

Calling Functions Across Languages



@rtfeldman

“This task would be a lot easier if I could use another language”

1. I want a **library** that's in another language
2. I want more **performance** than I can get here
3. I want more **ergonomics** at the expense of speed
4. I want to incrementally **transition** languages



```
import java.foo.bar;
```



HTTP



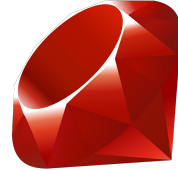
SORCERY?!

Fortran
1957

dot_product(array1, array2)



HTTP



Fortran

Peel back layers of abstraction



1. Build HTTP request **string**
2. Send that string to a **socket**

JS

GET **/dot-product** HTTP/1.1

Host: **localhost**

Connection: close

Content-Type: application/json

Content-Length: 28

[[1, 2, 3, 4], [5, 6, 7, 8]]

dot_product(a, b)



HTTP/1.1 200 OK

Content-Type: application/json

Content-Length: 2

Connection: close

70

JS

build HTTP request **string**, send to **socket**

```
const socket = new net.Socket();
```

```
socket.connect(3000, "localhost", () => {  
  const requestString = buildHttpRequest("GET", {  
    path: "/dot-product",  
    headers: ...,  
    body: JSON.stringify([array1, array2]),  
  });
```

```
    socket.write(requestString);  
  });
```



deserialize args from socket, **serialize** answer to socket

```
server = TCPServer.new("localhost", 3000)
loop do
  client = server.accept
  req = parse_http_request(read_all(client))
  array1, array2 = JSON.parse(req.body)
  answer = dot_product(array1, array2)
  answer_as_json = JSON.dump(answer)
  response_string = build_http_response(json)
  client.write(response_string)
  client.close
end
```

GET **/dot-product** HTTP/1.1
Host: localhost
Connection: close
Content-Type: application/json
Content-Length: 28

JS

[[1, 2, 3, 4], [5, 6, 7, 8]]

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 2
Connection: close



JS

[[1, 2, 3, 4], [5, 6, 7, 8]]

70



JS

build HTTP request **string**, send to **socket**

```
const socket = new net.Socket();
```

```
socket.connect(3000, "localhost", () => {  
  const requestString = buildHttpRequest("GET", {  
    path: "/dot-product",  
    headers: ...,  
    body: JSON.stringify([array1, array2]),  
  });
```

```
    socket.write(requestString);  
  });
```

JS

build HTTP request **string**, send to **socket**

```
const socket = new net.Socket();
```

```
socket.connect(3000, "localhost", () => {  
    const requestString = JSON.stringify(args);
```

```
    socket.write(requestString);  
});
```



deserialize args from socket, **serialize** answer to socket

```
server = TCPServer.new("localhost", 3000)
loop do
  client = server.accept
  req = parse_http_request(read_all(client))
  array1, array2 = JSON.parse(req.body)
  answer = dot_product(array1, array2)
  answer_as_json = JSON.dump(answer)
  response_string = build_http_response(json)
  client.write(response_string)
  client.close
end
```



deserialize args from socket, **serialize** answer to socket

```
server = TCPServer.new("localhost", 3000)
```

```
loop do
```

```
  client = server.accept
```

```
  array1, array2 = JSON.parse(read_all(client))
```

```
  answer = dot_product(array1, array2)
```

```
  answer_as_json = JSON.dump(answer)
```

```
  client.write(answer_as_json)
```

```
  client.close
```

```
end
```



deserialize args from socket, **serialize** answer to socket

```
server = TCPServer.new("localhost", 3000)
```

```
loop do
```

```
  client = server.accept
```

```
  array1, array2 = deserialize(read_all(client))
```

```
  answer = dot_product(array1, array2)
```

```
  response_bytes = serialize(answer)
```

```
  client.write(response_bytes)
```

```
  client.close
```

```
end
```



deserialize args from socket, **serialize** answer to socket

```
server = TCPServer.new("localhost", 3000)
loop do
  client = server.accept

  array1, array2 = deserialize(read_all(client))
  answer = dot_product(array1, array2)
  response_bytes = serialize(answer)

  client.write(response_bytes)
  client.close
end
```



deserialize args from socket, **serialize** answer to socket

```
server = UNIXServer.new("/tmp/blah.sock")
```

```
loop do Inter-Process Communication (IPC)
```

```
  client = server.accept
```

```
  array1, array2 = deserialize(read_all(client))
```

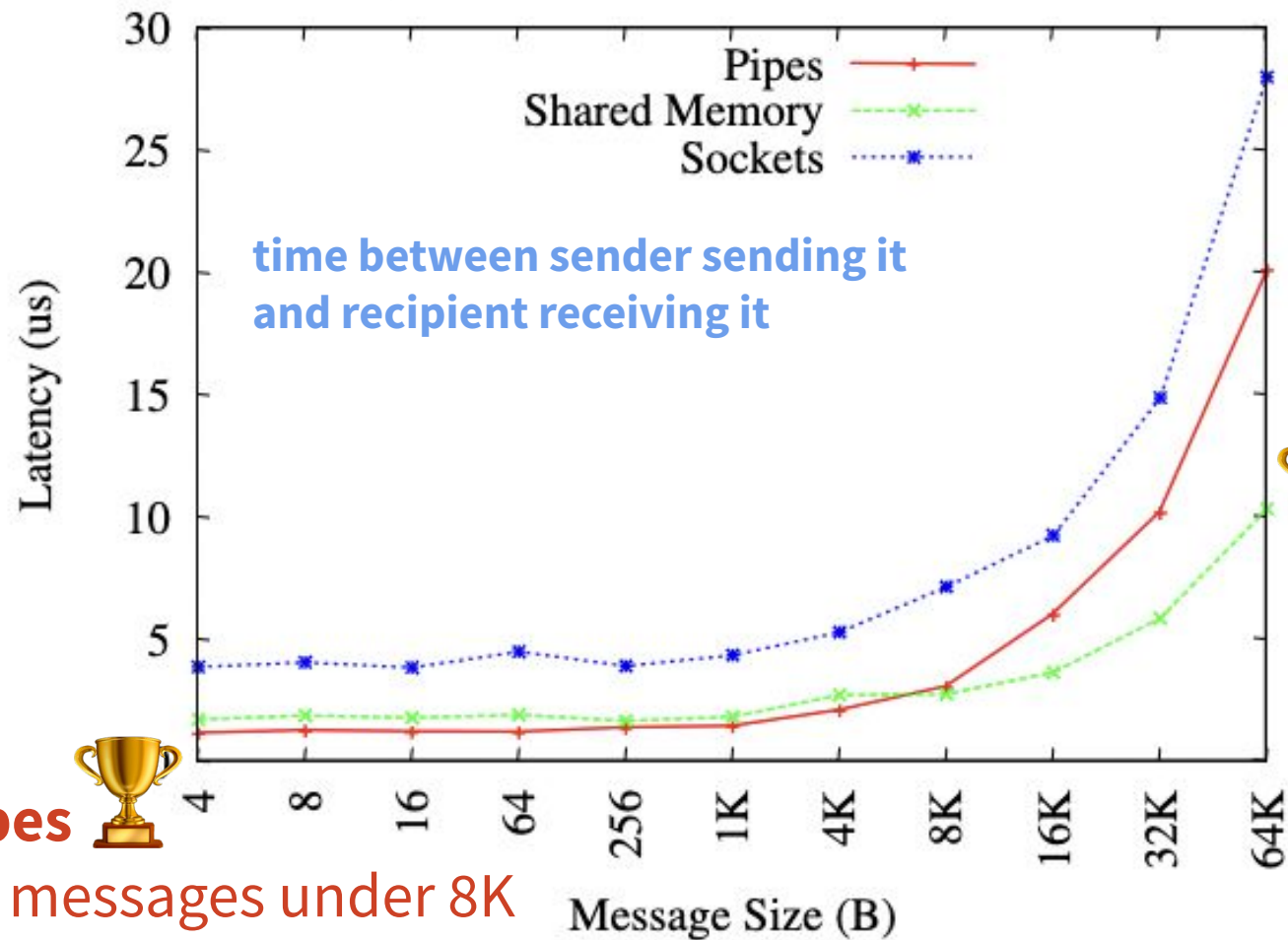
```
  answer = dot_product(array1, array2)
```

```
  response_bytes = serialize(answer)
```

```
  client.write(response_bytes)
```

```
  client.close
```

```
end
```



time between sender sending it
and recipient receiving it



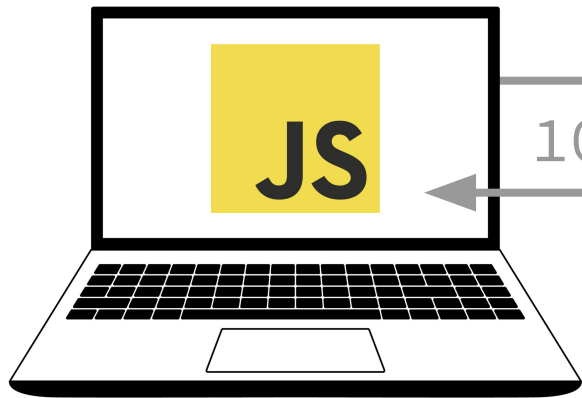
shared memory
for messages 8K+

pipes



for messages under 8K

serialize to URL + headers + request body

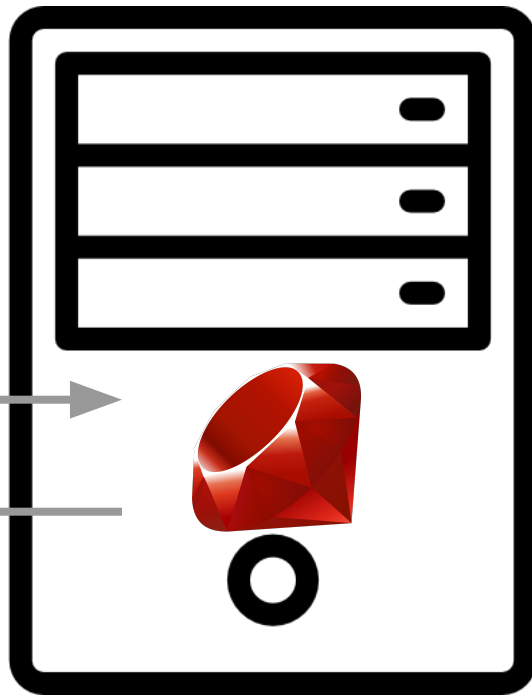


deserialize response

HTTP

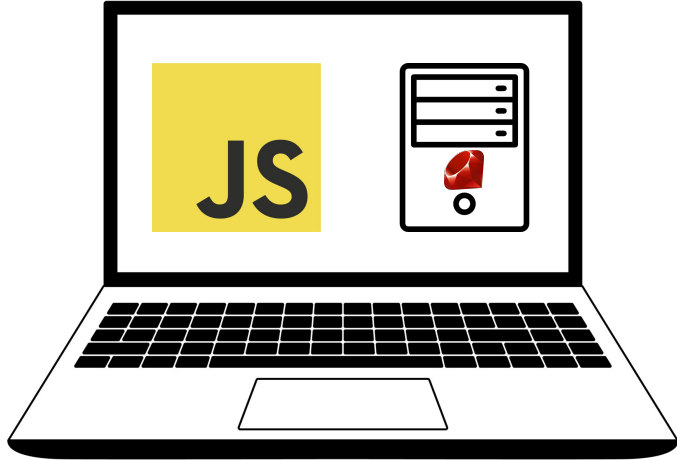
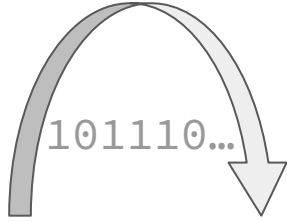
1001101110010...

