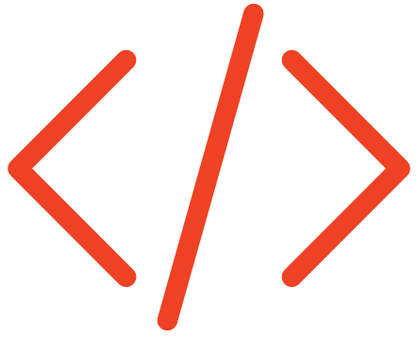
How to write Ruby extensions with Crystal via gem

Shcherbinina Anna @gaar4ica

http://www.devconf.ru

I like Ruby



But, sometimes,
I need my code to work
faster

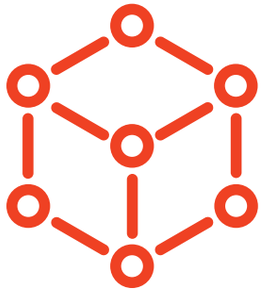
Case of possible bottle neck

```
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

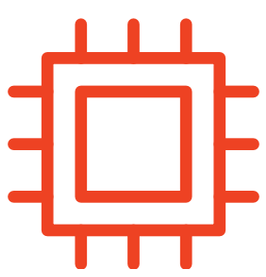
C

Ruby has C bindings

C is hard to learn and understand



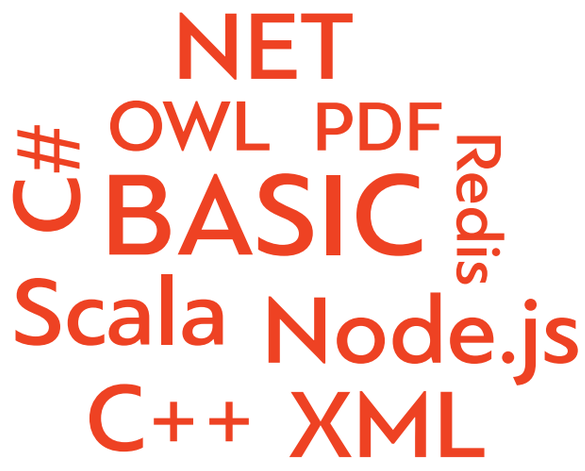
Static types



Allocating memory

And tons of other complications for Ruby developer

I’m ruby developer,
I don’t want to write my code in C



