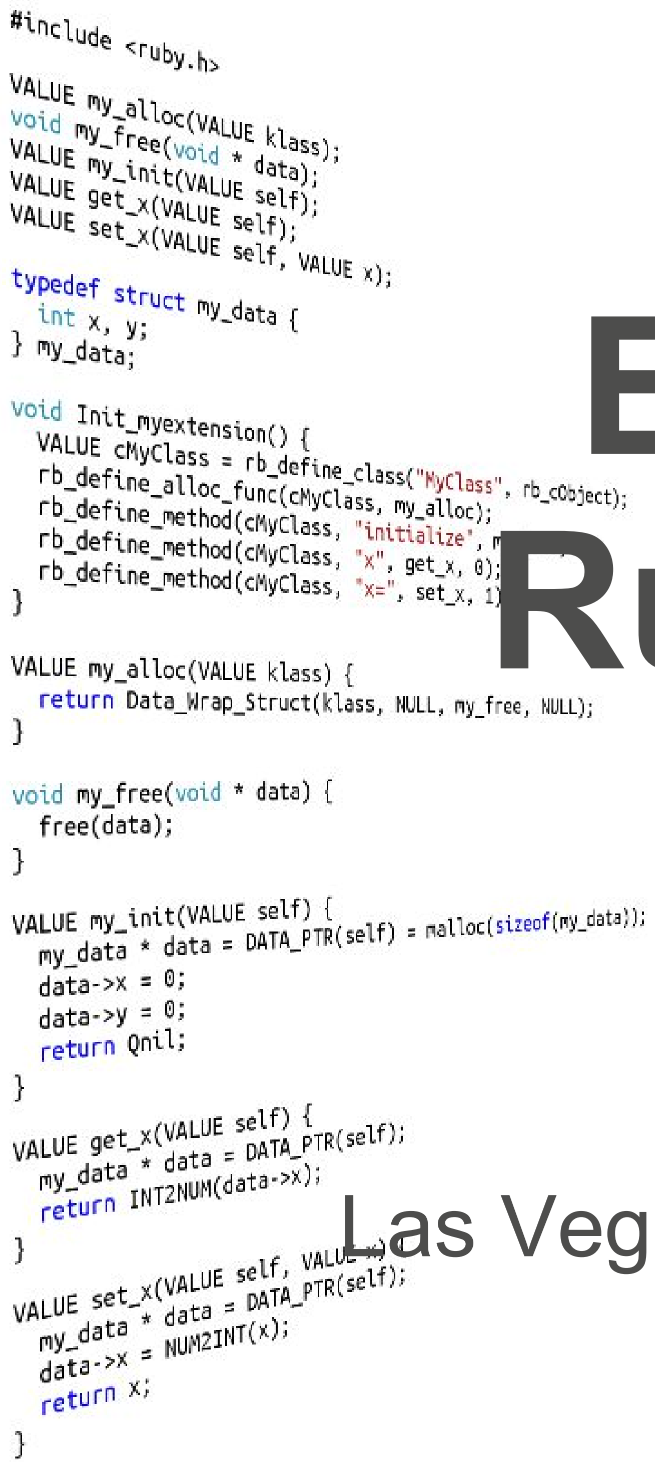


```c
#include <ruby.h>

VALUE my_alloc(VALUE klass);
void my_free(void * data);
VALUE my_init(VALUE self);
VALUE get_x(VALUE self);
VALUE set_x(VALUE self, VALUE x);

typedef struct my_data {
  int x, y;
} my_data;

void Init_myextension() {
  VALUE cMyClass = rb_define_class("MyClass", rb_cObject);
  rb_define_alloc_func(cMyClass, my_alloc);
  rb_define_method(cMyClass, "initialize", m
  rb_define_method(cMyClass, "x", get_x, 0);
  rb_define_method(cMyClass, "x=", set_x, 1)
}

VALUE my_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, my_free, NULL);
}

void my_free(void * data) {
  free(data);
}

VALUE my_init(VALUE self) {
  my_data * data = DATA_PTR(self) = malloc(sizeof(my_data));
  data->x = 0;
  data->y = 0;
  return Qnil;
}

VALUE get_x(VALUE self) {
  my_data * data = DATA_PTR(self);
  return INT2NUM(data->x);
}

VALUE set_x(VALUE self, VALUE
  my_data * data = DATA_PTR(self);
  data->x = NUM2INT(x);
  return x;
}
```