deserialize request



serialize to response headers + body

HTTP



localhost

serialize args to TCP socket

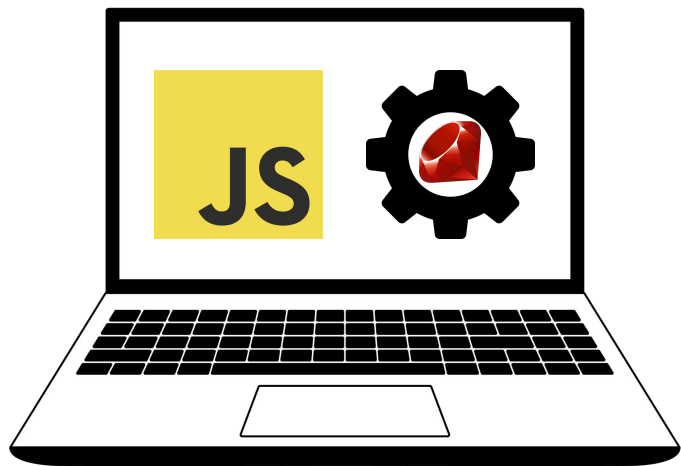
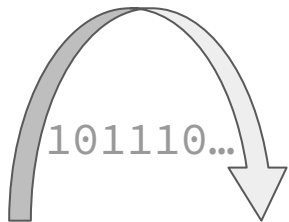
deserialize args from TCP socket

(**actual Ruby function** runs)

serialize answer to TCP socket

deserialize answer from TCP socket

IPC



serialize args to socket

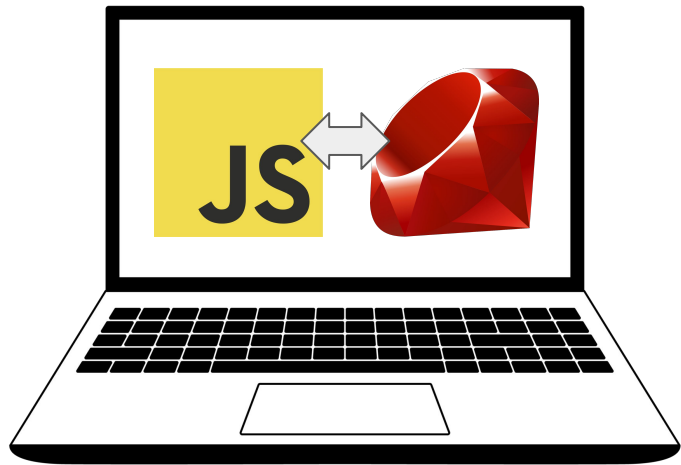
deserialize args from socket

(**actual Ruby function** runs)

serialize answer to socket

deserialize answer from socket

single process



serialize args to socket

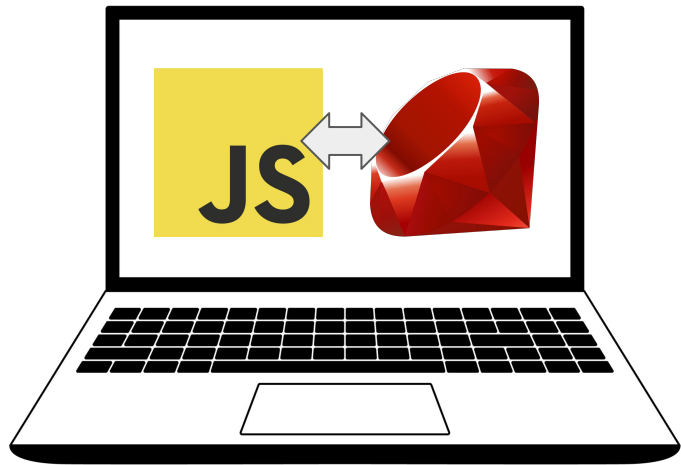
deserialize args from socket

(**actual Ruby function** runs)

serialize answer to socket

deserialize answer from socket

single process



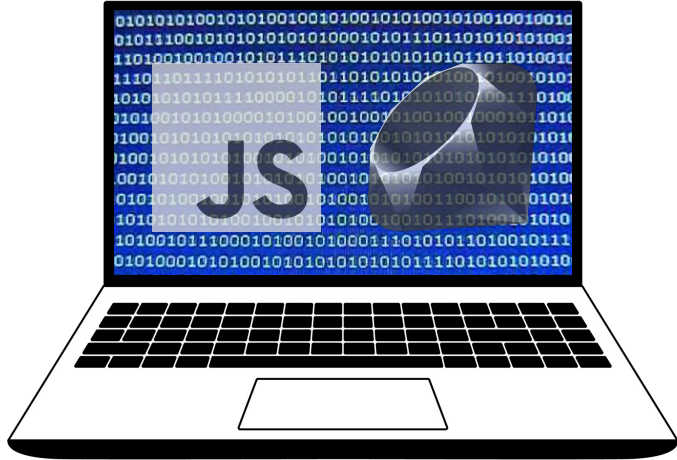
(**actual Ruby function** runs)

dot_product(a, b)



JSQ

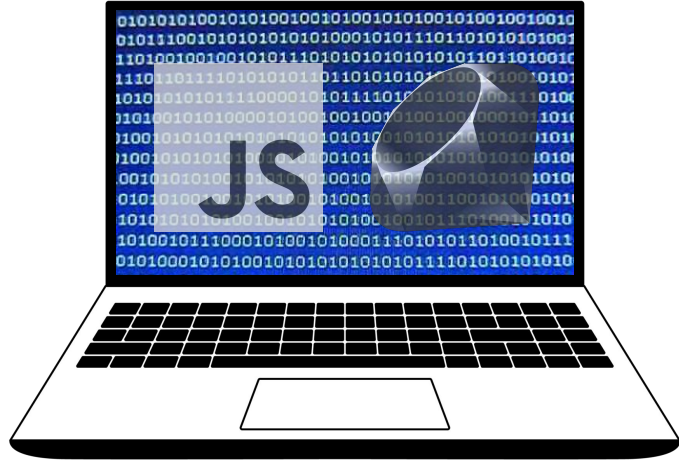
single process



(**actual Ruby function** runs)

`dot_product(a, b)` 🚀

single process

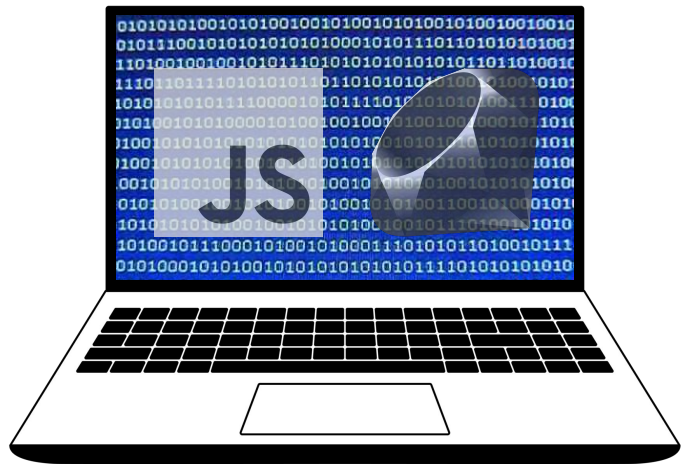


executable binary

```
0101110101011010101010001001
0101001011010101011010101010
10010111010101101010100010
0101010010110101011010101010
10100101110101011010101000
100101010010110101010101000
1001010100101101010101101010
1010100100011101010110101010
1000100101010010111010101011
0101010101001011110101011010
1010100010010101001011010101
0111010101010100101110101011
```

executable binary

single process

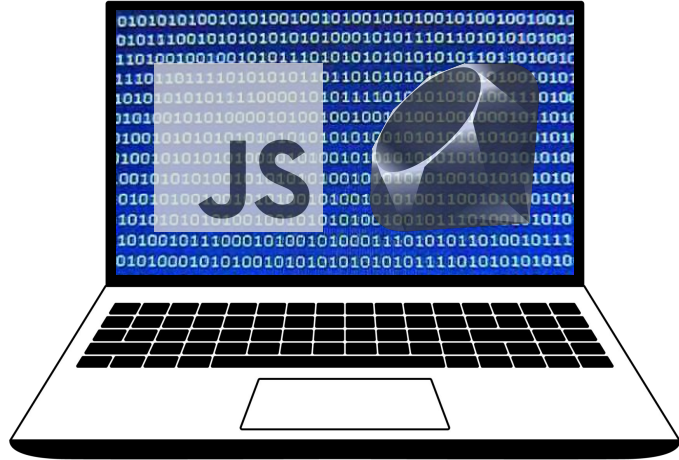


[

```
"add next two entries",  
42,  
123,  
"add next entry to answer",  
200,  
"jump to index in next",  
3,
```

]

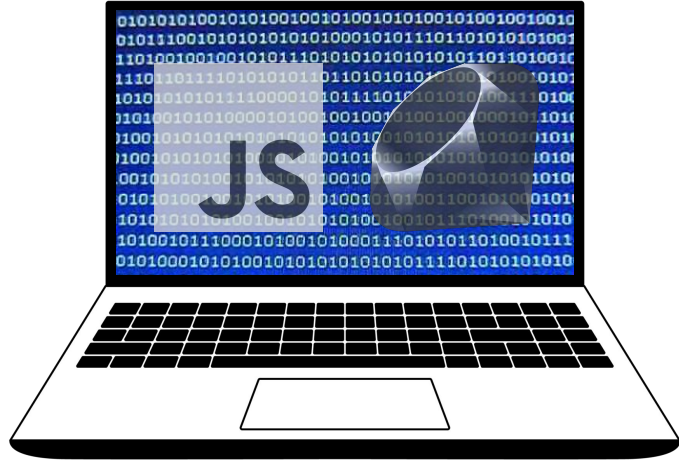
single process



executable binary

```
[  
0    "add next two entries",  
1    42,  
2    123,  
3    "add next entry to answer",  
4    200,  
5    "jump to index in next",  
6    3,  
]
```