I’m ruby developer,
I want to write my code in Ruby.
But to be honest…

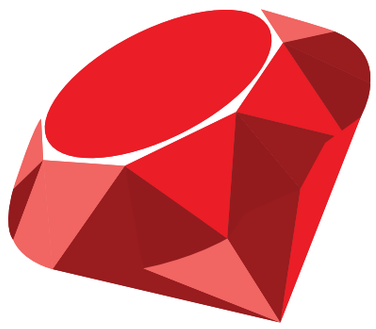
Also, I like Crystal

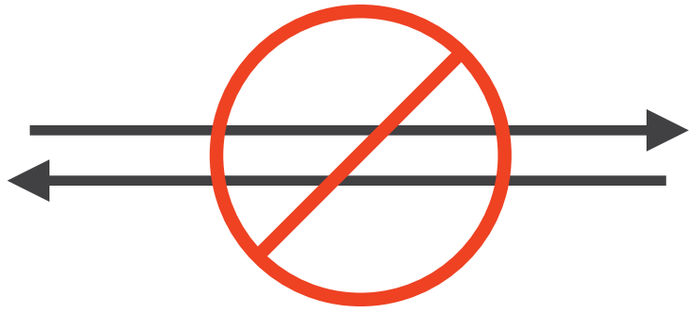

THINKING
CAP
Whoa.
HICKS

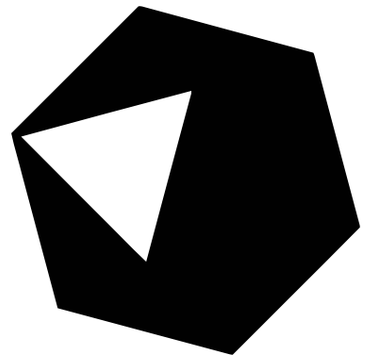
Google Russia

Search Google or type URL

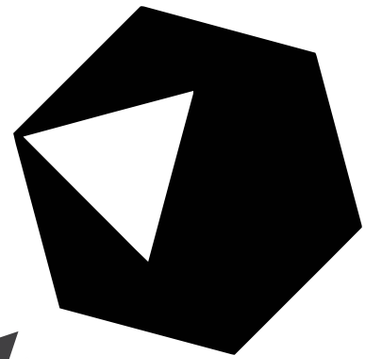
No way

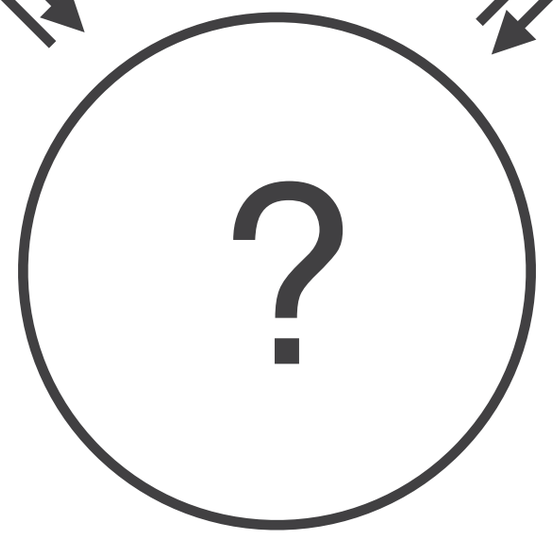




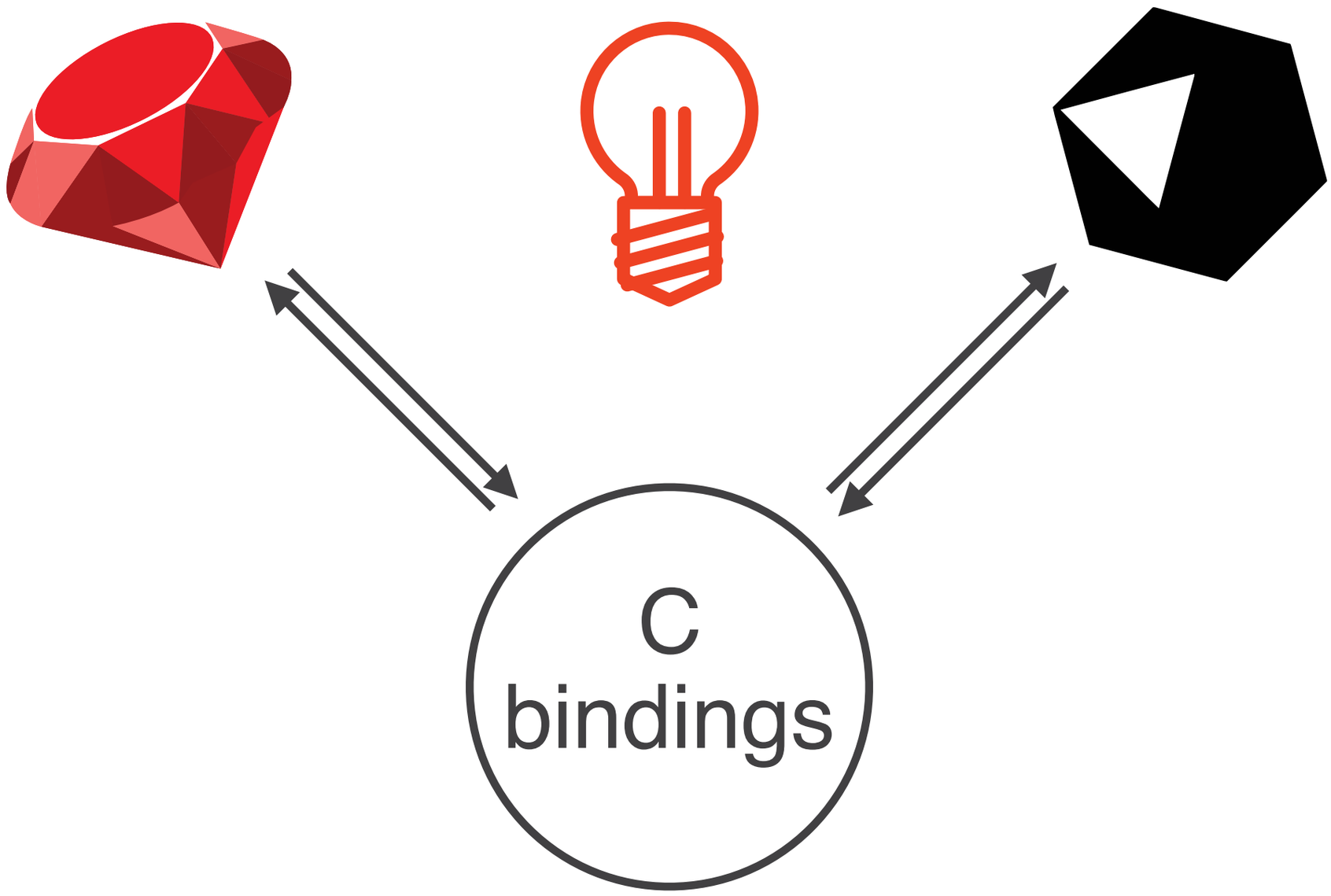


No way





C bindings



C layer

Link Crystal classes
and methods to Ruby,
so Ruby knows about
them

Translate
Ruby-types
to Crystal-types

Back to case of possible bottle neck