Extending Ruby with C

David Grayson

Las Vegas Ruby Meetup, 2013-10-23

Previously presented 2011-11-16

Why make a C extension?

- To access C libraries from Ruby
- To run CPU-intensive algorithms

Along the way you learn more about <u>Ruby</u>!

What can a C extension do?

- Almost everything that Ruby can!
  - Convert between Ruby and C data
  - Call Ruby methods
  - Define classes, modules, methods, constants
  - Throw or rescue exceptions
  - Access Ruby variables
  - Define blocks and yield values to blocks
  - Be packaged in a gem

What else can a C extension do?

- Lots of things that Ruby can't!
  - Looks like the source of MRI
  - Store a hidden pointer in a Ruby object
  - Add hooks to Ruby interpreter
  - Define read-only global variables
  - Global variables with get and set hooks

Your first C extension

myextension.c:

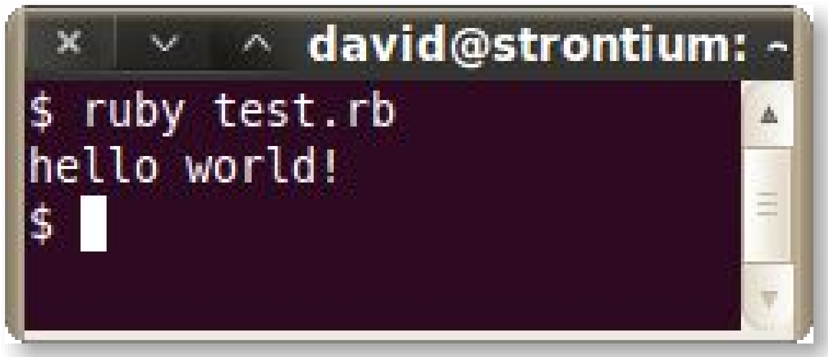
```c
#include <ruby.h>
void Init_myextension()
{
    printf("hello world!\n");
}
```

test.rb:

```ruby
require_relative 'myextension'
```

extconf.rb:

```ruby
require 'mkmf'
$CFLAGS += ' -std=gnu99'
create_makefile 'myextension'
```



```
$ ruby test.rb
hello world!
$ 
```

Run these commands:

```
ruby extconf.rb
make
```

Defining a Ruby class

What we want to do:

```ruby
class MyClass
  def foo(arg1)
  end
end
```

Equiavalent code in C:

```c
#include <ruby.h>

VALUE foo(VALUE self, VALUE arg1)
{
    return Qnil;
}

void Init_myextension()
{
    VALUE cMyClass = rb_define_class("MyClass", rb_cObject);
    rb_define_method(cMyClass, "foo", foo, 1);
}
```

The VALUE type

- Represents any Ruby object
- Is defined in ruby.h to be an unsigned integer the same size as a pointer (void *).

```
        typedef unsigned long VALUE;


                32-bit VALUE space
        MSB ------------------------ LSB
object  oooooooooooooooooooooooooooooo00 : pointer to C struct
fixnum  fffffffffffffffffffffffffffffff1 : 31-bit signed int
symbol  ssssssssssssssssssssssss00001110 : ruby symbol
false   00000000000000000000000000000000
true    00000000000000000000000000000010
nil     00000000000000000000000000000100
undef   00000000000000000000000000000110
```

Source: gc.c in Ruby source code

Getting the TYPE of a VALUE

- The TYPE(obj) macro gets the type of a VALUE:

