A Look at Type Introspection in Phobos v3

by Jonathan M Davis

DO IT IN D
LONDON '25

Overall Goals

- Fix mistakes that we can't fix in Phobos v2 without breaking code.
- Apply the lessons we've learned over the years towards improving the design.
- Improve documentation.
- Improve tests.
- Improve some of the symbol names.

Phobos v2 -> v3

- std.traits -> phobos.sys.traits
- std.meta -> phobos.sys.meta

traits and meta

- **__traits** and trait templates give information about symbols.

- Meta templates operate on **AliasSeq**s (i.e. alias sequences)

traits

```
isDynamicArray!Foo
```

```
EnumMembers!Foo
```

```
__traits(isCopyable, Foo)
```

```
__traits(getOverloads, Foo, "bar")
```

AliasSeq

```d
alias AliasSeq(TList...) = TList;
```

```d
template AliasSeq(TList...)
{
    alias AliasSeq = TList;
}
```

meta

```
alias Types = AliasSeq!(string, int, int[], bool[],
                        ulong, double, ubyte);

static assert(is(Filter!(isDynamicArray, Types) ==
                 AliasSeq!(string, int[], bool[])));
```

```
static assert(is(Reverse!(int, byte, long) ==
                 AliasSeq!(long, byte, int)));
```

Some Design Choices

- Avoid implicit conversions as the default (e.g. enums are not their base type).
- Have traits operate on types unless they need to operate on symbols which aren't types.
- Minimize magic.
- Make what traits do as clear as possible.
- Give control rather than make assumptions.

Implicit Conversions are Problematic

- A template constraint only decides whether that template will be instantiated.
- Template specializations only decide which types match that particular overload.
- Implicit conversions are not actually forced.
- The code may fail to compile without the conversion, or it may work but do the wrong thing.

Template Specialization

```d
T foo(T : long)(T t)
{
    return t + (long.max / 3);
}

void main()
{
    // compiles
    auto l = foo(long.init);

    // does not compile
    auto i = foo(int.init);
}
```

Template Constraint Which Allows Enums

```
enum E { a = 1, b = 4 }

T foo(T)(T t)
    if(isIntegral!T)
{
    return t + 22;
}

void main()
{
    // compiles
    auto i = foo(42);

    // does not compile
    auto e = foo(E.a);
}
```

isConvertibleToString

```
auto foo(R)(R range)
    if(isForwardRange!R && isSomeChar!(ElementType!R))
{
    return range;
}

auto foo(T)(T t)
    if(isConvertibleToString!T)
{
    return foo!(StringTypeOf!T)(t);
}

void main()
{
    // no problem
    auto str = foo("hello");

    // garbage
    char[5] sArr = "12345";
    auto unsafe = foo(sArr);
}
```

Implicit Conversions

In general, to deal with implicit conversions correctly, either

1. Force the implicit conversion within the function.
2. Have a non-templated overload which takes the type being converted to.

Enums Are Not Their Base Type

```
enum S : string { a = "hello", b = "world" }

static assert( isDynamicArray!string);
static assert(!isDynamicArray!S);
static assert( isDynamicArray!(OriginalType!S));

enum E { a = 0, b = 17, c = 42 }

static assert( isInteger!int);
static assert(!isInteger!E);
static assert( isInteger!(OriginalType!E));
```

typeof Is Ambiguous

```
typeof(12 + 19)
```

```
typeof(foo("hello"))
```

```
typeof(foo())
```

```
typeof(foo)
```

typeof Is Ambiguous

```
int foo;
static assert(is(typeof(foo) == int));
```

```
enum foo = 42;
static assert(is(typeof(foo) == int));
```

```
int foo();
static assert(is(typeof(foo) ==
                ToFunctionType!(int function())));
```

typeof Is Ambiguous

```
func(foo);
```

```
auto bar = foo;
```

```
auto bar = var.foo;
```

Variables vs Functions

- The type of a variable as a symbol is the same as the type of of a variable as an expression.
- The type of a function as a symbol is a function type.
- The type of a function as an expression is the return type of that function - or it's not a valid expression.

Getter Property

- A value, variable, or enum. Using it gets its value.

- A function which

  1. can be called with no arguments - and thus can be called without parens.

  2. returns a value.

Optional Parens and **@property** Create Ambiguity

```
typeof(foo())
```

```
typeof(foo)
```

@property Makes Things Worse

- **typeof** on functions without **@property** gives the type of the function itself.

- **typeof** on functions with **@property** gives the type of the function as an expression.

  - For **@property** getter functions, **typeof** gives the return type.

  - For **@property** setter functions, **typeof** gives an error.

@property Makes Things Worse

- Without **@property**, **typeof** would be consistent for all functions.
- **@property** solves the problem in the wrong place.
  - If code is doing type introspection on the symbol itself, it always wants the type of the symbol itself.
  - If code is trying to determine the type of the symbol within an expression, then it always wants the type of the symbol as an expression.

Ideal Solution

```
typeof_sym(foo)
```

```
typeof_expr(foo)
```

Actual Solution

```
SymbolType!foo
```

```
PropertyType!foo
```

When to Use

- **SymbolType**: When getting the type of the symbol itself.
- **PropertyType**: When the symbol is going to be used in an expression as a getter property.
- **typeof**: When getting the type of a general expression.

