

Metaprogramming in D: Real world examples

Bill Baxter

Northwest C++ Users Group

November 18, 2009

What is D?

- Systems programming language
- Development led by Walter Bright
- Compiles to native code
- Garbage collected
- Under continuous development for 10 years
- “C++ without the mistakes”
- Open source
- Some very nice metaprogramming features

What is metaprogramming?

“Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime.”

(Wikipedia definition)

Metaprogramming use cases

- Compile time constants
- Static checking
- Code specialization using introspection
- Code generation

D Metaprogramming: *Dramatis Personae*

- templates
- static assert
- static if
- Compile time function evaluation (CTFE)
- pragma(msg, ...)
- .stringof
- is()
- string mixin
- __traits

D Templates

- Much like C++ templates

C++:

```
template<class T> T add(T a, T b)
{ return a+b; }
```

C++ call syntax:
add<int>(2,3)

D:

```
template T add(T)(T a, T b)
{ return a+b; }
```

D Call syntax:
add!(int)(2,3)
add!int(2,3)
add(2,3)

or

```
T add(T)(T a, T b)
{ return a+b; }
```

Use case: Compile-time Constants

Classic C++ functional-style definition

```
template <int i>
struct factorial {
    static const int value = i*factorial<i-1>::value;
};
```

```
template <>
struct factorial<0> {
    static const int value = 1;
};
```

```
static const int X = factorial<5>::value;
```

Factorial – D template version

```
struct factorial(int i) {
    static const value = i*factorial!(i-1).value;
}
struct factorial(int i : 0) {
    static const value = 1;
}
static const X = factorial!(5).value;
```

OR

```
struct factorial(int i) {
    static if(i==0) {
        static const value = 1;
    }
    else {
        static const value = i*factorial!(i-1).value;
    }
}
```

Factorial – D template version(2)

Or better, eliminate unnecessary struct using the eponymous template member trick

```
template factorial(int i) {  
    static if (i==0) {  
        enum factorial = 1;  
    }  
    else {  
        enum factorial = i*factorial!(i-1);  
    }  
}  
enum X = factorial!(5);
```

Factorial – D CTFE

```
int factorial(int i) {  
    return (i==0)? 1 : i*factorial(i-1);  
}  
enum X = factorial(5);  
auto y = factorial(5);
```

- Plain functions can be evaluated at compile time!
(Or runtime, of course)
- Ok, classic example. Check. But have you ever really needed to do this in real-world code? Yeh, me neither.

A possibly more useful example

```
enum string url = urlencode("a url with spaces")
```

- `assert(url=="a%20url%20with%20spaces")`

Use case: Static checking

- Check template parameter values
- for non-trivial condition
- At compile-time

Three examples:

- (values) checking if token in grammar
- (values) proper linear congruential parameters
- (types) checking if type supports concept

Example:

Static checking of grammar tokens

- Want statically checked representation for grammar tokens
 - Could make them be identifiers : TOK_if
 - But some tokens are not valid identifiers
 - TOK_plus instead of TOK_+
 - Solution: Statically checked template Tok! "+"
-
- Idea thanks to Nick Sabalausky & his Goldie parser
 - <http://www.dsource.org/projects/goldie>

```
enum ValidTokens = ["+", "-", "sin", "cos"];
```

```
bool isValidToken(string tok) {  
    foreach(t; ValidTokens) {  
        if (t==tok) return true;  
    }  
    return false;  
}
```

```
struct Token(string Name) {  
    static assert(isValidToken(Name), "Invalid token: "~Name);  
    // ...  
}
```

```
Token!(Name) Tok(string Name)() { return Token!(Name)(); }
```

```
auto O = Tok!"cos";  
auto M = Tok!"+";  
auto Y = Tok! "-";  
auto X = Tok!"*"; // Error: static assert "Invalid token: *"
```

