...

RuDy

Perform like a [redacted] Star

Rock

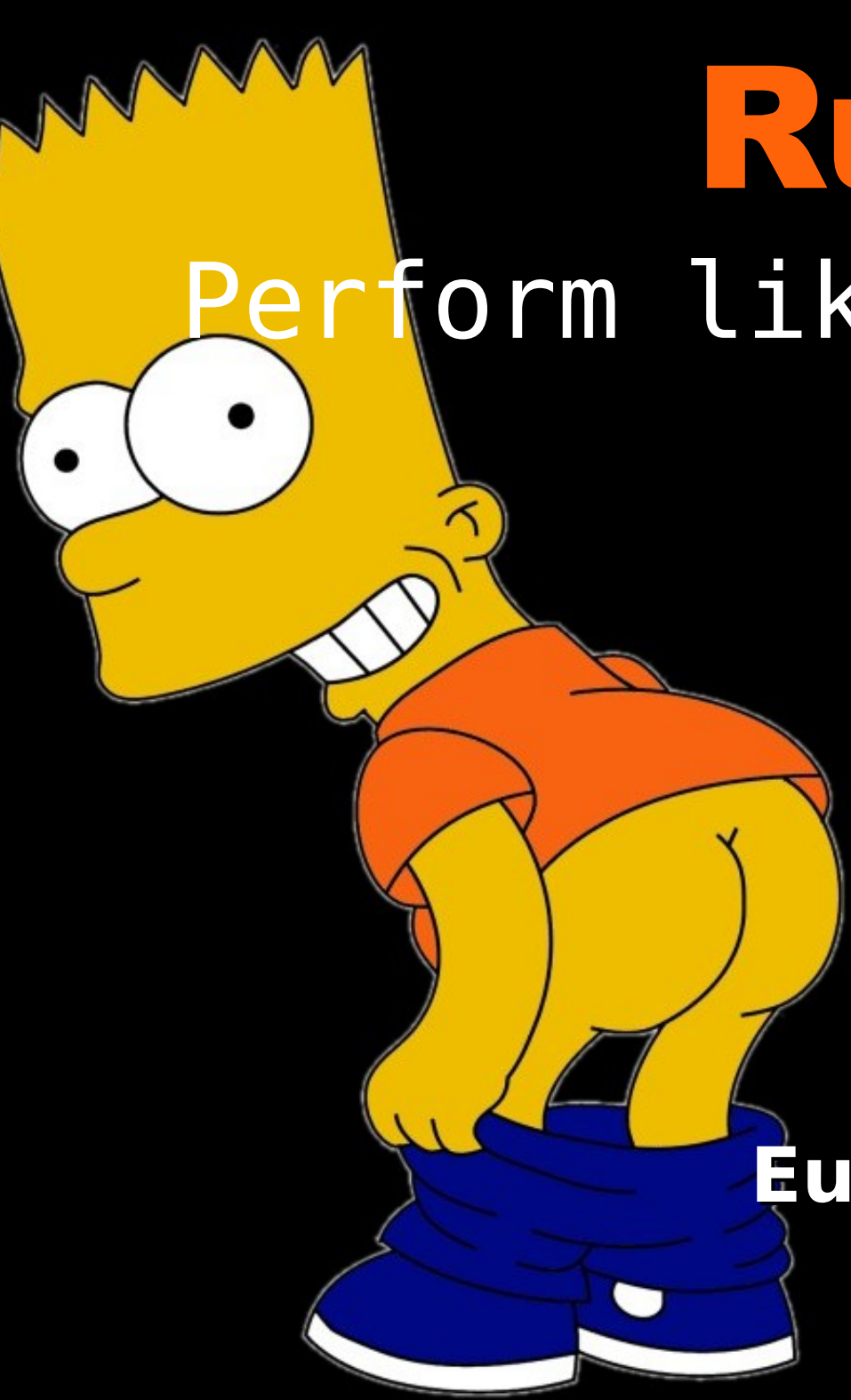
Tomasz Stachewicz

Euruko 2009 :: Barcelona