```
def fibonacci(n)
     n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

Less work for C

Passing primitive data structures:

- integer
- string
- boolean
- float / decimal
- etc.

No allocation memory for complex data types

Don’t do a lot of work for casting variables passed as attributes

Ruby C API

Entry point

Then every C extension has to implement a function named Init_xxxxx (xxxxx being your extension's name).

It is being executed during the "require" of your extension.

```
void Init_xxxxx() {
  // Code executed during "require"
}
```

Ruby C API

Types

Ruby C API defines a bunch of handful C constants defining standard Ruby objects:

C	Ruby
Qnil	nil
Qtrue	TRUE
Qfalse	FALSE
rb_cObject	Object
rb_mKernel	Kernel
rb_cString	String

Ruby C API

Declaring modules and classes

```c
// Creating classes

// rb_define_class creates a new class named name
// and inheriting from super.
// The return value is a handle to the new class.

VALUE rb_define_class(const char *name, VALUE super);
```

```c
// Creating modules

// rb_define_module defines a module whose name
// is the string name.
// The return value is a handle to the module.

VALUE rb_define_module(const char *name);
```

Ruby C API

Declaring methods

```
// Creating a method

// rb_define_method defines a new instance method in the class
// or moduleklass.
// The method calls func with argc arguments.
// They differ in how they specify the name - rb_define_method
// uses the constant string name
```

```
void rb_define_method(VALUE klass, const char *name, VALUE
(*func)(ANYARGS), int argc);

// rb_define_protected_method and rb_define_private_method are
similar to rb_define_method,
// except that they define protected and private methods,
respectively.

void rb_define_protected_method(VALUE klass, const char *name,
VALUE (*func)(ANYARGS), int argc);
void rb_define_private_method(VALUE klass, const char *name,
VALUE (*func)(ANYARGS), int argc);
```

Ruby C API

Ruby objects → C types

```c
// Convert Numeric to integer.
long rb_num2int(VALUE obj);

// Convert Numeric to unsigned integer.
unsigned long rb_num2uint(VALUE obj);
```

```c
// Convert Numeric to double.
double rb_num2dbl(VALUE);

// Convert Ruby string to a String.
VALUE rb_str_to_str(VALUE object);
char* rb_string_value_cstr(volatile VALUE* object_variable);
```