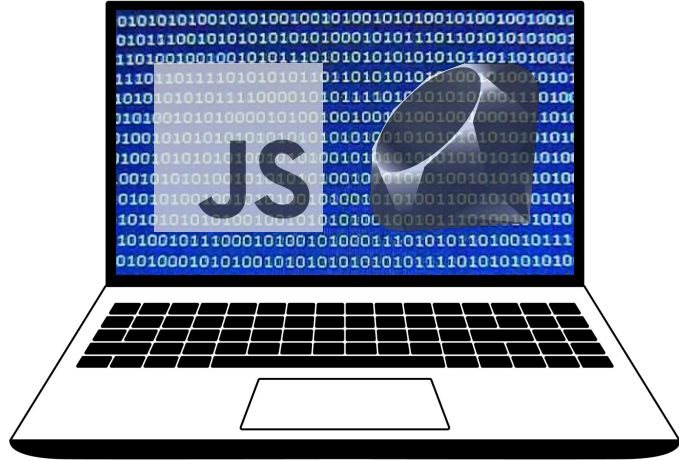
single process



executable binary

```
[  
0 → "add next two entries",  
1 42,  
2 123,  
3 "add next entry to answer",  
4 200,  
5 "jump to index in next",  
6 3,  
]
```

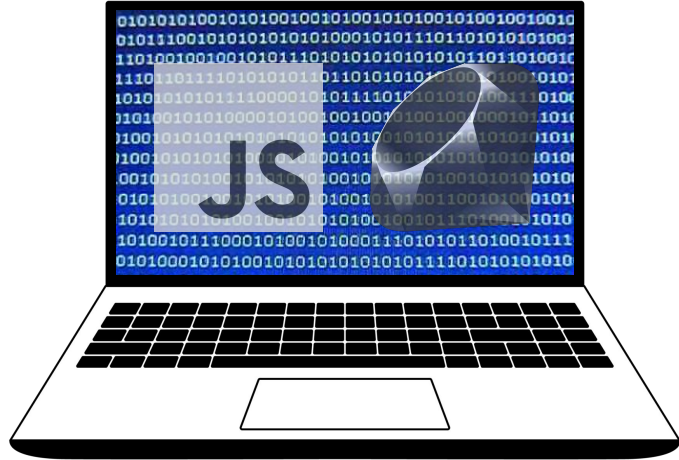
single process



executable binary

```
[  
0    "add next two entries",  
1    42,  
2    123,  
3 →  "add next entry to answer",  
4    200,  
5    "jump to index in next",  
6    3,  
]
```

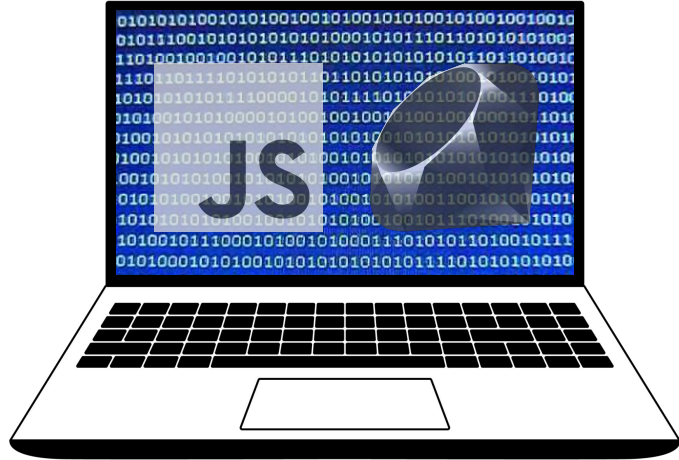
single process



executable binary

```
[  
0    "add next two entries",  
1    42,  
2    123,  
3    "add next entry to answer",  
4    200,  
5    "jump to index in next",  
6    3,  
]
```

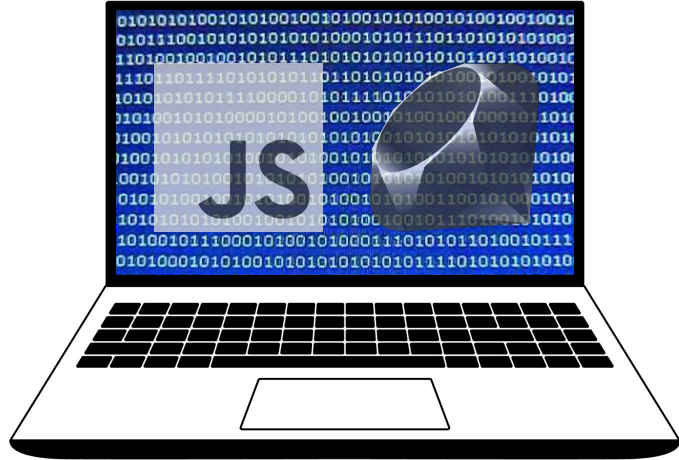
single process



executable binary

```
[  
0    "add next two entries",  
1    42,  
2    123,  
3 →  "add next entry to answer",  
4    200,  
5    "jump to index in next",  
6    3,  
]
```

single process

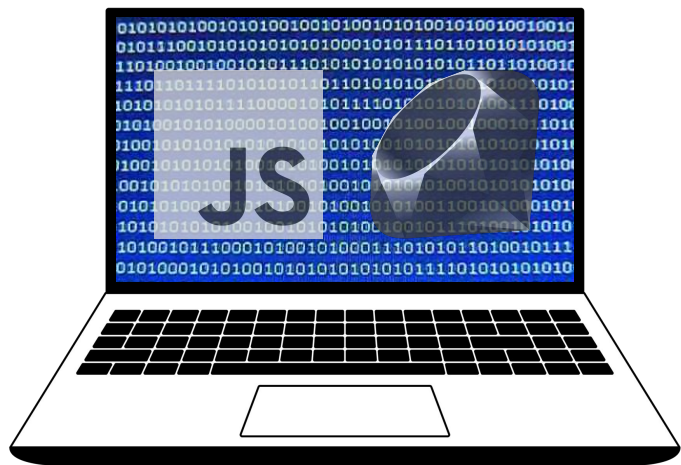


executable binary

```
[  
0    "add next two entries",  
1    42,  
2    123,  
3    "add next entry to answer",  
4    200,  
5    "jump to index in next",  
6    3,  
]
```

executable binary

single process



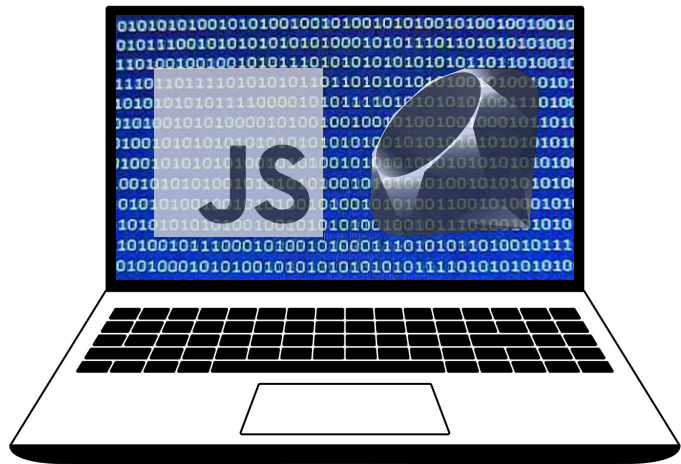
[

```
"add next two entries",  
42,  
123,  
"add next entry to answer",  
200,  
"jump to index in next",  
3,  
...
```

]

executable binary

single process



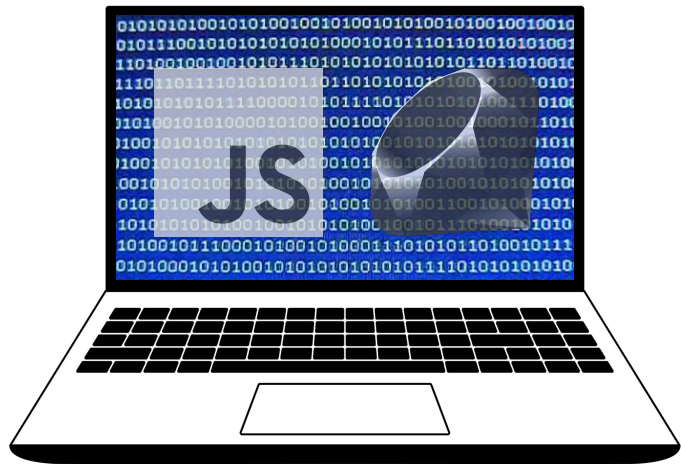
[

```
5, // (add next two entries)
42,
123,
"add next entry to answer",
200,
"jump to index in next",
3,
...
```

]

executable binary

single process



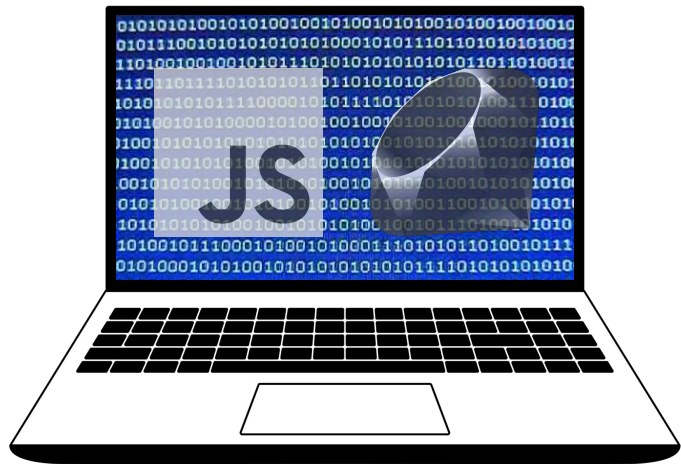
[

```
5, // (add next two entries)
42,
123,
17, // (add next to answer)
200,
"jump to index in next",
3,
...
```