RuDy

Perform like a [redacted] Star

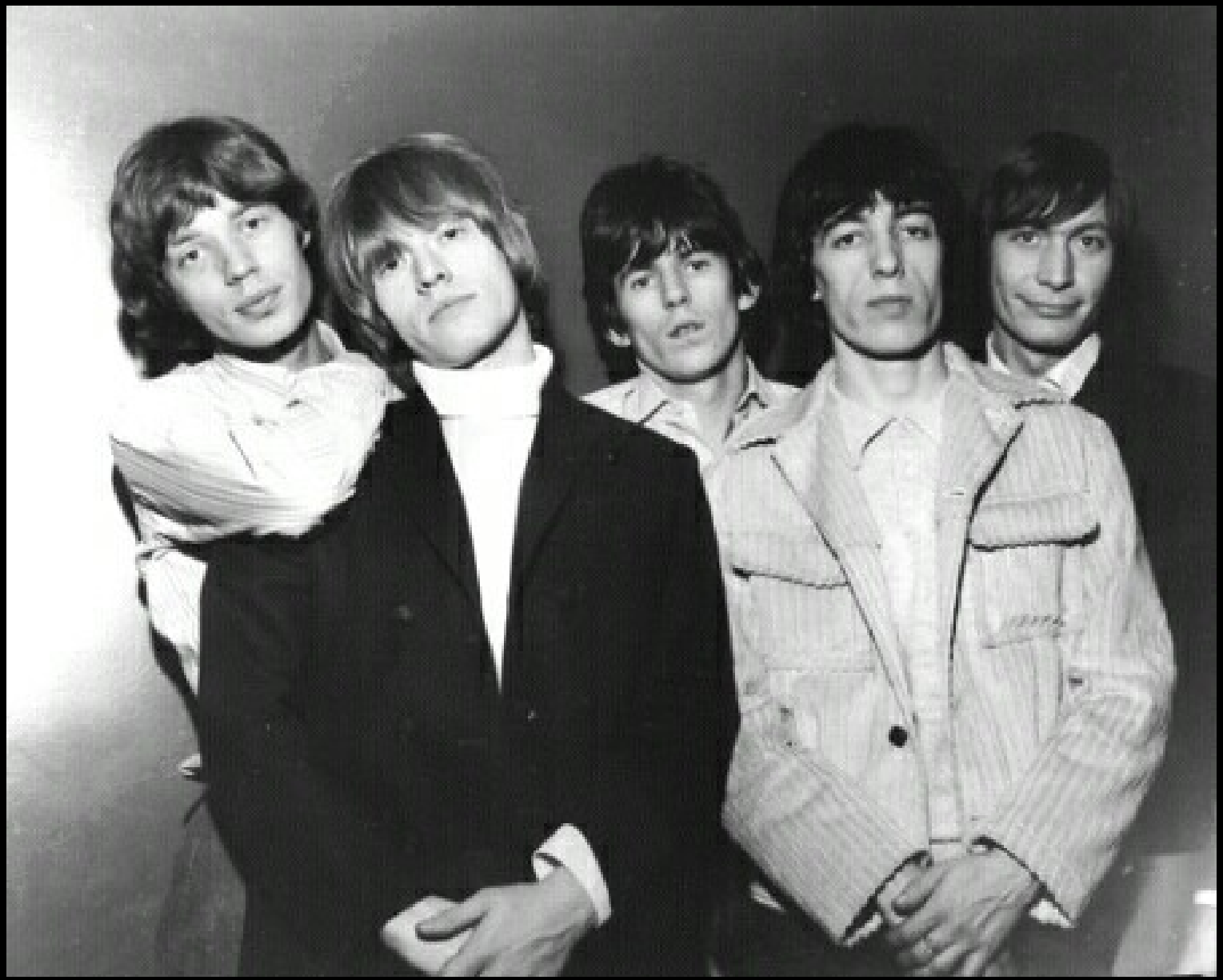
Rock



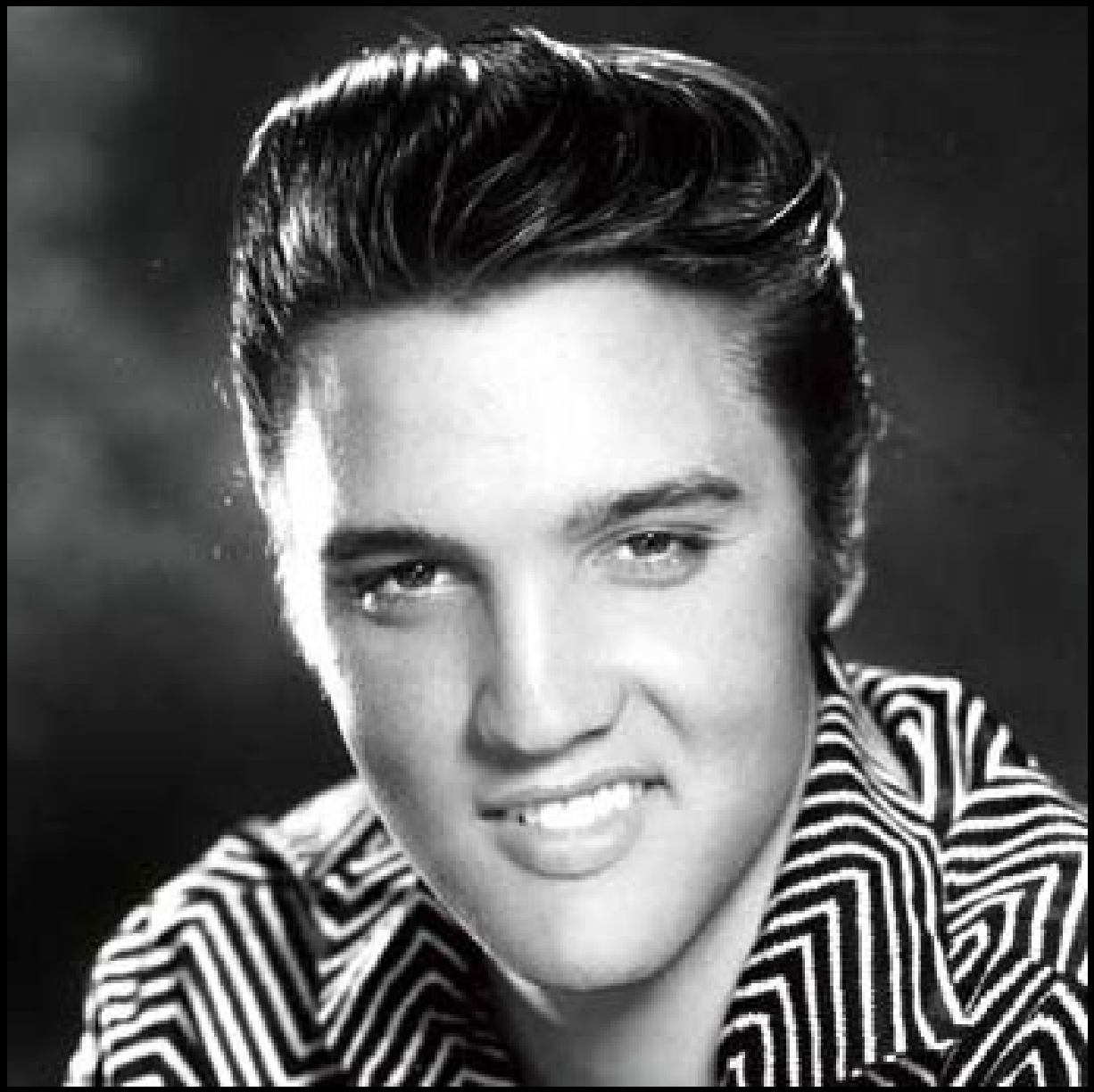
Tomasz Stachewicz

Euruko 2009 :: Barcelona

Is your language a Rock Star?



(Lisp)



(COBOL)

Metal!

But we want performance.