T_NIL	nil
T_OBJECT	ordinary object
T_CLASS	class
T_MODULE	module
T_FLOAT	floating point number
T_STRING	string
T_REGEXP	regular expression
T_ARRAY	array
T_HASH	associative array
T_STRUCT	(Ruby) structure
T_BIGNUM	multi precision int
T_FIXNUM	Fixnum(31/63bit int)
T_COMPLEX	complex number
T_RATIONAL	rational number
T_FILE	IO
T_TRUE	true
T_FALSE	false
T_DATA	data
T_SYMBOL	symbol

The TYPE is not the same thing as the class!

Reading basic types

```
VALUE foo(VALUE self, VALUE obj)
{
    switch (TYPE(obj)) {
    case T_FIXNUM:;
        int val = NUM2INT(obj);
        printf("Fixnum: %d\n", val);
        break;
    case T_STRING:;
        char * string = StringValuePtr(obj);
        printf("String: %s\n", string);
        break;
    case T_ARRAY:;
        unsigned long length = RARRAY_LEN(obj);
        printf("Array:  %ld\n", length);
        break;
    }
    return Qnil;
}
```

```
MyClass.new.foo 12    #=> Fixnum: 12
MyClass.new.foo "hi"  #=> String: hi
MyClass.new.foo [0,3] #=> Array:  2
```

Creating basic types

```
VALUE foo2(VALUE self)
{
  VALUE string = rb_str_new2("hello");
  VALUE number = INT2NUM(44);
  VALUE array = rb_ary_new3(2, string, number);
  return array;
}
```

```
MyClass.new.foo2 #=> ["hello", 44]
```

Calling Ruby methods

What we want to do:

```
def foo(obj)
  obj + obj
end
```

Equiavalent code in C:

```
VALUE foo(VALUE self, VALUE obj)
{
    VALUE doubled = rb_funcall(obj, rb_intern("+"), 1, obj);
    return doubled;
}
```

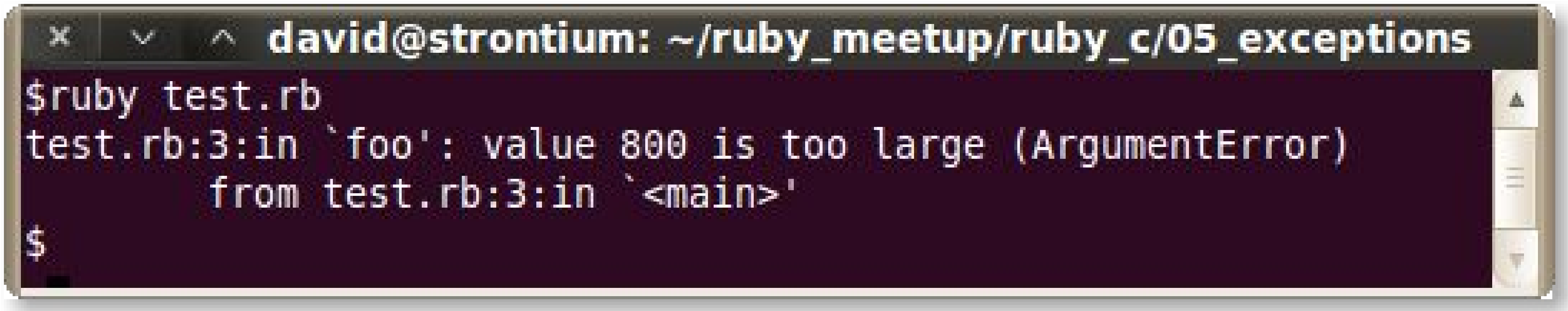
```
MyClass.new.foo "boo"  #=> "booboo"
MyClass.new.foo 44     #=> 88
```

Raising Exceptions

```c
VALUE foo(VALUE self, VALUE num)
{
  int x = NUM2INT(num);
  if (x > 100)
  {
    rb_raise(rb_eArgError, "value %d is too large", x);
    printf("this does not ever run\n");
  }
  return Qnil;
}
```