Ruby C API

C types → Ruby objects

```
// Convert an integer to Fixnum or Bignum.
INT2NUM( int );

// convert an unsigned integer to Fixnum or Bignum.
UINT2NUM( unsigned int );

// Convert a double to Float.
rb_float_new( double );

// Convert a character string to String.
rb_str_new2( char* );

// Convert a character string to ID (for Ruby function names,
etc.).
rb_intern( char* );

// Convert a character string to a ruby Symbol object.
ID2SYM( rb_intern(char*) );
```

Ruby C API

Simple example

```ruby
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end
```

```c
// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

Ruby C API

Simple example

```ruby
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end
```

```c
// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

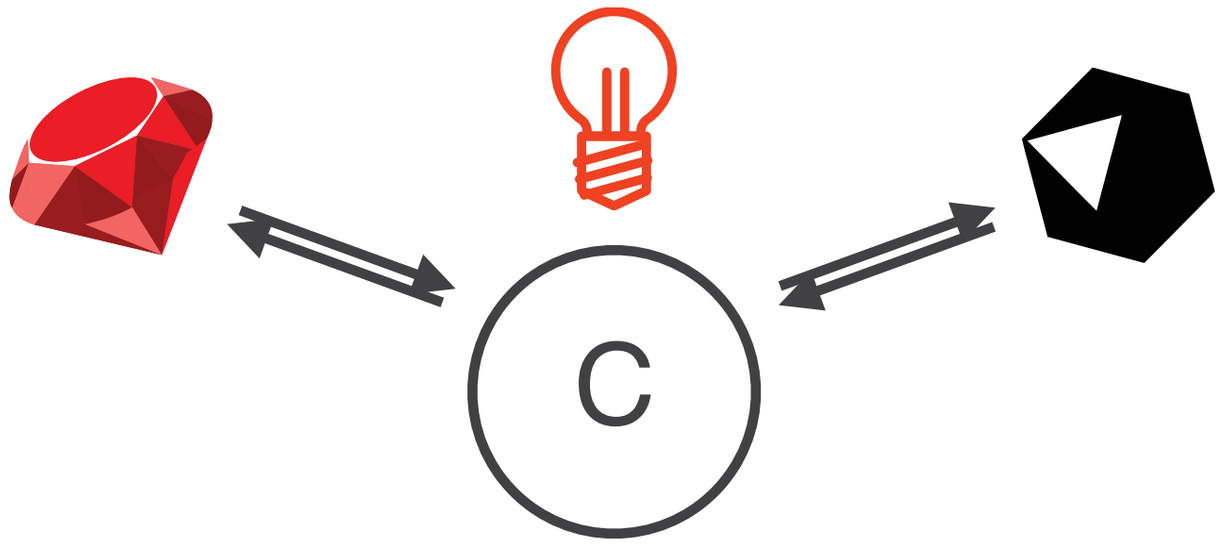
Ruby C API

Simple example

```ruby
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end
```

```c
// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

Back to case of possible bottle neck And solution




```
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

Crystal extension

It’s easy

A lib declaration groups C functions and types that belong to a library.

```
lib CrRuby
  type VALUE = Void*
  $rb_cObject : VALUE
end
```

Crystal extension

It’s easy

Every ruby object is treated as type VALUE in C.

```
lib CrRuby
  type VALUE = Void*
  $rb_cObject : VALUE
end
```

Crystal extension

It’s easy

rb_cObject is a C constant defining standard Ruby Object class.

```
lib CrRuby
  type VALUE = Void*
  $rb_cObject : VALUE
end
```

Crystal extension

It’s easy

A fun declaration inside a lib binds to a C function.

We bind rb_num2int & rb_int2inum, to use it later.

```
lib CrRuby
  type VALUE = Void*

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE
end
```

Crystal extension

It’s easy

We bind rb_define_module, rb_define_class_under & rb_define_method.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super:
VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func:
METHOD_FUNC, argc: Int32)
end
```

Crystal extension

It’s easy

Our new and shiny fibonacci method in Crystal.

```
def fibonacci_cr(n)
  n <= 1 ? n : fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Do you see the difference with implementation in Ruby?

```
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

Crystal extension

It’s easy

Let’s create a wrapper for this method, used for:

- convert inbound Ruby-type parameter to Crystal;
- convert outbound result back to Ruby type.

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

We bind **init** function, the first one to be called.

As you remember, a function named Init_xxxxx is being executed during the “require” of your extension.