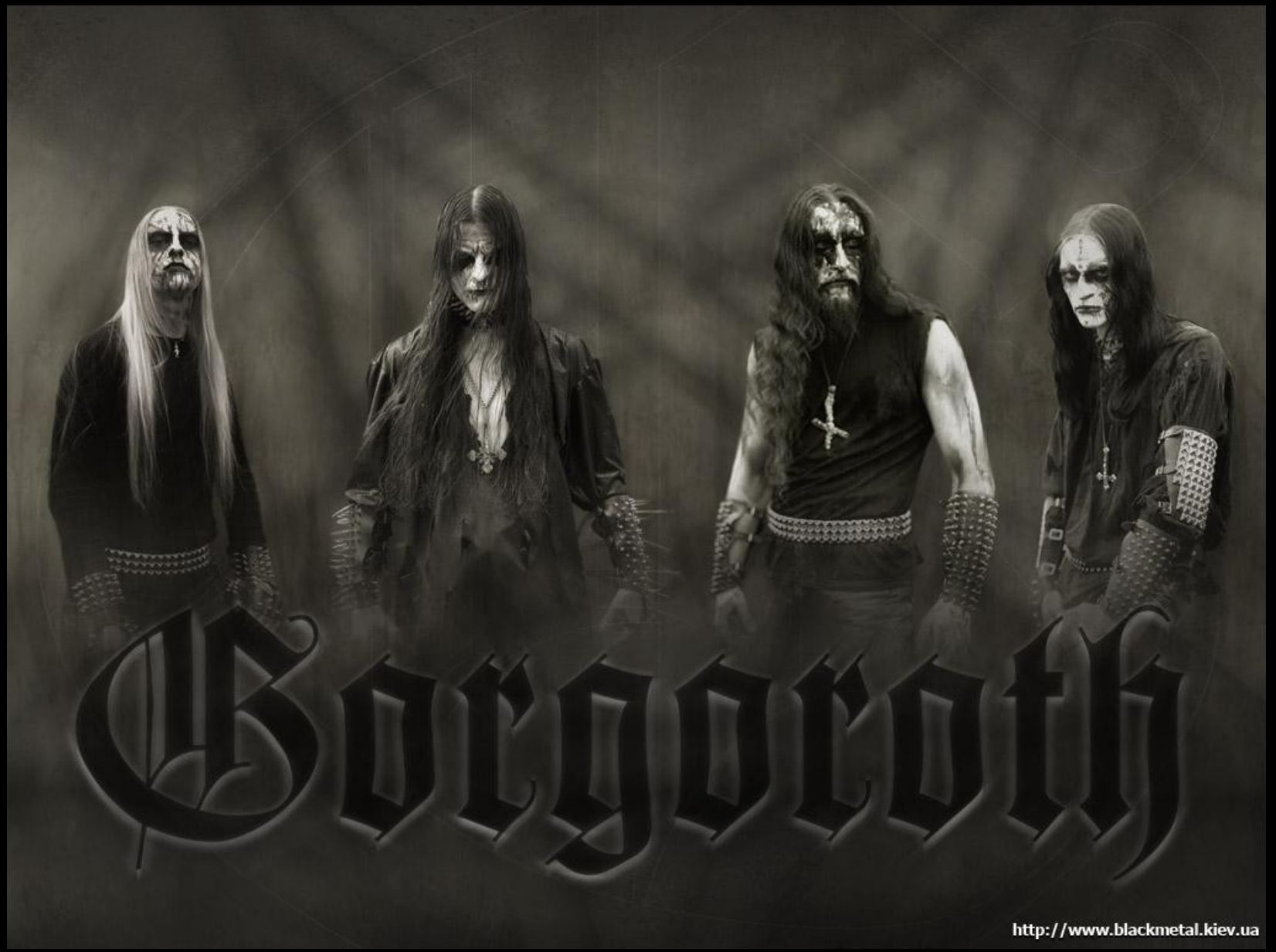
Even Matz wants performance.

We want Metal, or at least close to.

What Metal Star your language is?