Example 2: Static checking proper linear congruential parameters

- A simple kind of pseudorandom number generator based on three parameters: a , c , m

$$x_{n+1} := (ax_n + c) \bmod m$$

- Good parameters satisfy:
 1. c and m are relatively prime;
 2. $a-1$ is divisible by all prime factors of m ; and
 3. if $a-1$ is multiple of 4, then m is a multiple of 4 too.
- Thanks to Andrei Alexandrescu for this one (used in D's `std.random`)

```
bool properLinConParams(ulong m, ulong a, ulong c)
{
// Bounds checking
if (m == 0 || a == 0 || a >= m || c == 0 || c >= m) return false;

// c and m are relatively prime
if (gcd(c, m) != 1) return false;

// a - 1 is divisible by all prime factors of m
if ((a - 1) % primeFactorsOnly(m)) return false;

// if a - 1 is multiple of 4, then m is a multiple of 4 too.
if ((a - 1) % 4 == 0 && m % 4) return false;

// Passed all tests
return true;
}
```

```
// Example bad input
static assert(!properLinConParams(1UL<<32, 210, 123));
// Example from Numerical Recipes
static assert(properLinConParams(1UL<<32, 1664525, 1013904223));
));
```

Usage example

```
struct LinearCongruentialEngine(  
    UIntT, UIntT a, UIntT c, UIntT m)  
{  
    static assert(properLinConParams(m, a, c),  
        "Incorrect instantiation of  
        LinearCongruentialEngine");  
    ...  
}
```

Creating one of these with bad parameters is now impossible, and results in a **compile-time** error

Example 3: Concept checking

- Do introspection on types
- Make sure they support a particular interface concept.
 - E.g. does this type have `.length` and allow indexing?
- Elaborate Concept support planned for C++0X but collapsed under own weight at the end.
- D offers some similar capabilities

Vector example

- Say we want to create a Vector. It should accept a scalar type which is closed under:
 - Addition
 - Subtraction
 - Multiplication
 - Division

```
struct Vector(S) {  
    static assert(isScalarType!(S),  
                  S.stringof ~ " is not a proper scalar type");  
    //...  
}
```

Now, to define isScalarType...

Take Zero

- Just enumerate the allowable types

```
template isScalarType(S)
{
    enum isScalarType =
        is(S == float) ||
        is(S == double) ||
        is(S == real) ||
        is(S == int) ||
        is(S == long);
};
```

```
Vector!(float) a; // ok!
Vector!(string) b; /* nope:
Error: static assert
"string is not a proper
scalar type"
*/
```

- In C++ a bunch of specializations, or some funky typelist thing.
- Doesn't extend to user types.
- Call out the ducks!

Take One

```
template isScalarType(S)
{
    enum isScalarType =
        is(typeof(S.init + S.init) == S) &&
        is(typeof(S.init * S.init) == S) &&
        is(typeof(S.init - S.init) == S) &&
        is(typeof(S.init / S.init) == S);
}
```

- Looking much more ducky now
- We can clean it up a bit more with a small trick...

Take Two

- { ...code } is a delegate literal in D
- is(typeof(...)) returns false if the arg is not valid.

```
template isScalarType(S)
{
    enum isScalarType = is(typeof({
        S v,r;
        r = v+v;
        r = v-v;
        r = v*v;
        r = v/v;
    }));
}
```

- Main drawback to this kind of static concept checking is lack of specific error messages. (E.g. “Not a scalar because it doesn’t support addition”)
- See also: `__traits(compiles, expr)`

Template constraints

- D allows this kind of “constraint syntax” too:

```
struct Vector(S) if(isScalarType!(S)) {  
    //...  
}
```

- Can overload templates on different `if(...)` constraints
- Problem is still error messages:
 - “Error: Template instance `Vector!(string)` does not match template declaration `Vector(S) if (isScalarType!(S))`”
-- well? why not?

A cumbersome solution

```
template checkScalarType(S)
{
    static if(!is(typeof(S.init + S.init) == S))
        pragma(msg, S.stringof ~ "Error: is not closed under addition");
    static if(!is(typeof(S.init - S.init) == S))
        pragma(msg, S.stringof ~ "Error: is not closed under subtraction");
    static if(!is(typeof(S.init * S.init) == S))
        pragma(msg, S.stringof ~ " is not closed under multiplication");
    static if(!is(typeof(S.init / S.init) == S))
        pragma(msg, S.stringof ~ " is not closed under division");

    enum isScalarType = is(typeof({
        S v,r;
        r = v+v;
        r = v-v;
        r = v*v;
        r = v/v;
    }));
}
```

Compare with

- Go:

```
interface ScalarType<T>
{
    T operator+(T,T);
    T operator-(T,T);
    T operator*(T,T);
    T operator/(T,T);
}

struct Vector {
    ScalarType x;
}
```

- Er... except Go doesn't have operator overloading
- or generics.