]

executable binary

single process



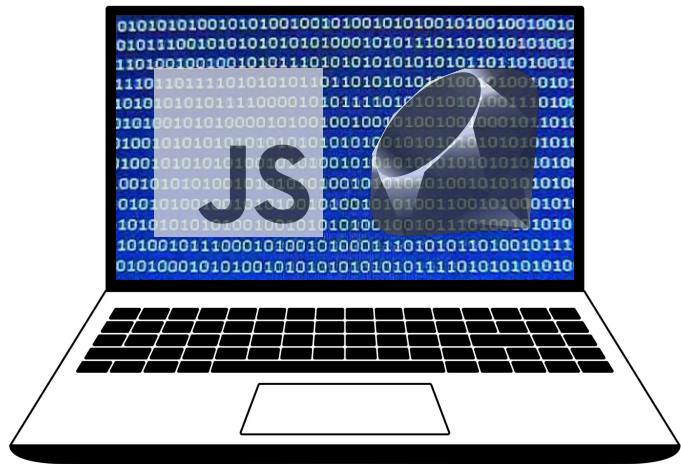
[

```
5, // (add next two entries)
42,
123,
17, // (add next to answer)
200,
29, // (jump to index)
3,
...
```

]

executable binary

single process



[

00000101

42,

123,

17, // (add next to answer)

200,

29, // (jump to index)

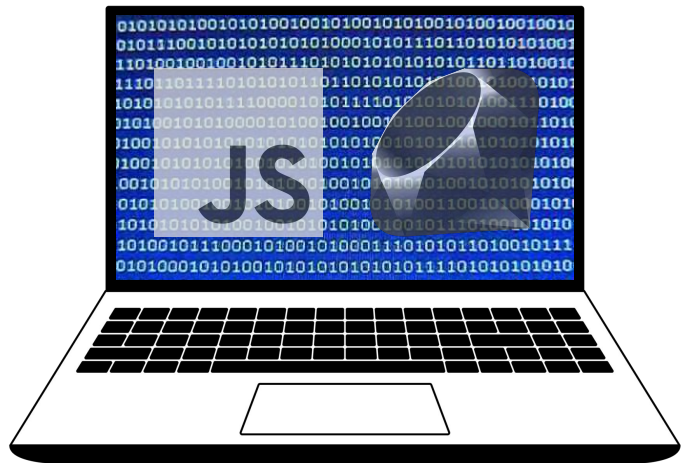
3,

...

]

executable binary

single process



[

00000101

00101010

01111011

17, // (add next to answer)

200,

29, // (jump to index)

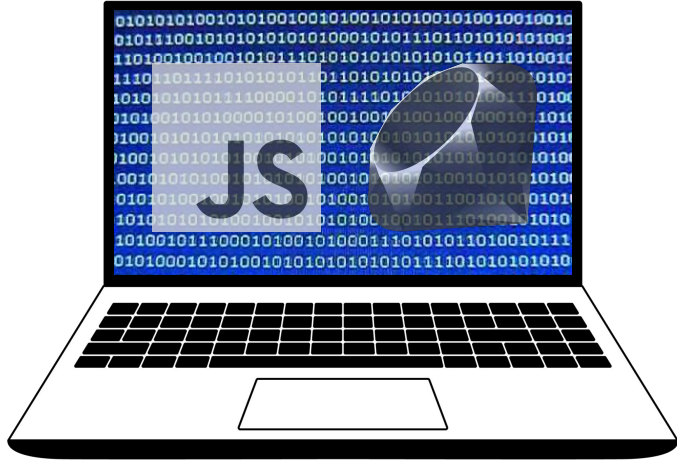
3,

...

]

executable binary

single process



00000101

00101010

01111011

00010001

11001000

00011101

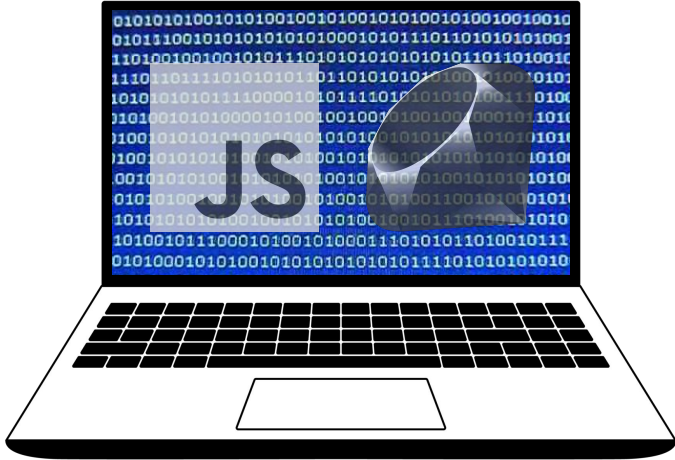
00000011

...

]

executable binary

single process



00000101

00101010

01111011

00010001

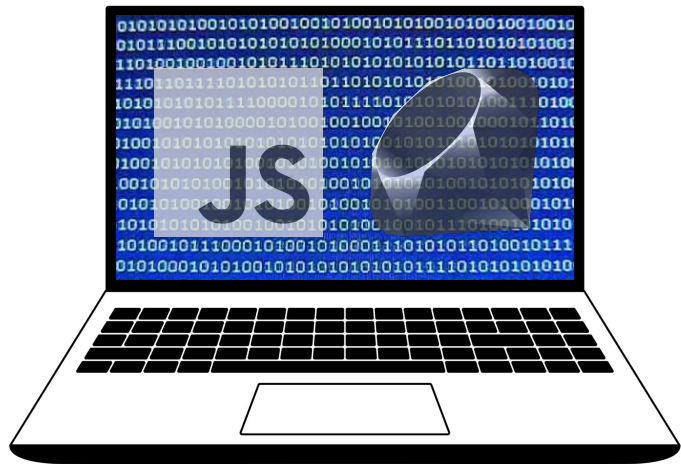
11001000

00011101

00000011

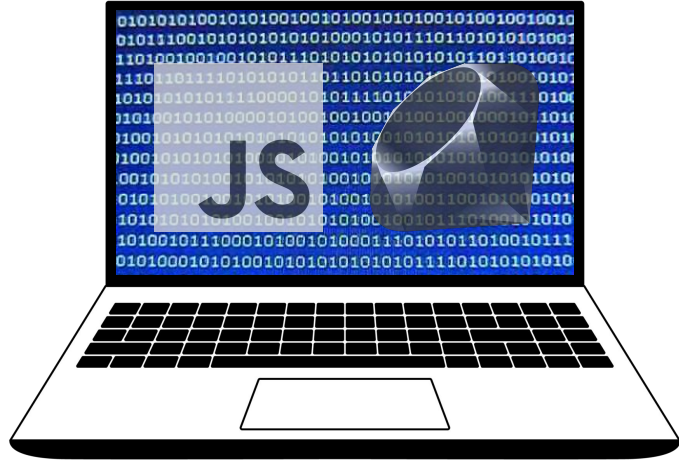
executable binary

single process



```
"add next two entries",  
42,  
123,  
"add next entry to answer",  
200,  
"jump to index in next",  
3,  
...
```

single process



executable binary

```
0101110101011010101010001001
0101001011010101011010101010
10010111010101101010100010
0101010010110101011010101010
10100101110101011010101000
100101010010110101010101000
1001010100101101010101101010
1010100100011101010110101010
1000100101010010111010101011
0101010101001011110101011010
1010100010010101001011010101
0111010101010100101110101011
```

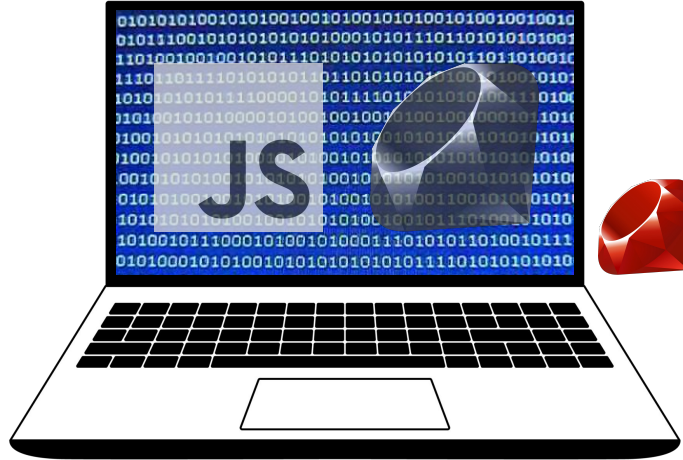
executable binary

"jump"

single process

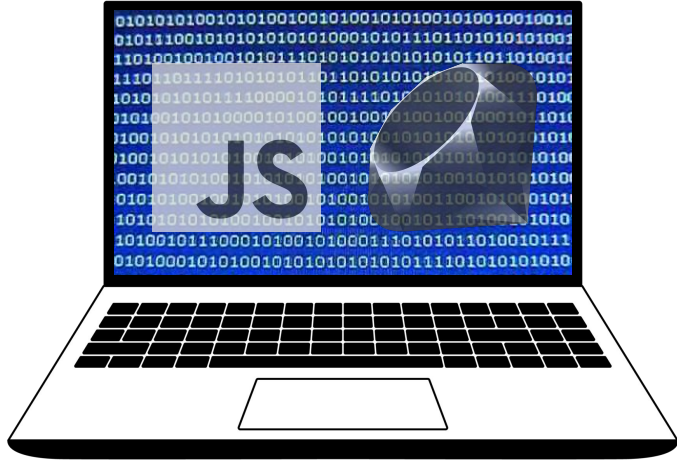
JS

```
0101110101011010101010001001
0101001011010101011010101010
10010111010101101010100010
0101010010110101011010101010
1010010111010101101010101000
1001010111010101101010101000
1001010100101010101010101010
1001010100101010101010101010
0010101001010101010101010101
0101001010101001001001001010
1010101000101001001001001010
0100010100101010101010101010
1001010100101101010101101010
1010100100011101010110101010
1000100101010010111010101011
010101010100101110101011010
10101000100101001001011010101
0111010101010100101110101011
```



executable binary

single process



"add next two entries",
42, ← integer constant

123,
"add next entry to answer",

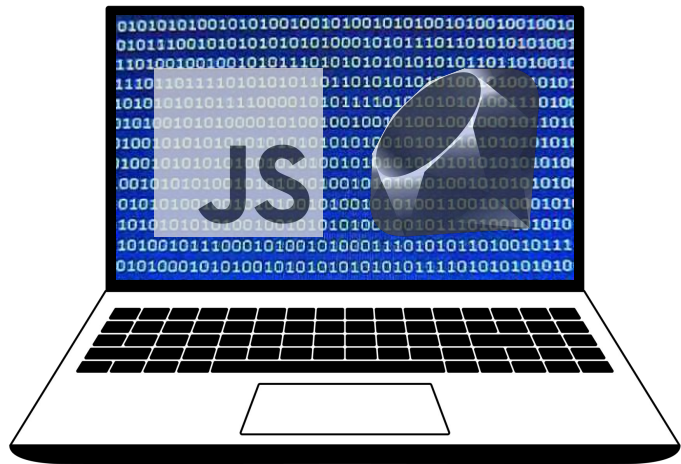
200,
"jump to index in next",

3, ← memory address

...