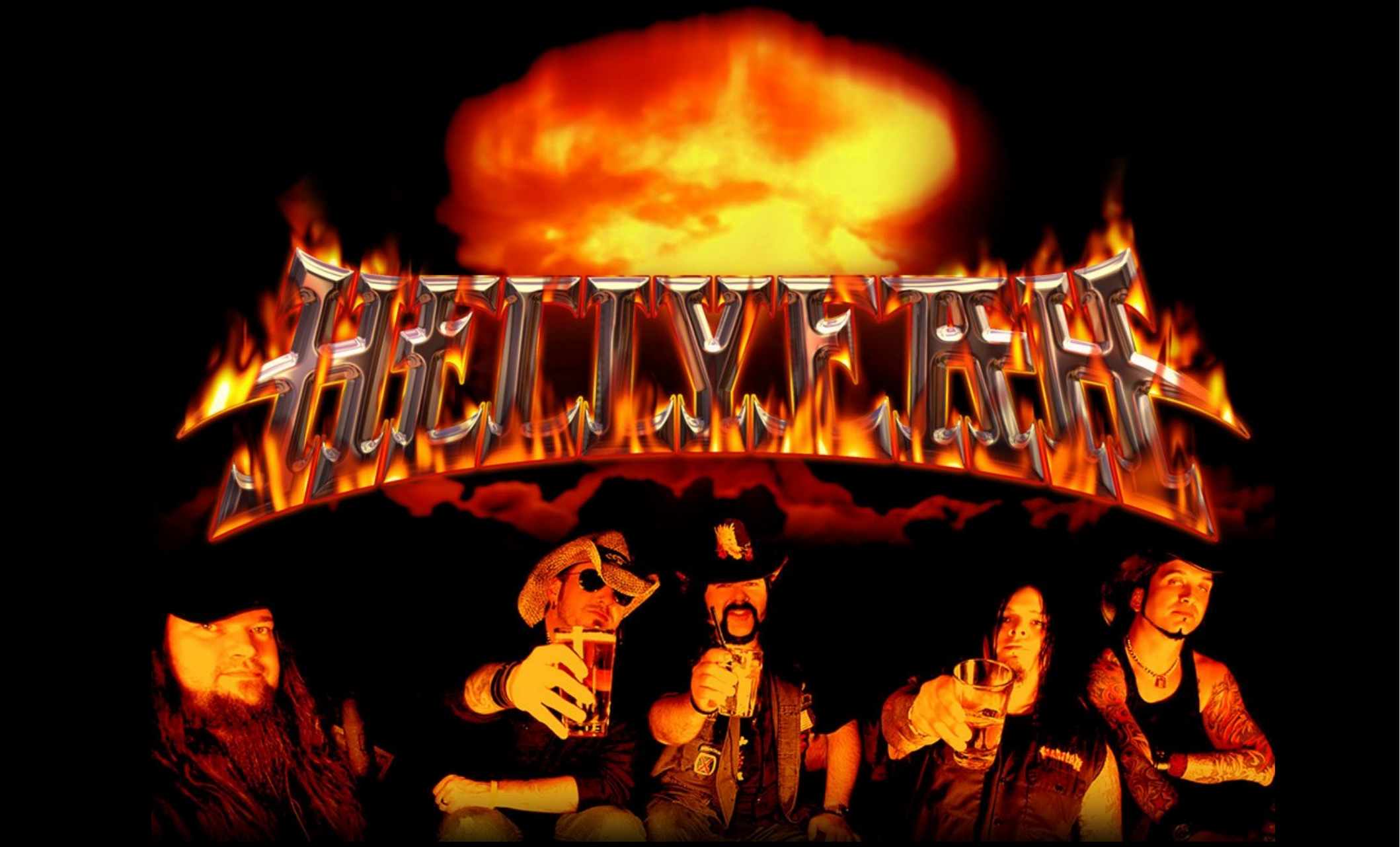
C

C++



Assembler

Pascal



D

There are many flavors of metal

Let's have an overview

The main problem with writing extensions for Ruby?

The main problem with writing extensions for Ruby?