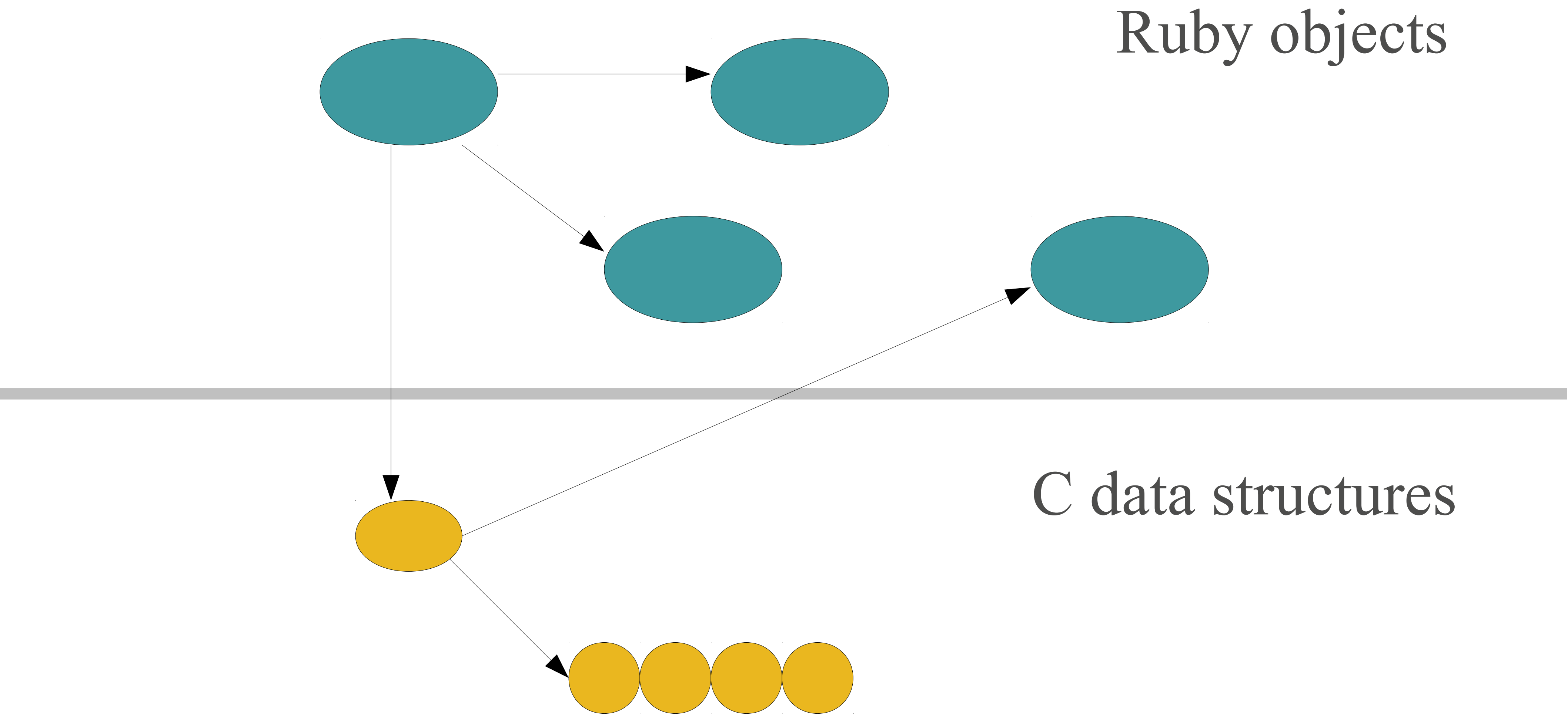
test.rb:

```ruby
require_relative 'myextension'
MyClass.new.foo 800
```



```
$ruby test.rb
test.rb:3:in `foo': value 800 is too large (ArgumentError)
        from test.rb:3:in `<main>'
$
```

Ruby objects, C objects, coexist!