Ok.. Compare with

- C++1X concepts -- not going in C++0x, but probably will some day

```
concept ScalarType<typename S>
{
    Var<S> v,r;
    r = v+v;
    r = v-v;
    r = v*v;
    r = v/v;
};

template<typename S>
    where ScalarType<S>
struct Vector {
    ...
};
```

- Here compiler will do the line-by-line checking of the concept against input, and give decent errors.

Code specialization w/ introspection

- Erroring if type doesn't support concept is often not as useful as working around the missing functionality.
- Example – just omit Vector ops if underlying scalar op not defined.

```
struct Vector(S)
{
    S[3] values;
    static if (is(typeof(S.init + S.init)==S)) {
        Vector opAdd(ref Vector o) {
            Vector ret;
            foreach(i; 0..3) {
                ret.values[i] = values[i] + o.values[i];
            }
            return ret;
        }
    }
}
```

```
Vector!(float) a;
Vector!(char) b;
a+a; // ok
b+b; // error
```

Code Generation

- Examples: loop unrolling, vector swizzles
- Key enabling feature: string mixin.
- Simple example from documentation:

```
string s = "int y;";  
mixin(s); // ok  
y = 4;    // ok, mixin declared y
```

- Now the one-two punch: The strings themselves can come from CTFE:

```
string declareInt(string name) {  
    return "int " ~ name ~ ";";  
}  
  
mixin(declareInt("y"));  
y = 4;
```

Codegen example:

Loop unrolling

- Handy in a `Vector(T, int N)` type.

```
string unroll(int N,int i=0)(string expr) {
    static if(i<N) {
        string subs_expr = ctReplace(expr, "%s", toStringNow!i);
        return subs_expr ~ "\n" ~ unroll!(N,i+1)(expr);
    }
    return "";
}
```

Example use

```
enum string expr = "values_[%s] += _rhs[%s];";
pragma(msg, unroll!(3)(expr));
```

Compile-time output:

```
values_[0] += _rhs[0];
values_[1] += _rhs[1];
values_[2] += _rhs[2];
```

- Typical use:

```
struct Vector(T, int N) {
    T[N] values_;
    ...
    /// this += rhs
    ref Vector opAddAssign(/*const*/ ref Vector rhs) {
        const string expr = "values_[%s] += rhs.values_[%s]";
        mixin( unroll!(N)(expr) );
        return this;
    }
    ...
}
```

Codegen example: Swizzle functions

- Modern GPU shading languages support special “swizzle” syntax for shuffling values in a vector
- Examples:

```
float4 A(1,2,3,4);  
float4 B = A.xxwz;  
assert( B == float4(1,1,4,3) );
```

```
float2 C = A.yz;  
assert( C == float2(2,3) );
```

```

/** Generate a swizzling function.
    Example: _gen_swizzle("xyzyz") returns
    """"
    VectorT!(T,6) xyzyz() const
    {
        return VectorT!(T,6)(x,y,z,z,y);
    }
    """"
*/

```

```

string _gen_swizzle(string swiz) {
    string args = ""~swiz[0];
    foreach(c; swiz[1..$]) args ~= ", " ~ c;
    string code =
        ctFormat(q{
            VectorT!(T,%s) %s() const
            {
                return VectorT!(T,%s)(%s);
            }
        },
        swiz.length, swiz, swiz.length, args);
    return code;
}