executable binary

single process



"array length is next entry",
4

010000000000010000
000000000000000000
000000000000000000
000000000000000000
000000000000000000



000000000000000000
000000000000000000
000000000000000000
000000000000000100

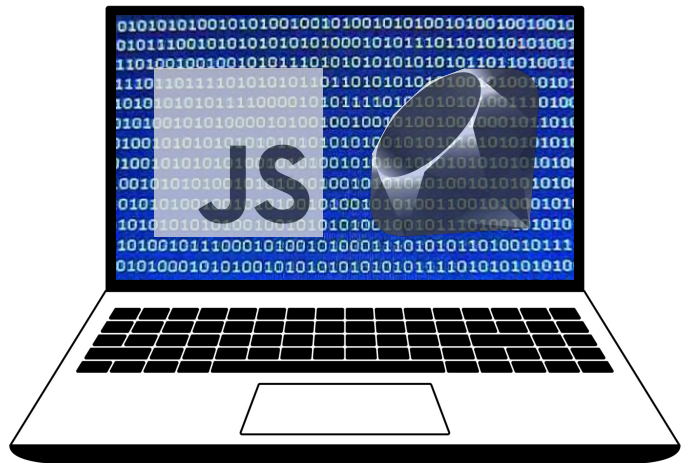


executable binary

single process



"array length is next entry",



010000000000010000
000000000000000000
000000000000000000
000000000000000000
000000000000000000

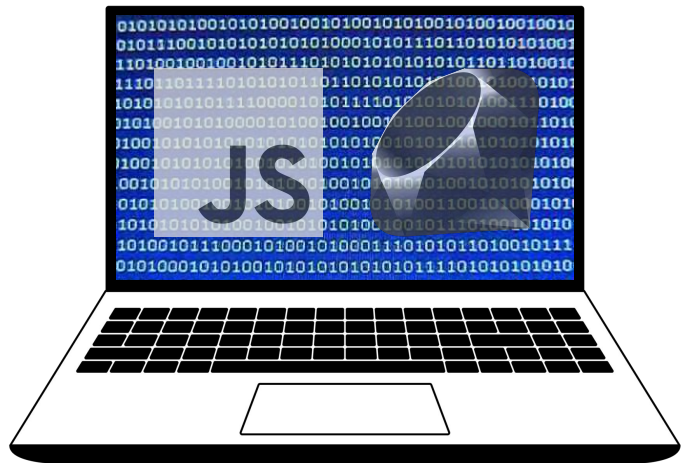
executable binary

single process



"array length is next entry",

4616189618054758400



010000000000010000

000000000000000000

000000000000000000

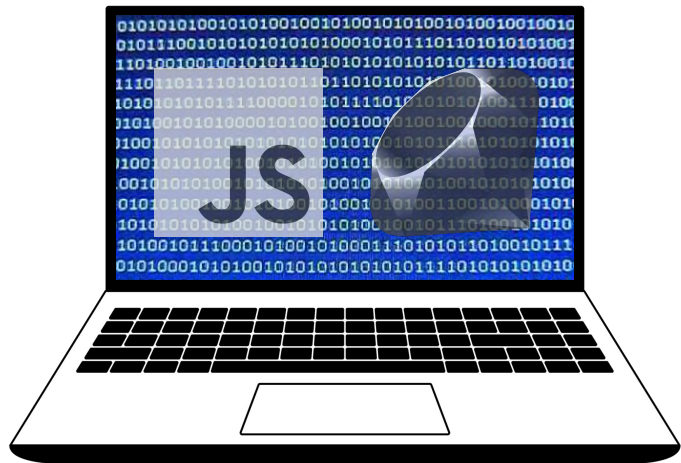
000000000000000000

executable binary

single process



"array length is next entry",
4616189618054758400

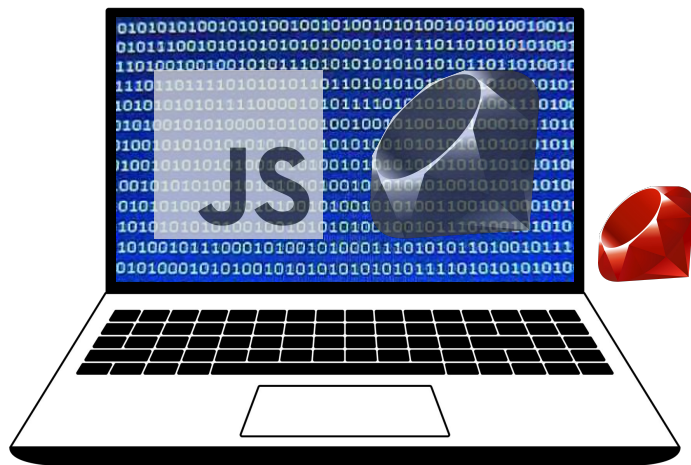


executable binary "jump"

"jump"

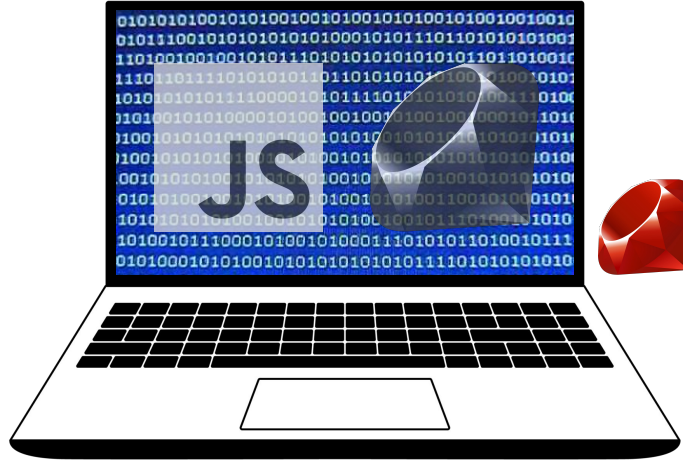
single process

JS

[illegible]

translate before we jump

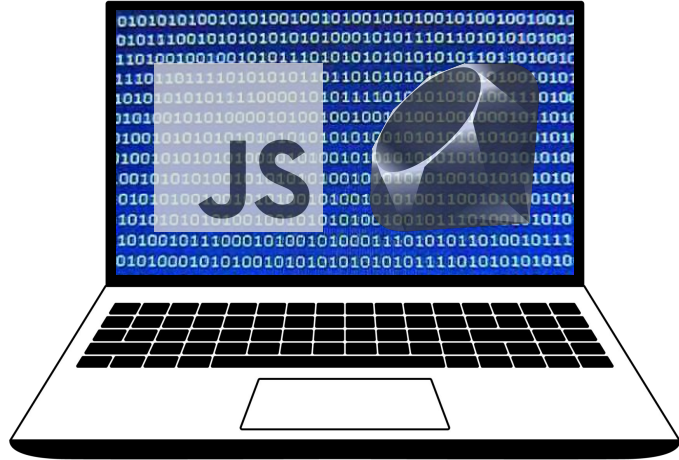
single process



JS

0101110101011010101010001001
0101001011010101011010101010
1001011101010110101010100010
0101010010110101010110101010
101001011100001011101010001101
100100101000100100010010001011
1001010101010101010101010101
1001010101001001010101010101
0010101001001001010100101010
0101001010101001001001001010
101001011000100010001101010101
010100010100101010101011101010
1001010100101101010101101010
1010100100011101010110101010
1000100101010010111010101011
0101010101001011110101011010
1010100010010101001011010101
011101010101010100101110101011

single process

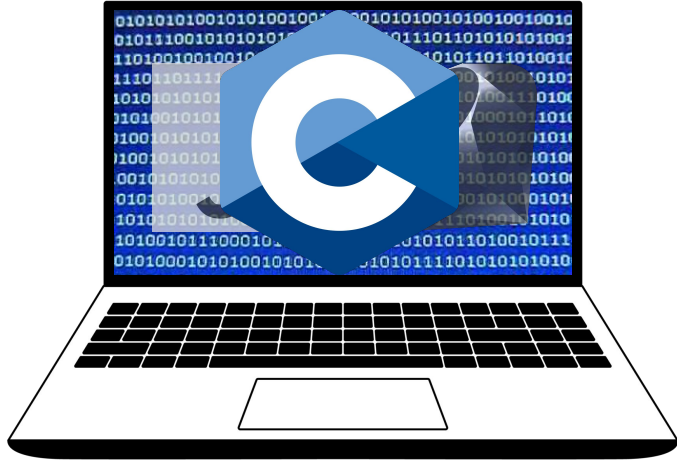


translate before we **jump**

1. Read the **JS-shaped bytes** and make some **Ruby-shaped bytes**
2. **Jump** to the Ruby instructions that will use those **Ruby-shaped bytes**
3. Read the **Ruby-shaped** answer and use it make a **JS-shaped** answer

translate before we **jump**

single process



What we need:

Some way to read arbitrary bytes and write other arbitrary bytes in memory

What we don't want to do:

Painstakingly figure out exactly what bytes need to go where in JS and Ruby

About this documentation

Usage and example

Assertion testing

Asynchronous context tracking

Async hooks

Buffer

C++ addons

C/C++ addons with Node-API

C++ embedder API

Child processes

Cluster

- Functions to convert from C types to Node-API

- `napi_create_int32`
- `napi_create_uint32`
- `napi_create_int64`
- `napi_create_double`
- `napi_create_bigint_int64`
- `napi_create_bigint_uint64`
- `napi_create_bigint_words`
- `napi_create_string_latin1`
- `napi_create_string_utf16`
- `napi_create_string_utf8`

- Functions to convert from Node-API to C types

- `napi_get_array_length`

Table of Contents

- Creating Extension Libraries for Ruby
- Basic Knowledge
- Data Types
- Check Data Type of the VALUE
- Convert VALUE into C Data
- Convert C Data into VALUE
- Manipulating Ruby Data
- String Functions
- Array Functions
- Extending Ruby with C
- Adding New Features to Ruby
- Class and Module Definition
- Method and Singleton Method Definition
- Constant Definition
- Use Ruby Features from C
- Evaluate Ruby Programs in a String
- ID or Symbol
- Invoke Ruby Method from C
- Accessing the Variables and Constants
- Information Sharing Between Ruby and C
- Ruby Constants That Can Be Accessed From
- Global Variables Shared Between C and Rub
- Encapsulate C Data into a Ruby Object
- C struct to Ruby object
- Ruby object to C struct
- Example - Creating the dbm Extension
- Make the Directory
- Design the Library
- Write the C Code
- Prepare extconf.rb
- Prepare Depend (Optional)
- Generate Makefile
- Run make
- Debug
- Building the Extension

Convert VALUE into C Data

The data for type T_NIL, T_FALSE, T_TRUE are nil, false, true respectively. They are singletons for the data type. The equivalent C constants are: Qnil, Qfalse, Qtrue. Note that Qfalse is false in C also (i.e. 0), but not Qnil.