Data_Wrap_Struct

```
typedef void (*RUBY_DATA_FUNC)(void*);

VALUE Data_Wrap_Struct(
  VALUE class,
  RUBY_DATA_FUNC mark,
  RUBY_DATA_FUNC free,
  void * ptr
);
```

You want to call this when
a new object is created...

How objects are made

- Object.new:
  1) Calls class's allocator method to allocate memory.
  2) Calls object's #initialize method.

Data Pointer Strategy

- A strategy for C extensions:
  - In allocator, use Data_Wrap_Struct, with NULL pointer.
  - In #initialize, set the value of the pointer.
  - Free the pointer when the ruby object is garbage collected.
  - If needed, provide a #close method to free the pointer early.

Data Pointer Example

```c
#include <ruby.h>

VALUE my_alloc(VALUE klass);
void my_free(void * data);
VALUE my_init(VALUE self);
VALUE get_x(VALUE self);
VALUE set_x(VALUE self, VALUE x);

typedef struct my_data {
  int x, y;
} my_data;

void Init_myextension() {
  VALUE cMyClass = rb_define_class("MyClass", rb_cObject);
  rb_define_alloc_func(cMyClass, my_alloc);
  rb_define_method(cMyClass, "initialize", my_init, 0);
  rb_define_method(cMyClass, "x", get_x, 0);
  rb_define_method(cMyClass, "x=", set_x, 1);
}

VALUE my_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, my_free, NULL);
}

VALUE my_init(VALUE self) {
  my_data * data = DATA_PTR(self) = malloc(sizeof(my_data));
  data->x = 0;
  data->y = 0;
  return Qnil;
}

void my_free(void * data) {
  free(data);
}
```

```c
VALUE get_x(VALUE self) {
  my_data * data = DATA_PTR(self);
  return INT2NUM(data->x);
}

VALUE set_x(VALUE self, VALUE x) {
  my_data * data = DATA_PTR(self);
  data->x = NUM2INT(x);
  return x;
}
```

