Programming ≫ Metaprogramming

Sometimes you just want a foreach

Átila Neves, Ph.D.

DConf 2025

Why?

- Andrei had a DConf 2018 talk about reflection with string mixins
- I had done enough reflection work to want a library
- My hammer was templates, reflection looked like a nail
- I wrote a string based API but as an afterthought
- The string API called into. . . templates

My path to D templates

- Before D: Spectrum Basic, Pascal, Common Lisp, C, Java, C++, Perl
- Interview question: write a compile-time factorial (pre-C++11)

```
template<int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template<>
struct Factorial<0> {
    enum { value = 1 };
};
```

Let's do everything at compile time

- constexpr: the fourth language?
- String mixins - what for?

Challenge Accepted

- A colleague got into Go
- Task: arrange strings contiguously in memory
- Do it at compile time with C++11?

The craziest code I ever wrote

```
// https://atilaoncode.blog/2015/02/11/the-craziest-code-i-ever-wrote/
    template<size_t N>
    struct String {
        template<typename... Args>
        constexpr String(Args... args):_str{ args... } { }
        const char _str[N];
    };
    //makeStringFromChars('f', 'o', 'o', '\0') -> String<4>::_str = "foo"
    template<typename... Args>
    constexpr auto makeStringFromChars(Args... args) ->
    String<sizeof...(Args)> {
      return String<sizeof...(args)>(args...);
    }
```

The craziest code I ever wrote

```
// https://atilaoncode.blog/2015/02/11/the-craziest-code-i-ever-wrote/

    //Takes a variadic list of string literals and returns a tuple of
    //String<> objects.  These will be contiguous in memory.
    template<size_t... Sizes>
    constexpr auto makeStrings(const char (&...args)[Sizes]) ->
    Tuple<String<Sizes>...> {
        //makeTuple(makeString(arg1), makeString(arg2), ...)
        return makeTuple(myMakeString(args)...);
    }
```

The 89 line C++ monstrosity in D

```
auto makeStrings(A...)(A args) {
    Tuple!A ret;
    foreach(i, arg; args)
        ret[i] = arg;
    return ret;
}

enum strings = makeStrings("foo", "bar", "baz");
```

mirror: a D reflection library

- Why a library for a famous D feature?
- Because this would be nice:

```
alias mod = Module!"modules.types";
typeNames!mod.shouldBeSameSetAs([
    "BaseClass",
    "ChildClass",
    "IMyThing",
]);
private string[] typeNames(alias module_)() {
    enum name(alias Symbol) = __traits(identifier, Symbol);
    return [ staticMap!(name, module_.Aggregates) ];
}
```

Using mirror — template metaprogramming

```
template Functions(alias mod_, bool alwaysExport, Ignore[] ignoredSymbols)
    if(!is(typeof(mod_) == string))
{
    alias mod = Module!(moduleName!mod_);
    enum isExport(alias F) = isExportFunction!(F.symbol, alwaysExport);
    enum shouldIgnore(alias F) = ignoredSymbols.canFind!
        (a => a.identifier == F.identifier);

    alias Functions = Filter!(templateNot!shouldIgnore,
                              Filter!(isExport, mod.FunctionsBySymbol));
}
```

Metaprogramming: the good

- Subjectively, it's pretty cool
- You get to feel smart
- The compile times mean you have time to get another coffee
- You can finally justify buying 128 GB of RAM

Metaprogramming: the bad

- `alias` vs `enum` instead of `auto` everywhere (`const`, cough)
- No lambdas so every intermediate metafunction needs a name
- Using `enum` to introduce a name requires another set of curlies
- Compile times
- Debugging, readability, tooling
- Writing tests is harder

Metaprogramming: the ugly

- Eponymous templates and refactoring do not mix
- A language within a language
- Ordering issues when checking if an expression compiles
- Compilation errors might be silenced
- Other inexistent features (e.g. static switch, ternary operator, UFCS)

Another way: CTFE and string mixins

- Write completely regular functions, run them at CT (or not)
- Use structs/strings to store information intead of templated symbols
- Generate code by mixing in the aforementioned strings

Another way: structs/classes/strings

```
class Function: Member {
    size_t overloadIndex;
    string[] attributes;
    Type returnType;
    Parameter[] parameters;
    bool isDisabled;
    size_t virtualIndex;
    bool isVirtualMethod;
    bool isAbstract;
    // ...
}
```

CTFE/strings: downsides

- Modules must be imported before naming their symbols:

```
enum func = getSymbol;
mixin("static import ", func.moduleName, ";");
mixin(func.fqn, "(args);");
```

- Slightly harder to generate a function:

```
auto funcptr = &MyTemplate!(args).impl;
```

CT reflection ⇒ RT reflection

- RTTI can be generated from compile-time information
- The structs/classes/strings can be (re)used at runtime
- Enables: plugins, calling pre-compiled code at runtime

```
const fun = mod.functionsByOverload[0];
fun([Variant(1), Variant(2)]).get!int.should == 4;
fun([Variant(2), Variant(3)]).get!int.should == 6;
```

CT reflection ⇒ RT reflection

```
module mymodule;
mixin registerModule!(); // creates gModuleInfo here
// elsewhere, at runtime
allModuleInfos[0].fullyQualifiedName.should == "mymodule";
```

Wrapping Python

- Python extension modules written in C (or C++, D, Rust, ...)
- For an extension module named "foo":

```
extern(C) export PyObject* PyInit_foo() {
  // ...
  return pyModuleCreate(&moduleDef);
}
```

Wrapping D for Python the normal way

- Loop over all D modules
- Find all functions in each D module
- For each D function, create a `PythonCFunction`:

```
PyObject* theAnswer(PyObject* self, PyObject *args) {
    return PyLong_FromUnsignedLong(42UL);
}
```

Wrapping D for Python at runtime

```
private extern(C) PyObject* PyInit_libpython_wrap_ctfe() {
    // ...
    auto pyMod =  pyModuleCreate(&moduleDef);
    foreach(module_; allModuleInfos) {
        foreach(function_; module_.functionsByOverload) {
            addMethod(pyMod, function_);
        }
    }
    return pyMod;
}
void addMethod(PyObject* pyMod, Function function_) { /* ... */ }
```

Wrapping D for Python at runtime

```
void addMethod(PyObject* pyMod, in Function function_) {
    static PyObject* wrappedCall(PyObject* self, PyObject* args) {
        auto variants = new Variant[PyTuple_Size(args)];
        foreach(const i, ref v; variants)
            v = toVariant(PyTuple_GetItem(args, i));
        auto func = cast(Function) PyCapsule_GetPointer(self, "f");
        return func(variant).toPython;
    }
    auto methodDef = wrapperCallMethDef(&wrapperCall);
    auto capsule = PyCapsule_New(cast(void*) function_, "f", null);
    auto pyFunction = PyCFunction_NewEx(methodDef, capsule, null);
    PyModule_AddObject(pyMod, function_.identifier.toStringz, pyFunction);
}
```

What's next?

- Use CTFE instead of templates in autowrap
- Measure compile times / memory usage
- Profit?

Conclusion

- Regular D is much easier and ergonomic than templated code
- The reflection data can be used at runtime when using CTFE

Links

- Mirror: `https://github.com/atilaneves/mirror`
- Monty: `https://github.com/atilaneves/monty`
- Autowrap: `https://github.com/symmetryinvstments/autowrap/`

Questions?

Slide intentionally left blank