```

```
• B
le
string _gen_all_swizzles(int len, int srcdig)
{
    if (len <= 0) return "";
    string elem = "xyzw"[0..srcdig];
    string code;

    foreach (int genlen; 2..len+1)
    {
        int combos = 1;
        foreach(a; 0..genlen) { combos *= srcdig; }
        foreach(i; 0..combos) {
            string swiz;
            int swizcode = i;
            foreach(j; 0..genlen) {
                swiz ~= elem[swizcode % srcdig];
                swizcode /= srcdig;
            }
            code ~= _gen_swizzle(swiz);
        }
    }
    return code;
}
```

```
VectorT!(T,2) xx()  
{  
    return VectorT!(T,2)(x, x);  
}
```

```
VectorT!(T,2) yx()  
{  
    return VectorT!(T,2)(y, x);  
}
```

```
VectorT!(T,2) xy()  
{  
    return VectorT!(T,2)(x, y);  
}
```

```
VectorT!(T,2) yy()  
{  
    return VectorT!(T,2)(y, y);  
}
```

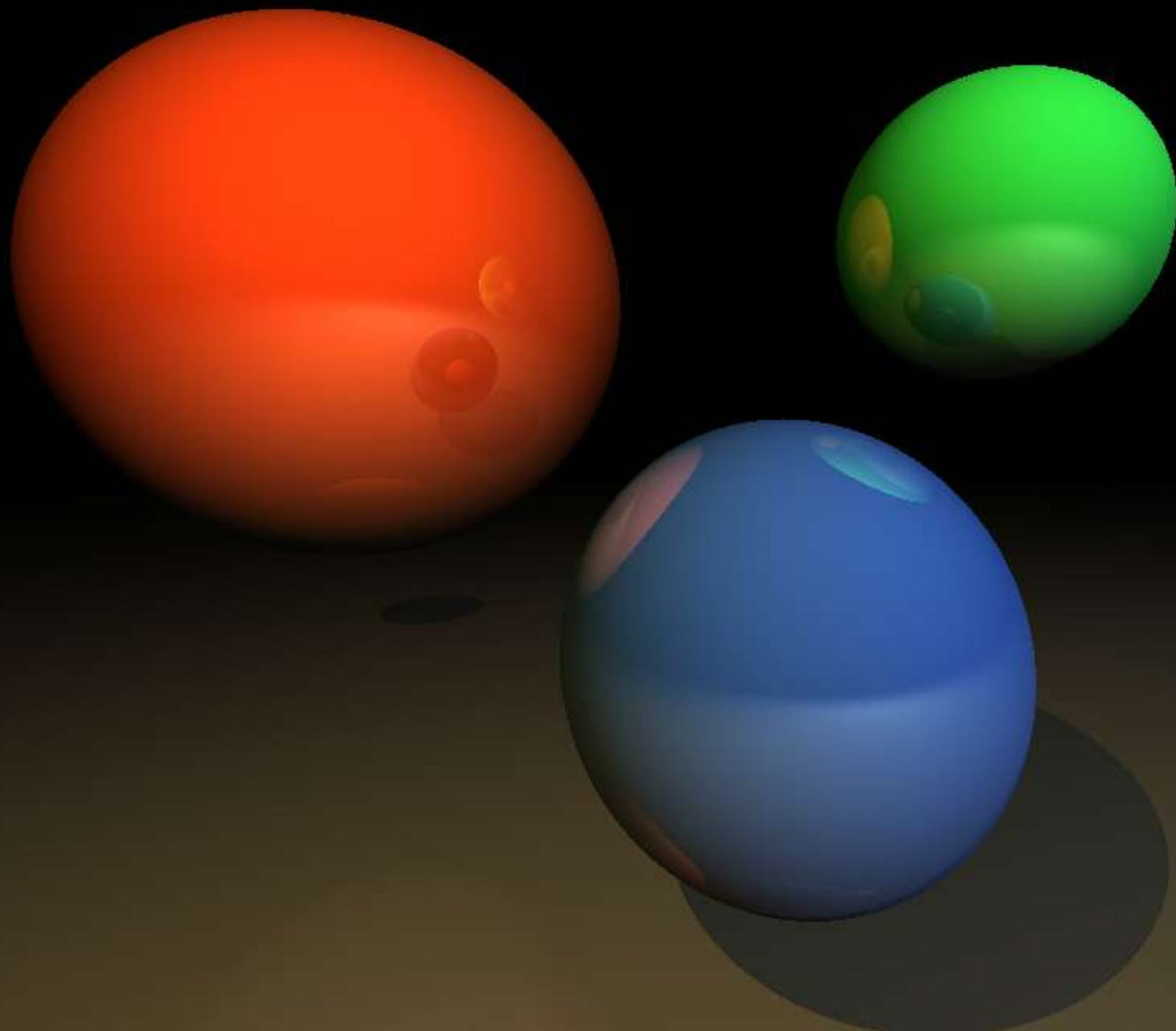
```
VectorT!(T,3) xxx()  
{  
    return VectorT!(T,3)(x, x, x);  
}
```

```
VectorT!(T,3) yxx()  
{  
    return VectorT!(T,3)(y, x, x);  
}
```