C

Switching from Ruby to C hurts

Ruby:

- Object-Oriented
- GC
- namespacing
- dynamic arrays
- hashes, strings
- unit testing
- exceptions
- poor performance

Switching from Ruby to C hurts

Ruby:

- Object-Oriented
- GC
- namespacing
- dynamic arrays
- hashes, strings
- unit testing
- exceptions
- poor performance

C:

- structs at most
- malloc, free
- only global or scope
- static only
- ???
- ???
- segmentation fault
- good performance

Why not use C++?

And why don't you?

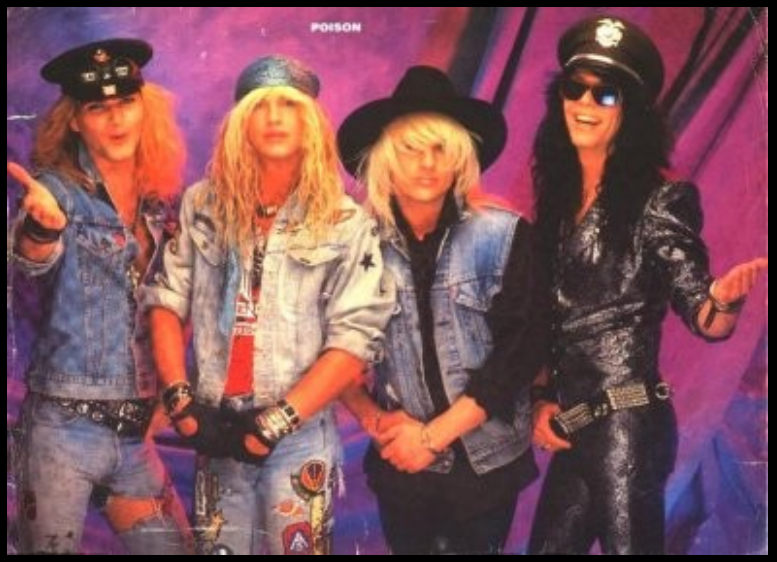
Popularity of writing extentions in C++ speaks for itself.

C++: A Classic Quote

The good news is that in 1995 we will have a good operating system and programming language; the bad news is that they will be Unix and C++.

Richard P. Gabriel, 1991

(http://naggum.no/worse-is-better.html)



Yeah, I don't like C++.

But what are our options when we don't want crudeness of C and awkwardness of C++?