The T_FIXNUM data is a 31bit or 63bit length fixed integer. This size depends on the size of long: if long is 32bit then T_FIXNUM is 31bit, if long is 64bit then T_FIXNUM is 63bit. T_FIXNUM can be converted to a C integer by using the FIX2INT() macro or FIX2LONG(). Though you have to check that the data is really FIXNUM before using them, they are faster. FIX2LONG() never raises exceptions, but FIX2INT() raises **RangeError** if the result is bigger or smaller than the size of int. There are also NUM2INT() and NUM2LONG() which converts any Ruby numbers into C integers. These macros include a type check, so an exception will be raised if the conversion failed. NUM2DBL() can be used to retrieve the double float value in the same way.

You can use the macros StringValue() and StringValuePtr() to get a char* from a VALUE. StringValue(var) replaces var's value with the result of "var.to_str()". StringValuePtr(var) does the same replacement and returns the char* representation of var. These macros will skip the replacement if var is already a String. Notice that the macros take only the lvalue as their argument, to change the value of var in place.

You can also use the macro named StringValueCStr(). This is just like StringValuePtr(), but always adds a NUL character at the end of the result. If the result contains a NUL character, this macro causes the **ArgumentError** exception. StringValuePtr() doesn't guarantee the existence of a NUL at the end of the result, and the result may contain NUL.

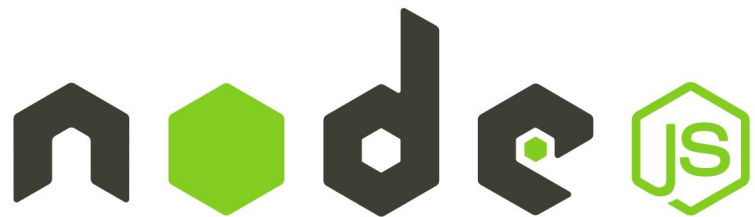
Other data types have corresponding C structures, e.g. struct RArray for T_ARRAY etc. The VALUE of the type which has the corresponding structure can be cast to retrieve the pointer to the struct. The casting macro will be of the form RXXXX for each data type; for instance, RARRAY(obj). See "ruby.h". However, we do not recommend to access RXXXX data directly because these data structures are complex. Use corresponding rb_xxx() functions to access the internal struct. For example, to access an entry of array, use rb_ary_entry(ary, offset) and rb_ary_store(ary, offset, obj).

There are some accessing macros for structure members, for example 'RSTRING_LEN(str)' to get the size of the Ruby String object. The allocated region can be accessed by 'RSTRING_PTR(str)'.

Notice: Do not change the value of the structure directly, unless you are responsible for the result. This ends up being the cause of interesting bugs.

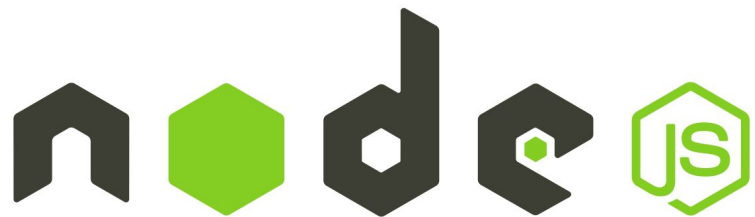
Convert C Data into VALUE

To convert C data to Ruby values:



same process





same process





implement **dot_product**

```
def dot_product(a, b)
  # implementation goes here
end
```



call **dot_product** from NodeJS

```
import { dot_product } from "../ruby_linear_algebra"

console.log("Answer: ", dot_product(a, b))
```



implement bindings between Ruby and Node

```
#include <node_api.h>
```

```
#include <ruby.h>
```

```
static napi_value dot_product(  
    napi_env env,  
    napi_callback_info info  
) {
```

```
    VALUE ruby_array1 = rb_ary_new();
```

```
    VALUE ruby_array2 = rb_ary_new();
```

```
    ... // convert Node arrays to Ruby arrays here
```



implement bindings between Ruby and Node

```
VALUE ruby_answer = rb_funcall(rb_cObject,  
    rb_intern("dot_product"), // name of Ruby fn  
    2, // number of arguments  
    ruby_array1, ruby_array2 // args to fn  
); // Ruby integer  
// Ruby integer ⇒ C integer  
int c_answer = (int)NUM2INT(ruby_answer);  
napi_value js_answer; // C integer ⇒ JS number  
napi_create_int32(env, c_answer, &js_answer);  
return js_answer;  
}
```



implement bindings between Ruby and Node

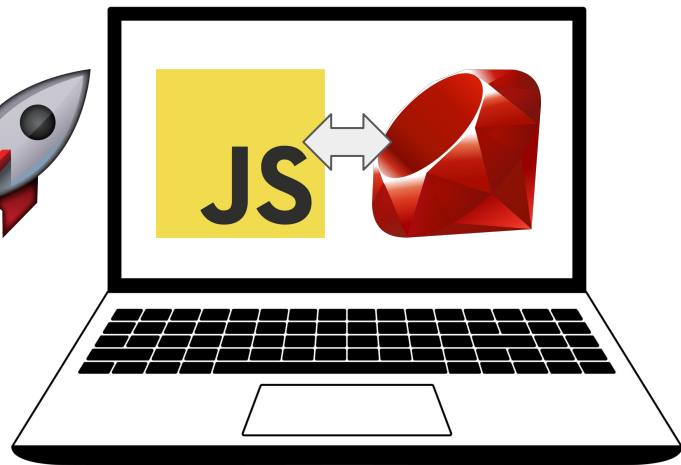
```
static napi_value dot_product(...) {  
    // convert Node arrays to Ruby arrays  
    // * create 2 empty Ruby arrays  
    // * iterate over each Node array  
    // * convert each element: Node number  $\Rightarrow$  C int  
    // * convert each C integer  $\Rightarrow$  Ruby integer  
    // * push Ruby integer onto Ruby array  
    // pass the Ruby arrays to Ruby's `dot_product`  
    // convert its returned Ruby int  $\Rightarrow$  C int  
    // convert that C int  $\Rightarrow$  Node number, return it.  
}
```



implement **dot_product**

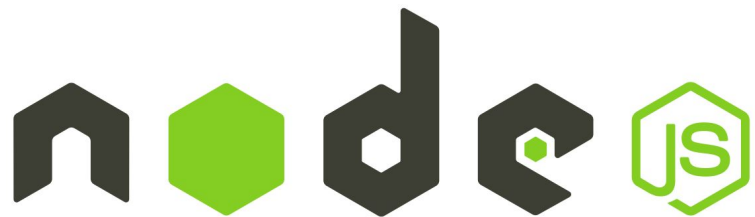
```
def dot_product(a, b)
  # implementation goes here
end
```

single process



call **dot_product** from NodeJS

```
import { dot_product } from "../ruby_linear_algebra"
static napi_value dot_product(...)
console.log("Answer: ", dot_product(a, b))
```

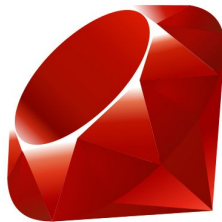


same process



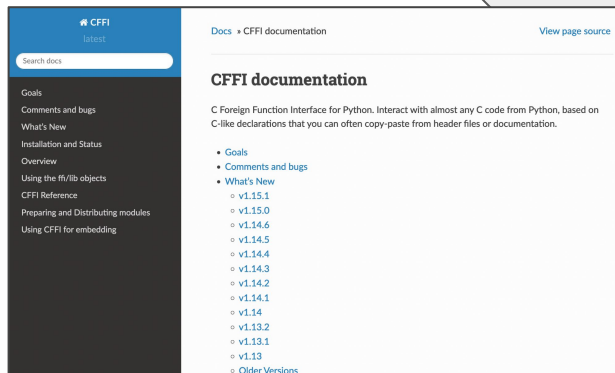


python



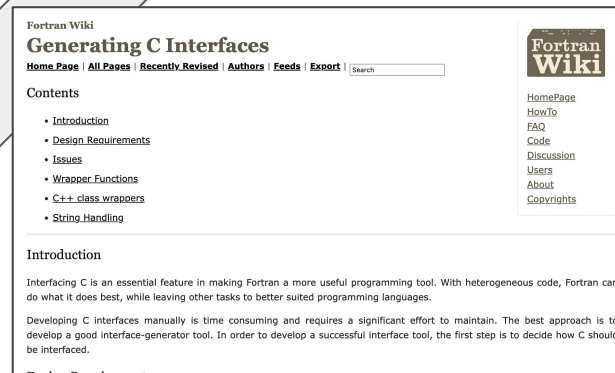
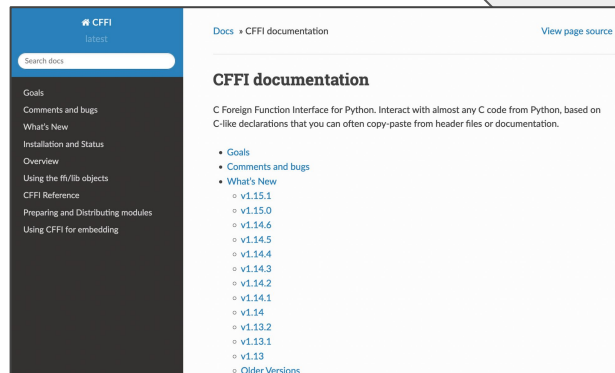
Ruby

same process





same process





1957



CFFI

latest

Search docs

Goals

Comments and bugs

What's New

Installation and Status

Overview

Using the ffi/lib objects

CFFI Reference

Preparing and Distributing modules

Using CFFI for embedding

Docs

CFFI documentation

View page source

CFFI documentation

C Foreign Function Interface for Python. Interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation.