...

Are there limits to the possibilities?

- Almost no.
- Which is to say, yes.
 - CTFE limitations: no structs/classes, no C funcs
 - Compiler bugs: massive memory leak in CTFE
- However, these have been demonstrated:
 - Compile-time raytracer (Tomasz Stachowiak)
 - Wrapper generation (PyD, Kirk McDonald)
 - Compile-time regular expression parsing (meta.regexp, Eric Anderton).
 - Compile-time parsing DSL parsing (???, BCS?)



Total rendering/compilation time: 26148 seconds (on a 1.7GHz laptop)

Misc observations

CTFE vs Templates

- CTFE calc vs Template calc:
 - Templates can do calculations on **Types** or **Values**
 - CTFE can only do **Value** calculations
 - Template → CTFE is OK
 - But not CTFE → Template
 - CTFE supports richer syntax
 - Also Templates instantiate symbols and cause bloat
- Conclusion
 - If doing value calcs, always prefer CTFE.

CTFE vs AST macros

- Nemerle macro:

```
macro m () {  
    Nemerle.IO.printf ("compile-time\n");  
    <[ Nemerle.IO.printf ("run-time\n") ]>;  
}  
  
m ();
```

- D “macro”:

```
string m() {  
    pragma(msg, "compile-time");  
    return q{writefln("run-time");}  
}  
  
mixin(m());
```

About CTFE

- CTFE is constant folding on steroids.
 - “Simply taking constant folding to its logical conclusions”
– Don Clugston
- `const int x = 2+4; // many compilers will fold this`
- `int add(int a, int b){return a+b;}`
`const int x = add(2,4); // some may inline/fold this`
- `const int x = factorial(5); // does any but DMD do this?`
`// maybe some functional languages?`
- In general, why not evaluate as much as you can up front?

```
macro For (init, cond, change, body) {
  <[$init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}
```

```
For(mutable i = 0, i < 10, i++, printf ("%d", i))
```

```
string For(string init, string cond, string change, string bod)
{
  return ctFormat(q{
    %s;
    void loop() {
      if (%s) { %s; %s; loop(); }
    };
    loop();
  }, init, cond, bod, change);
}
```

```
void main() {
  mixin(For("int i = 0", "i < 10", "i++",
    q{writeln("%d", i)}));
}
```

```
macro For (init, cond, change, body) {
  <[$init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}
```

```
For(mutable i = 0, i < 10, i++, printf ("%d", i))
```

```
string For(string init, string cond, string change, string bod)
{
  return mixin(ctInterpolate(q{
    $init;
    void loop() {
      if ($cond) { $bod; $change; loop(); }
    });
    loop();
  });
}
```

```
void main() {
  mixin(For("int i = 0", "i < 10", "i++",
    q{writeln("%d", i)}));
}
```

Acknowledgements

- Nick Sabalausky – encode/decode, tokens examples
- Andrei Alexandrescu – std.random example
- Tom Stachowiak – swizzle example and ctrace
- Walter Bright – for D

References

- Reis, Stroustrup, "Specifying C++ Concepts"
Annual Symposium on Principles of Programming Languages, 2006.
- meta.regex: <http://www.dsource.org/meta>
- Ctrace raytracer: <http://h3.teamoxf.com/ctrace/>
- std.random:
http://www.digitalmars.com/d/2.0/phobos/std_random.html
- An excerpt from Andrei's upcoming TDPL book:
<http://erdani.com/tdpl/excerpt.pdf>