# Mirror, mirror on the wall

Which language has the best introspection of all?

Átila Neves, Ph.D.

DConf Online 2022

## Reflection — what?

- The ability to inspect code.
- What functions and types are in a particular module?
- What parameters does this function take?

# Reflection — what for?

- Code generation.
    - Serialisation.
    - Pretty printers.
    - Interfacing to other languages.
- Custom testing frameworks.
- In short: avoiding boilerplate.

# Reflection in D

- Compile-time instead of at run-time.
- `std.traits`
- `__traits`
- Requires metaprogramming.

# Template Metaprogamming

- Not quite D.
- Not quite regular programming.
  - Functions → metafunctions
  - `auto`/`const` → `alias`/`enum`
  - `std.algorithm.map` → `std.meta.staticMap`
- **Much** easier than in C++ but still not **easy**.
  - `static foreach`
  - Indexing into variadic template arguments.
  - Anything can be a template argument.
  - Built-in arrays/slices.
- Implementation problems.
  - Compile times.
  - Symbol emission bugs.

# Metafunctions

```
import std.meta;
enum isLarge(T) = T.sizeof > 4;
pragma(msg, Filter!(isLarge, int, double));
// (double)

struct Struct {
  int foo();
  double foo(int);
}
alias overloads(string name) = __traits(getOverloads, Struct, name);
pragma(msg, staticMap!(overloads, "foo"));
// tuple(foo, foo)
```

# Mirror

- An attempt at a better, centralised, API.
- `https://github.com/atilaneves/mirror`
- Unfinished, experimental.
- Extended RTTI support à la Java.
- Reflection: metaprogramming with symbols.
- Reflection: CTFE API with values/strings.

# Mirror: Better RTTI

```
abstract class Abstract {}
class Class: Abstract {
    int i;
    string s;
}
const Abstract obj = new Class;
with(types!Class) { // register types
    const type = rtti(obj);
    type.fields.map!(a => a.type.typeInfo).should ==
        [
            typeid(int),
            typeid(string),
        ];
}
```

# Mirror: Reflection via metaprogramming/symbols

```
alias mod = Module!("modules.functions");
import modules.functions; // so add1, etc. are visible

alias expected = AliasSeq!(
    FunctionSymbol!(add1, Protection.public_, Linkage.D),
    // ...
);

// even custom comparison function...
shouldEqual!(mod.FunctionsBySymbol, expected);
```

# Mirror: Reflection via CTFE/values

```
enum mod = module_!"modules.functions";
const add1 = mod.functionsByOverload[0]; // no enum
add1.should ==  // regular equality
    Function(
        "modules.functions",
        0,
        "add1",
        Type("int", int.sizeof),
        [
            Parameter(type!int, "i"),
            Parameter(type!int, "j"),
        ],
    );
```

# Mirror: Reflection via CTFE/values: mixins

```
enum mod = module_!"modules.functions";
enum add1 = mod.functionsByOverload[0]; // this time: enum

mixin(add1.importMixin); // so the symbol is visible

mixin(add1.fullyQualifiedName, `(1, 2)`).should == 4;
mixin(add1.fullyQualifiedName, `(2, 3)`).should == 6;
```

# Wrapping Python

```d
// D module dmodule with functions foo and bar
extern(C) export PyObject* PyInit_mymodule() {
  return createPythonModule!("mymodule", foo, bar)
}
extern(C) PyObject* foo(PyObject* self, PyObject* args) {
    auto dRet = dmodule.foo(PyTuple_GetItem(args, 0).to!int);
    return dRet.toPython;
}
extern(C) PyObject* bar(PyObject* self, PyObject* args) {
    auto dRet = dmodule.bar(
        PyTuple_GetItem(args, 0).to!double,
        PyTuple_GetItem(args, 1).to!string);
    return dRet.toPython;
}
```

# Wrapping Python

```
string init(in string moduleName, in Module module_) @safe pure {
    const functionNames = module_
        .functionsBySymbol
        .map!(f => f.identifier)
        .join(", ");

    return q{
        extern(C) export PyObject* PyInit_%s() nothrow {
            import python.d: createPythonModule;
            return createPythonModule!("%s", %s);
        }
    }.format(moduleName, moduleName, functionNames);
}
```

# Wrapping Python

```d
string cfunctions(OverloadSet[] overloads) @safe pure {
    string ret;

    foreach(overloadSet; overloads) {
        assert(overloadSet.overloads.length == 1, "No overloads yet");
        const fun = overloadSet.overloads[0];
        ret ~= functionDef(fun);
    }

    return ret;
}
```

# Advantages of mixin-based programming

- Compile times!
- Regular programming: imperative, OOP, or functional.
- Even easier than D’s templates.

# Disadvantages of mixin-based programming

- So easy it's. . . hard?
- No prior art.
    - Loop and generate unrolled code or generate loops?
    - mixins within mixins?
    - How to organize the code?
- Minor: always having to import modules.
- The generated strings will have bugs.
- Lack of string interpolation.
- How to compartmentalise?

# Conjuring functions out of the ether with templates

```
/**
The C API implementation that calls a D function F.
*/
template PythonFunction(alias F) {
    static extern(C) PyObject* _py_function_impl(
        PyObject* self, PyObject* args) nothrow
    {
        return noThrowable!(callDlangFunction!(void, F))(self, args);
    }
}
// then...
PyMethodDef method; // name, ptr, flags, docstring
method.ml_meth = &PythonFunction!myDFunction._py_function_impl;
```

# Concluding

- D’s built-in batteries for reflection could use an upgrade.
- Template metaprogramming is harder and weirder.
- Generating strings isn’t a panacea.

# Questions?

Slide intentionally left blank