Goals

Comments and bugs

What's New

◦ v1.15.1

◦ v1.15.0

◦ v1.14.6

◦ v1.14.5

◦ v1.14.4

◦ v1.14.3

◦ v1.14.2

◦ v1.14.1

◦ v1.14

◦ v1.13.2

◦ v1.13.1

◦ v1.13

◦ Older Versions



Fortran Wiki

Generating C Interfaces

Home Page | All Pages | Recently Revised | Authors | Feeds | Export | Search

Contents

• Introduction

• Design Requirements

• Issues

• Wrapper Functions

• C++ class wrappers

• String Handling

Introduction

Interfacing C is an essential feature in making Fortran a more useful programming tool. With heterogeneous code, Fortran can do what it does best, while leaving other tasks to better suited programming languages.

Developing C interfaces manually is time consuming and requires a significant effort to maintain. The best approach is to develop a good interface-generator tool. In order to develop a successful interface tool, the first step is to decide how C should be interfaced.

Fortran Wiki

HomePage

HowTo

FAQ

Code

Discussion

Users

About

Copyrights



`import java.foo.bar;`

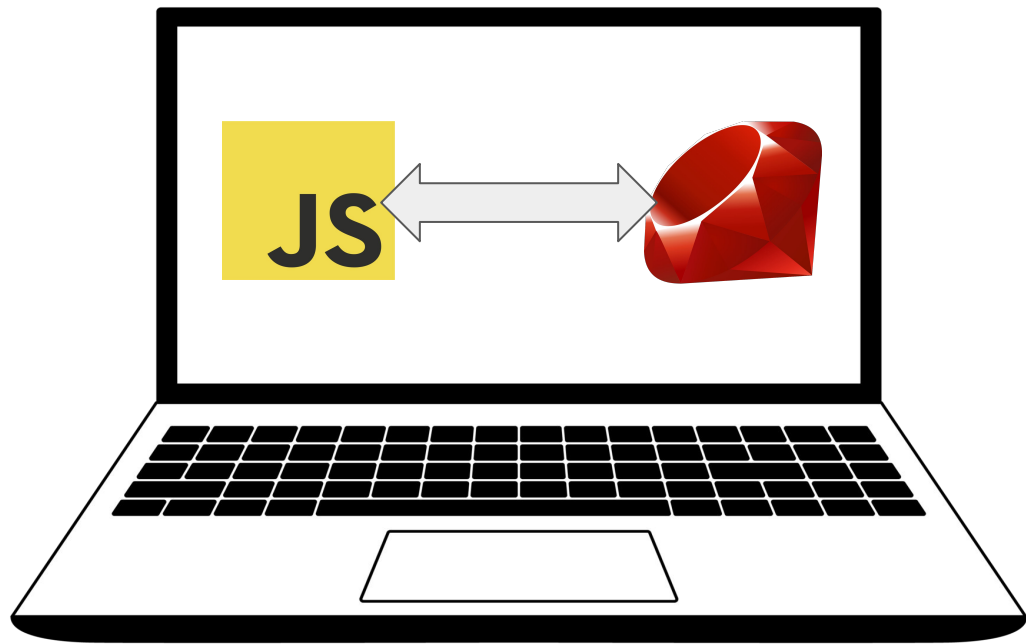


HTTP

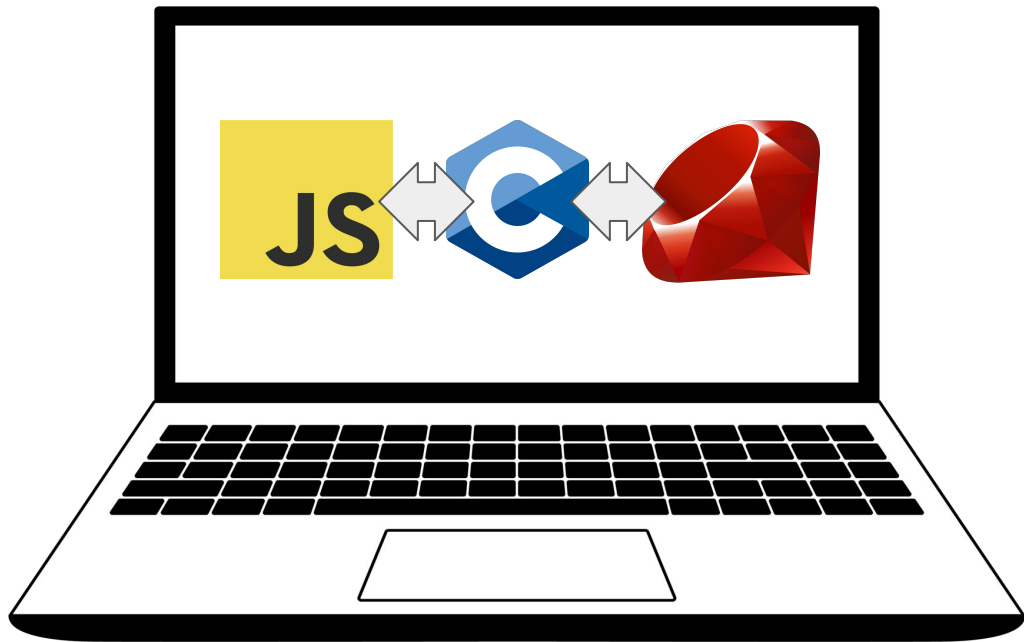


SORCERY?!

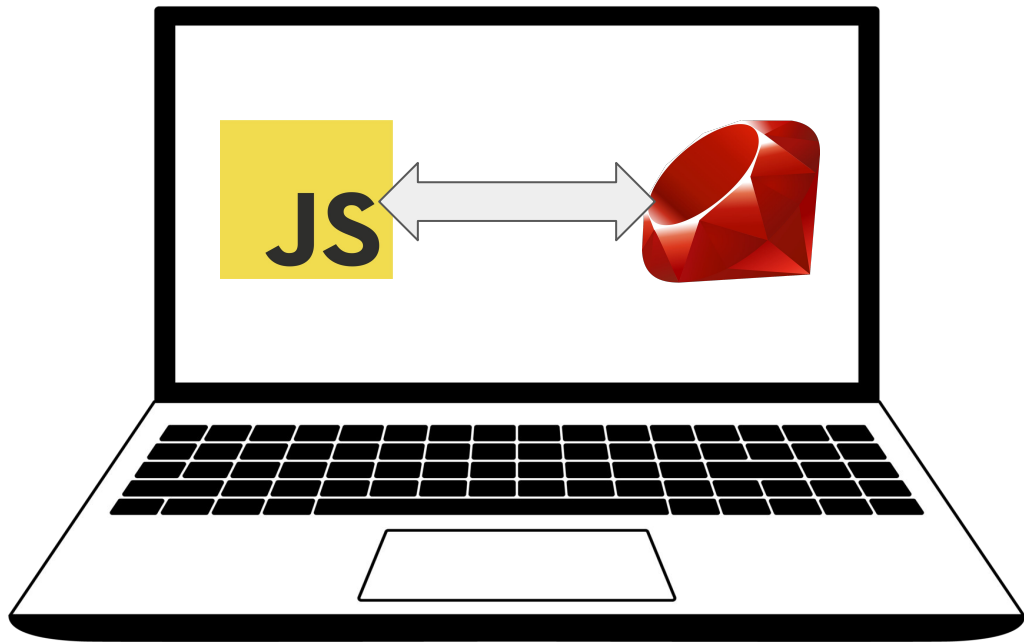
Fortran



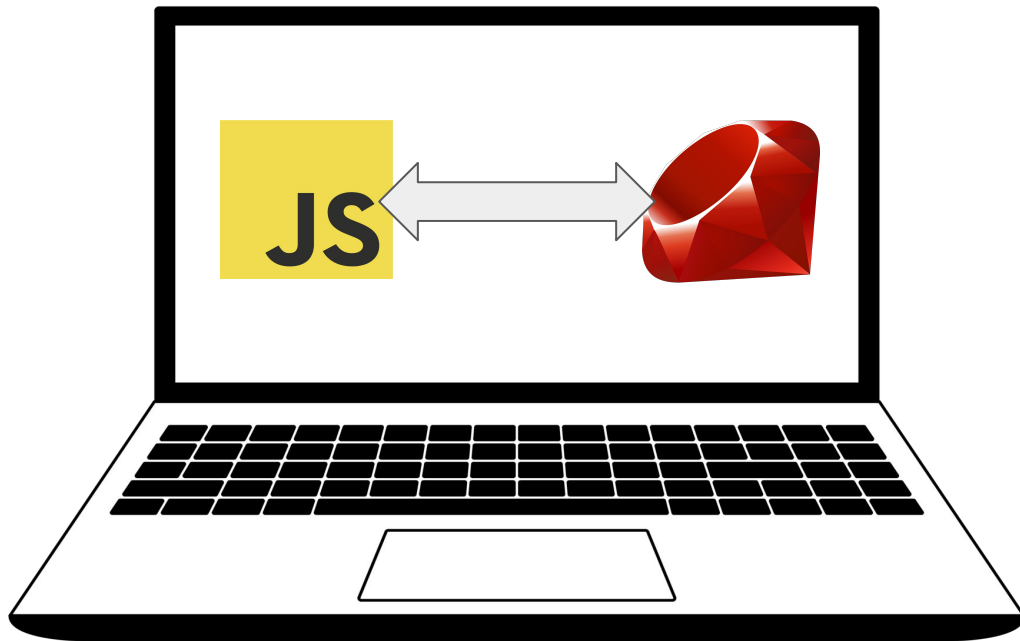
```
int c_answer = (int)NUM2INT(ruby_answer);  
napi_create_int32(env, c_answer, &js_answer);
```



