log("done!");
```

It looks more like the Ruby example than the Python example. We use the `ffi` module to get access to `ffi.Library()`, which loads up our shared object. We need to annotate the return type and argument types of the function, which are `void` for return and an empty array to signify no arguments. From there, we just call it and print the result.

On my system, this takes a quick `0.092` seconds.

# Conclusion

As you can see, the basics of doing this are *very* easy. Of course, there's a lot more that we could do here. Check out the FFI (ffi.html) chapter for more details.