Supporting Ruby implementations

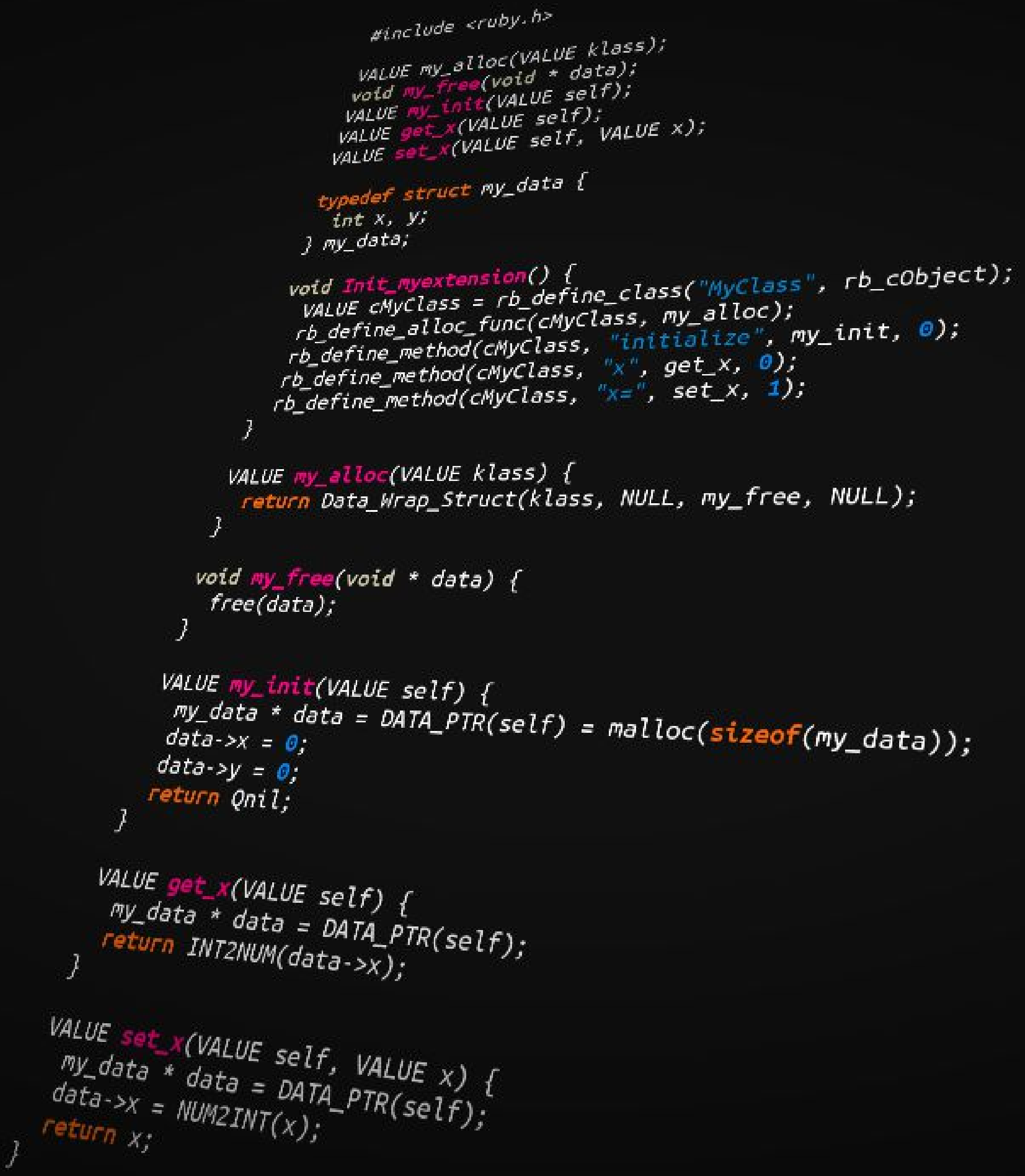
- Matz Ruby Interpreter: yes
- Rubinius: yes
- JRuby: it used to support C extensions

Alternatives

- CLI written in C
- dl, fiddle, ffi

Resources

- Excellent PDF by Dave Thomas:
  - http://media.pragprog.com/titles/ruby3/ext_ruby.pdf
- Official document in Ruby source code:
  - https://github.com/ruby/ruby/blob/trunk/README.EXT
- EscapeUtils, a simple gem with a C extension:
  - https://github.com/brianmario/escape_utils



```c
#include <ruby.h>

VALUE my_alloc(VALUE klass);
void my_free(void * data);
VALUE my_init(VALUE self);
VALUE get_x(VALUE self);
VALUE set_x(VALUE self, VALUE x);

typedef struct my_data {
  int x, y;
} my_data;

void Init_myextension() {
  VALUE cMyClass = rb_define_class("MyClass", rb_cObject);
  rb_define_alloc_func(cMyClass, my_alloc);
  rb_define_method(cMyClass, "initialize", my_init, 0);
  rb_define_method(cMyClass, "x", get_x, 0);
  rb_define_method(cMyClass, "x=", set_x, 1);
}

VALUE my_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, my_free, NULL);
}

void my_free(void * data) {
  free(data);
}

VALUE my_init(VALUE self) {
  my_data * data = DATA_PTR(self) = malloc(sizeof(my_data));
  data->x = 0;
  data->y = 0;
  return Qnil;
}

VALUE get_x(VALUE self) {
  my_data * data = DATA_PTR(self);
  return INT2NUM(data->x);
}

VALUE set_x(VALUE self, VALUE x) {
  my_data * data = DATA_PTR(self);
  data->x = NUM2INT(x);
  return x;
}
```