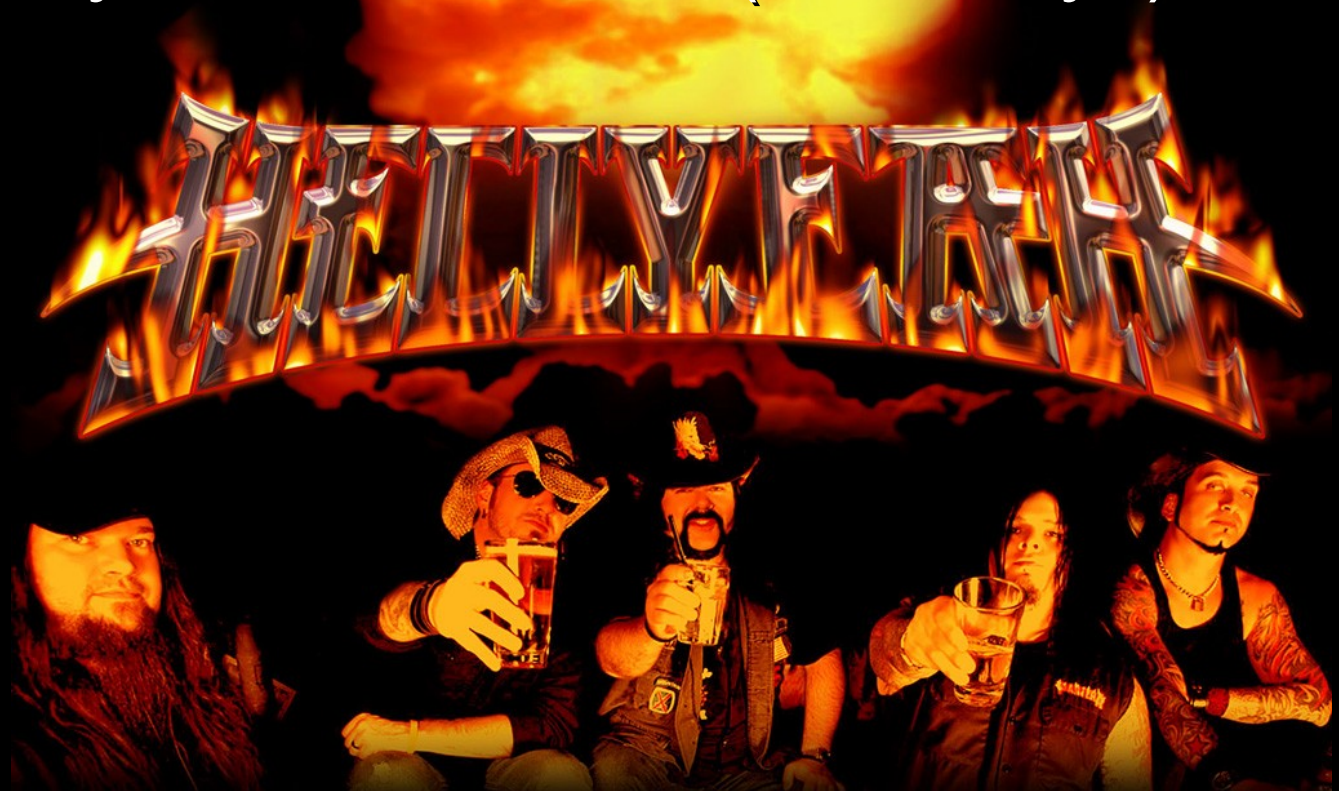
D

D?

A language that C++ could have been, should have been but will never be.

Why D is cool

- New and young (1.0: Jan 2007)
- Compiles to machine code, C++ performance
- features inspired by Ruby, Python, Java
- binary-linkable with C (both ways)



(a few) features of D

- Java-like OOP (single inheritance, interfaces)
- templates, mixins
- reflection and meta-programming
- dynamic arrays, hashmaps, strings
- GC
- exceptions with scopes
- unit tests

So how do I write
Ruby extension in D?

use

RuDy

http://github.com/tomash/rudy

inspired by Pyd (pyd.dsource.org)
from Python world

What's so cool about RuDy?

bindings (on steroids): write extensions like you would in C

C

```c
#include "ruby.h"
VALUE t_init(VALUE self)
{
  VALUE arr;
  arr = rb_ary_new();
  rb_iv_set(self, "@arr", arr);
  return self;
}

void Init_1()
{
  VALUE klass = rb_define_class("SomeClass", rb_cObject);
  rb_define_method(klass, "initialize", t_init, 0);
}
```

D

```d
module rudy.test1;
import bcd.ruby;

extern (C) VALUE t_init();
extern (C) VALUE t_init(VALUE self)
{
  VALUE arr;
  arr = rb_ary_new();
  rb_iv_set(self, "@arr", arr);
  return self;
}

extern (C) void Init_1()
{
  VALUE klass = rb_define_class("SomeClass", rb_cObject);
  rb_define_method(klass, "initialize", &t_init, 0);
}
```

Just bindings?

Hell no.

Even despite they'd be enough
by bringing D to Ruby world.

Converting to Ruby Value

Different conversion code depending on type?
Why?

C

```
VALUE rb_int2inum _((long));
#define INT2NUM(v) rb_int2inum(v)
#define LONG2NUM(v) INT2NUM(v)
#define rb_int_new(v) rb_int2inum(v)
VALUE rb_uint2inum _((unsigned long));
#define UINT2NUM(v) rb_uint2inum(v)
#define ULONG2NUM(v) UINT2NUM(v)
#define rb_uint_new(v) rb_uint2inum(v)
...
VALUE rb_float_new (double);
VALUE rb_fix2str (VALUE, int);
..
VALUE rb_str_new _((const char*, long));
VALUE rb_str_new2 _((const char*));
VALUE rb_str_new3 _((VALUE));
VALUE rb_str_new4 _((VALUE));
VALUE rb_str_new5 _((VALUE, const char*, long));
```

RuDy

```
to_ruby_value(true);
to_ruby_value(null);
to_ruby_value(2009);
to_ruby_value(20.0);
to_ruby_value("euruko");
to_ruby_value("array","of","strings"]);
float[char[]] h;
h["registry"] = 20.0;
h["plane"] = 165.5;
to_ruby_value(h);
```

Converting from Ruby value to native type?