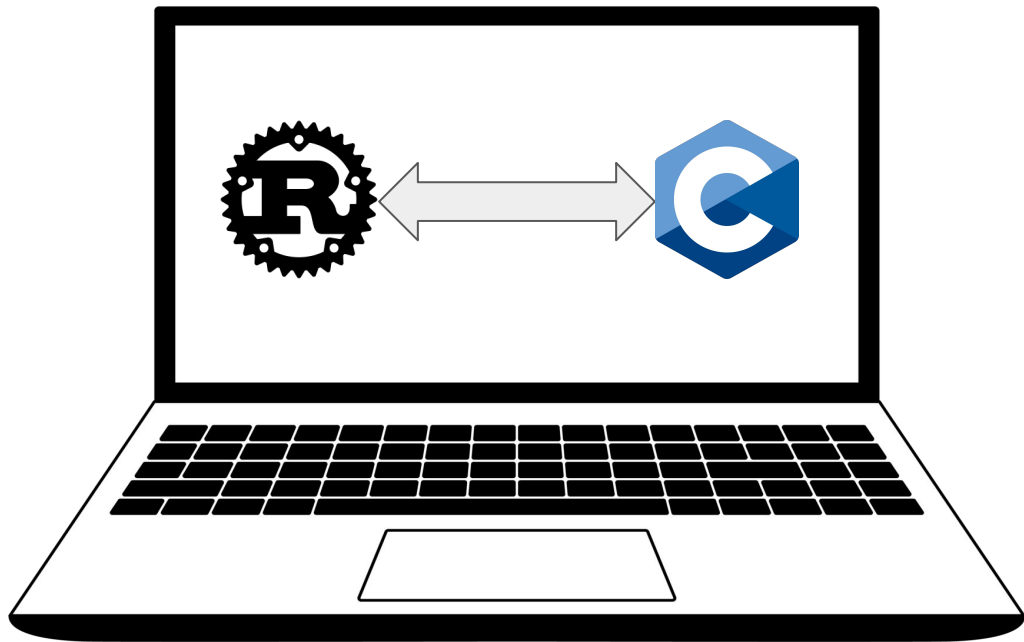
```
napi_value js_answer = NUM2JS(ruby_answer);
```



```
napi_value js_answer = ruby_answer;
```



```
let rust_integer = c_integer;  
let rust_string = c_string;
```





```
import java.foo.bar;
```

[LEARN](#)[INSTALL](#)[PLAY](#)[docs](#)[Getting Started](#)[Scala 3](#) ▾

COLLECTIONS

CONVERSIONS BETWEEN JAVA AND SCALA COLLECTIONS



```
import java.foo.bar;
```



nicer **ergonomics** than writing C

much less **explosion-prone**

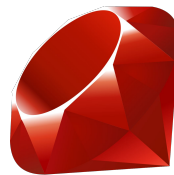
JS $\Rightarrow$ C data structures $\Rightarrow$ **Ruby**

JS $\Rightarrow$ **JSON** $\Rightarrow$ C $\Rightarrow$ **JSON** $\Rightarrow$ **Ruby** less C code



```
import java.foo.bar;
```

TYPE-CHECKED



NOT TYPE-CHECKED

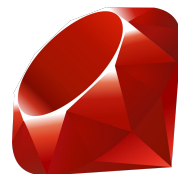
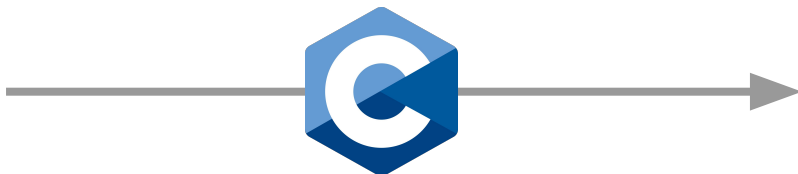
TypeScript



Sorbet

C bindings can't leverage type definitions

```
static napi_value dot_product(  
    napi_env env,  
    napi_callback_info info  
)
```



NOT TYPE-CHECKED

TypeScript



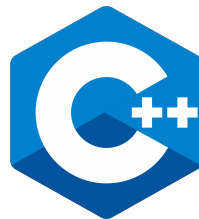


```
import java.foo.bar;
```

TYPE-CHECKED



```
rust-bindgen
```



generate Rust type definitions & bindings
from C or C++ type definitions

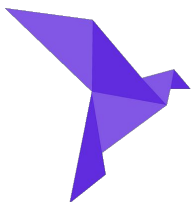
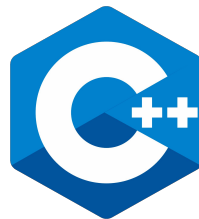


```
import java.foo.bar;
```

TYPE-CHECKED



```
rust-bindgen
```



roc-lang.org

“fast, friendly, functional language”

nice to call from other languages

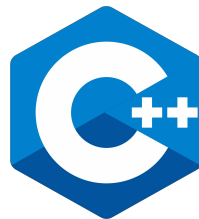


```
import java.foo.bar;
```

TYPE-CHECKED

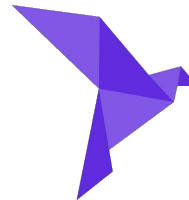


```
rust-bindgen
```



```
roc glue roc-to-rust.roc
```

use your Roc code's types
to generate Rust types & bindings



roc-to-rust.roc

```
makeGlue : List Types -> Result (List File) Str
```

```
Types.walkShapes types fileHeader \buf, type, id ->
```

```
  when type is
```

```
    Function rocFn ->
```

```
      if rocFn.isToplevel then
```

```
        buf
```

```
      else
```

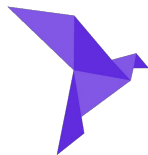
```
        generateFunction buf types rocFn
```

```
    Struct { name, fields } ->
```

```
      generateStruct buf types id name fields Public
```

roc-to-rust.roc

```
makeGlue : List Types -> Result (List File) Str
```



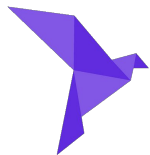
```
dotProduct = \a, b ->  
    ... # return an integer
```

```
roc glue roc-to-rust.roc
```



```
// generated by roc glue roc-to-rust.roc
```

```
fn dot_product(a: &[i64], b: &[i64]) -> i64 {  
    ... // calls Roc dotProduct function  
}
```

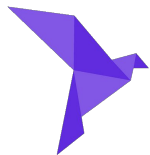


```
dotProduct = \a, b ->  
    ... # return an integer
```

```
roc glue roc-to-node.roc makeGlue : List Types ->
```

JS

```
static napi_value dot_product(  
    napi_env env,  
    napi_callback_info info  
) {  
    ... // do conversions, call Roc dotProduct
```



```
dotProduct = \a, b ->  
    ... # return an integer
```

```
roc glue roc-to-node.roc makeGlue : List Types ->
```



```
static napi_value dot_product
```



```
export function dot_product(  
    a: number[], b: number[]): number;
```



```
import java.foo.bar;
```

TYPE-CHECKED



```
roc glue roc-to-node.roc
```

JS

```
static napi_value dot_product
```

TS

```
export function dot_product(  
  a: number[], b: number[]): number;
```

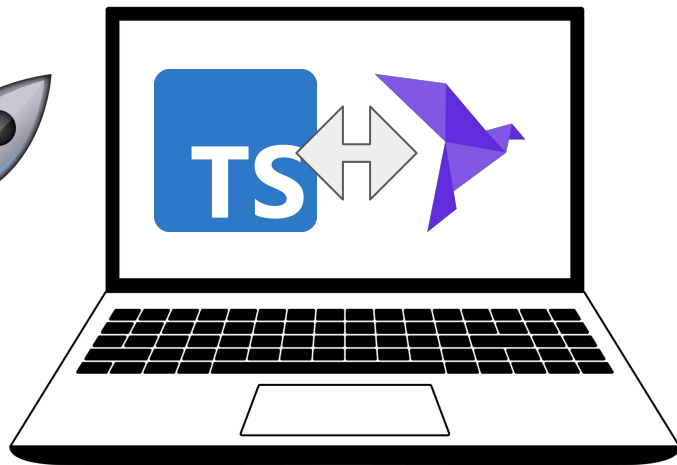


```
import java.foo.bar;
```

TYPE-CHECKED

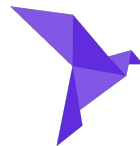


```
roc glue roc-to-node.roc
```



```
import { dotProduct } from "./roc"
```

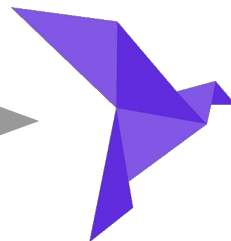
TYPE-CHECKED



vendr.com/careers



vendr

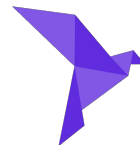


```
roc glue roc-to-node.roc
```



```
import { dotProduct } from "../roc"
```

TYPE-CHECKED





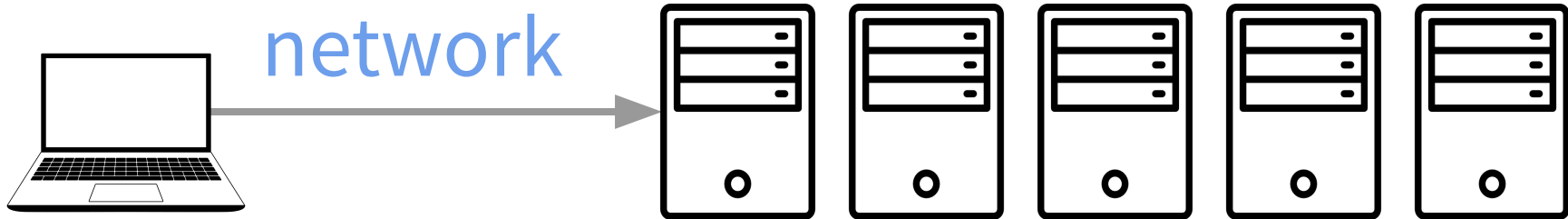
machines are stateful
network errors happen
deployment timing



state across processes
processes can go down



some runtimes are stateful



Language Server Protocol



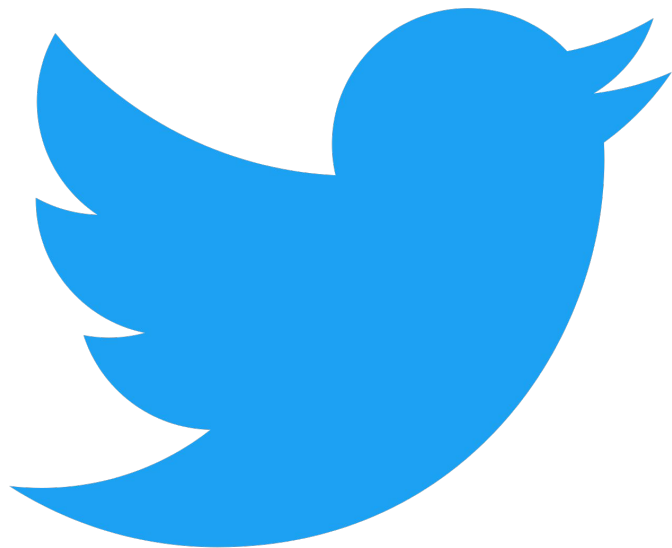
“This task would be a lot easier if I could use another language”

1. I want a **library** that's in another language
2. I want more **performance** than I can get here
3. I want more **ergonomics** at the expense of speed
4. I want to incrementally **transition** languages

Calling Functions Across Languages



@rtfeldman



I host a podcast!



software-unscripted.com