```
fun init = Init_common_math
end
```

Crystal extension

It’s easy

Firstly, we start garbage collector.

We need to invoke Crystal's "main" function, the one that initializes all constants and runs the top-level code.

We pass 0 and null to argc and argv.

```
fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)
end
```

Crystal extension

It’s easy

We define module CommonMath and class CrMath.

```
fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  math_module = CrRuby.rb_define_module(“CommonMath")
  cr_math = CrRuby.rb_define_class_under(math_module, "CrMath",
CrRuby.rb_cObject);
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

It’s easy

We define module CommonMath and class CrMath.

Attach method fibonacci to class.

```
fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  math_module = CrRuby.rb_define_module(“CommonMath")
  cr_math = CrRuby.rb_define_class_under(math_module, "CrMath",
CrRuby.rb_cObject);
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have Crystal library with C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end
```

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end

fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  math_module = CrRuby.rb_define_module(“CommonMath”)
  cr_math = CrRuby.rb_define_class_under(math_module, "CrMath", CrRuby.rb_cObject);
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have method fibonacci_cr and wrapper for it.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_module(name: UInt8*) : VALUE
  fun rb_define_class_under(module: VALUE, name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end
```

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

```
fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  math_module = CrRuby.rb_define_module(“CommonMath”)
  cr_math = CrRuby.rb_define_class_under(math_module, "CrMath", CrRuby.rb_cObject);
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have entry point.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

```
fun init = Init_common_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  math_module = CrRuby.rb_define_module(“CommonMath”)
  cr_math = CrRuby.rb_define_class_under(math_module, "CrMath",
CrRuby.rb_cObject);
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

Makefile

```
install: common_math.cr
   crystal common_math.cr --release --link-flags
"-dynamic -bundle -Wl,-undefined,dynamic_lookup"
-o common_math.bundle

clean:
   rm -f common_math.bundle
```

Crystal extension

Using in Ruby

```
$ irb
2.1.6 :001 > require 'common_math'
 => true
2.1.6 :002 > CommonMath::CrMath.new.fibonacci(20)
 => 6765
```

Gem Gem Gem

Crystal extension

Create a gem

```
➜  ruby_ext_in_crystal_math git:(master) ✗ tree  .
.
├── README.md
├── ext
│   └── common_math
│       ├── Makefile
│       ├── common_math.cr
│       └── extconf.rb
├── lib
│   ├── common_math
│   │   ├── rb_math.rb
│   │   └── version.rb
│   └── common_math.rb
└── ruby_ext_in_crystal_math.gemspec

4 directories, 8 files
```

Crystal extension

Create a gem

ruby_ext_in_crystal_math.gemspec

```
Gem::Specification.new do |s|
  s.name          = 'ruby_ext_in_crystal_math'
  s.version       = CommonMath::VERSION
  s.authors       = ['Anna Kazakova (gaar4ica)']

  s.extensions    = %w[ext/common_math/extconf.rb]
end
```

Crystal extension

Create a gem

ext/common_math/extconf.rb

```
require 'mkmf'

abort 'missing crystal' unless find_executable('crystal')

# Dirty patching
def create_makefile(_, _ = nil)
end

create_makefile(nil)
```

Crystal extension

Create a gem

lib/common_math.rb

```
require_relative '../ext/common_math/common_math'
require 'common_math/rb_math'
require 'benchmark'
```

Crystal extension

Create a gem

lib/common_math/rb_math.rb