Examples

```
isSomeFunction!foo // std.traits
```

```
is(SymbolType!foo == return)
isReturn!(SymbolType!foo)
```

```
hasIndirections!(SymbolType!foo)
hasIndirections!(PropertyType!foo)
```

```
isSignedInteger!(SymbolType!foo)
isSignedInteger!(PropertyType!foo)
```

Comparing Symbols

```
foo == bar
```

```
is(A == B)
```

```
__traits(isSame, foo, bar)
```

Comparing Symbols

```
enum a = 42;
enum b = 42;
static immutable int x = 42;
static immutable int y = 42;
static int z = 42;

static assert( isSame!(a, 42));
static assert( isSame!(42, a));
static assert( isSame!(a, b));

static assert( isSame!(x, 42));
static assert( isSame!(42, x));
static assert(!isSame!(x, y));

static assert( isSame!(z, z));
static assert(!isSame!(z, 42));
```

Comparing Symbols

```
enum a = 42;
int foo() { return 42; }
int bar() { return 42; }

static assert( isSame!(a, foo));
static assert( isSame!(foo, a));

static assert( isSame!(foo, foo));
static assert(!isSame!(foo, bar));
static assert(!isSame!(bar, foo));
```

More Template Predicates

```
template NoDuplicates(args...) {...}
```

```
template Unique(alias Pred, Args...) {..}
```

```
template staticIndexOf(args...) {...}
```

```
template indexOf(alias Pred, Args...) {...}
```

Example Predicates

```
isEqual
```

```
isSameSymbol
```

```
isSameType
```

Examples

```d
alias Types = AliasSeq!(int, float, string, Object, string);

static assert(is(Unique!(isSameType, Types) ==
                 AliasSeq!(int, float, string, Object)));
```

```d
alias values = AliasSeq!(17, 22, 49, 0, 22, 17, 99);

static assert(Unique!(isEqual, values) ==
              AliasSeq!(17, 22, 49, 0, 99));
```

```d
void foo();
alias Stuff = AliasSeq!(int, 42, foo, int, foo, string);

static assert(indexOf!(isSameSymbol!foo, Stuff) == 2);
```

isCallable Can't Work

- **isCallable** attempts to say whether the given symbol is "callable."
- This works in simple cases, but in the general case, it's not possible.
- It's not possible with templated functions.
- It's problematic with types.
- In the general case, the only way to know if a symbol is "callable" is to see whether calling it with actual arguments compiles.

Callable?

```d
// Does this count as callable?
auto t = T();
```

```d
// Does this count as callable?
auto t = T(42);
```

```d
// Until it's instantiated, it's not callable.
void foo(T)(T t)
{
    ...
}
```

Better Solution

Instead checking whether a symbol is “callable,” do one of

1. Test whether a function call compiles with a specific set of arguments.
2. Have the code only operate on types and require that the type be a function, function pointer, or delegate.

Default Initialization

- In principle, all types in D have a default value, and if a variable is not given an explicit value, it's default-initialized to its **init** value.

- In practice, there are corner cases where this is not true:

  - Structs can disable default initialization with **@disable this();**

  - Non-**static** nested structs have a context pointer.

When **T.init** Is a Problem

```d
// This works even when default initialization
// is disabled.
auto t = T.init;
foo(T.init);
```

```d
// If T is non-static nested struct, then the
// context pointer will be null.
auto t = T.init;
foo(T.init);
```

Incomplete Workaround

```d
// If T has disabled default initialization, this
// will not compile.
auto t = T();
foo(T());
```

```d
// If T is non-static nested struct, then the
// context pointer will be properly initialized.
auto t = T();
foo(T());
```

static opCall Is a Problem

```
struct S
{
    int i;

    static S opCall()
    {
        S retval;
        retval.s = 42;
        return retval;
    }
}
```

```
struct S
{
    static void opCall()
    {
    }
}
```

Better Solution

```
template defaultInit(T)
    if(is(typeof({T t;})))
{
    enum defaultInit =
        (){ T retval; return retval; }();
}
```

```
static assert(defaultInit!int == 0);

static struct S
{
    int i = 42;
}
static assert(defaultInit!S == S(42));
```

Bug in Destructor Detection

```
template hasComplexDestruction(S)
{
    static if (__traits(isStaticArray, S))
    {
        enum bool hasComplexDestruction =
            S.sizeof && hasComplexDestruction!(BaseElemOf!S);
    }
    else static if (is(S == struct))
    {
        enum hasComplexDestruction =
            __traits(hasMember, S, "__xdtor");
    }
    else
    {
        enum bool hasComplexDestruction = false;
    }
}
```

Bug in Destructor Detection

```d
static struct S2 { ~this() {} }
static struct S3 { S2 field; }
static struct S6 { S3[0] field; }

static assert( hasComplexDestruction!S2);
static assert( hasComplexDestruction!S3);
static assert(!hasComplexDestruction!S6); // fails
```

Workaround

```
enum hasComplexDestruction =
    hasDtor([__traits(allMembers, S)]);
```

```
private bool hasDtor(string[] members)
{
    foreach (name; members)
    {
        if (name == "__xdtor")
            return true;
    }

    return false;
}
```

Questions?