Can you remember all the methods from API?

```c
VALUE rb_str_to_str _((VALUE));
VALUE rb_string_value _((volatile VALUE*));
char *rb_string_value_ptr _((volatile VALUE*));
char *rb_string_value_cstr _((volatile VALUE*));
#define StringValue(v) rb_string_value(&(v))
#define StringValuePtr(v) rb_string_value_ptr(&(v))
#define StringValueCStr(v) rb_string_value_cstr(&(v))
...
#define R_CAST(st)   (struct st*)
#define RBASIC(obj)  (R_CAST(RBasic)(obj))
#define ROBJECT(obj) (R_CAST(RObject)(obj))
#define RCLASS(obj)  (R_CAST(RClass)(obj))
#define RMODULE(obj) RCLASS(obj)
#define RFLOAT(obj)  (R_CAST(RFloat)(obj))
#define RSTRING(obj) (R_CAST(RString)(obj))
...
long rb_num2long _((VALUE));
unsigned long rb_num2ulong _((VALUE));
long rb_num2int _((VALUE));
#define NUM2INT(x) (FIXNUM_P(x)?FIX2INT(x):rb_num2int(
long rb_fix2int _((VALUE));
#define FIX2INT(x) rb_fix2int((VALUE)x)
unsigned long rb_num2uint _((VALUE));
#define NUM2UINT(x) rb_num2uint(x)
unsigned long rb_fix2uint _((VALUE));
```

C

RuDy

```d
int n    = d_type!(int)(obj);
double f = d_type!(double)(obj);
char[] s = d_type!(char[])(obj);
bool b   = d_type!(bool)(obj);
```

How about some nice wrapping?

for true OOP and convenient operator overload?

Well, there's RudyObject

C

```c
VALUE val1 = to_ruby_value("euruko2009");
VALUE val2 = to_ruby_value("euruko2008");
int res = rb_equal(val1, val2);
return res != 0;
```

RuDy

```
auto val1 = new RudyObject(to_ruby_value("euruko2009"));
auto val2 = new RudyObject(to_ruby_value("euruko2008"));
return val1 == val2;
```

Defining functions in Ruby...

...could use some love as well!

C

```c
VALUE c = rb_const_get(rb_cObject, rb_intern("SomeClass"));
rb_define_method(c, "my_method", &my_method, 0);
VALUE m = rb_const_get(rb_cObject, rb_intern("SomeModule"));
rb_define_module_function(m, "my_method", &my_method, 0);
```

RuDy

```
def!("SomeClass", my_method);
def!("SomeModule", my_method);
```

(RuDy still needs some development here)

What is your wish?

"**Combine the above:**
I'd like to have
my function taking and ret'ing native D types,
arguments converted Ruby->D,
return value converted D->Ruby,
defined by def!("Scope",my_method);
all automagically".

Possible? **Yes**, it's in PyD.

Here yet? Nope, but in weeks to come.

Any other wish?

"I'd like to have D class
being exposed fully to Ruby
with def!(MyClass)
with its public methods wrapped
by def!(method) described above."

It's in PyD, it's coming to RuDy.

About to come to RuDy near you

D function/delegate converted by to_ruby_value to callable Ruby Proc/lambda

Sensible build system (D-aware extconf)

Wished-for features

(and some fellow developers, I hope ;)

Where can I get it?

code: **http://github.com/tomash/rudy**

contact me: http://tomash.wrug.eu

note:
RuDy is far from 1.0
Features described above are fully working
and covered with unit tests,
but RuDy still needs a lot of work
(on the new wished features ;)

Before you start hacking...

Remember the rules of Metal!

\m/

Thank you!

Q&A time?