```
module CommonMath
  class RbMath
    def fibonacci(n)
      n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
    end
  end
end
```

Crystal extension

Create a gem

lib/common_math.rb

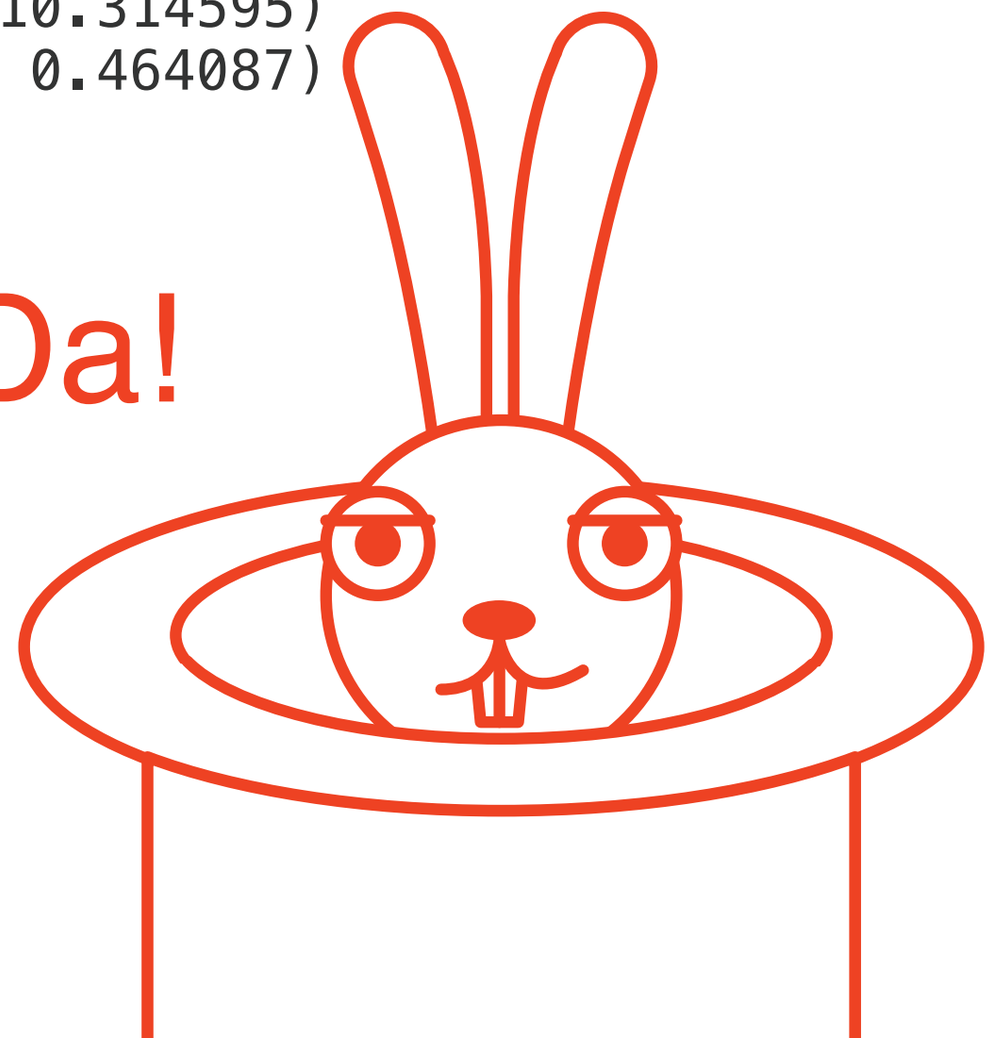
```
module CommonMath
  def self.measure
    iterations = 10_000
    number = 20

    Benchmark.bm do |bm|
      bm.report('rb') do
        iterations.times { RbMath.new.fibonacci(number) }
      end

      bm.report('cr') do
        iterations.times { CrMath.new.fibonacci(number) }
      end
    end

    return
  end
end
```

Crystal extension

Benchmarking

```
$ irb
2.1.6 :001 > require 'common_math'
 => true
2.1.6 :002 > CommonMath.measure

       user     system      total        real
rb 10.270000   0.030000  10.300000 ( 10.314595)
cr  0.460000   0.000000   0.460000 (  0.464087)

2.1.6 :003 > nil
```

Ta-Da!



Thank you

https://twitter.com/gaar4ica

https://github.com/gaar4ica/ruby_ext_in_crystal_math

http://crystal-lang.org/

https://github.com/manastech/crystal

https://github.com/5t111111/ruby_extension_with_crystal

http://blog.jacius.info/ruby-c-extension-cheat-sheet/