Mad With Power

The Hunt for New Compile-Time Idioms

by Ethan Watson

What I’ve Been Doing

- Starting my own thing? Of course I’ll use D

- Funding a game dev studio? Tricky
- Funding a tech company? Much easier

- Working on a narrative middleware tech company

Why D?

- Why spend ten times as long doing something in C++?
  - Definite cost saving case to be made here
- But it’s not widely used…

- D is a very logical choice for moving on from C++
  - Easy to pick up from C++/C#
  - Really easy to write clean code

- Use introspection to encourage maintainability

Goals of the codebase

Two things it needs to succeed at:

1. It needs to be performant
    - Good performance keeps server costs down
2. It needs to be maintainable by anyone
    - The reality of starting a company is that I probably won’t be able to full time program

How It’s Being Setup

- Multi-user collaborative toolset, with native runtime for target platforms

- Frontend in C# and WPF
  - WPF for .NET Core makes multi-platform support easier than the prior solution
  - Running on client hardware means .NET is an acceptable performance hit
  - And thanks to Binderoo, I only use C# for the UI
- Backend in D running either locally or on a server somewhere
  - Potentially hundreds of connection, absolutely needs to be performant
  - Also where people need to understand the codebase ***really quickly***

Maintainability of Code

- C++?
  - LOLNO
- C#
  - Even WPF requires massive amounts of boilerplate
- Rust?
  - Not that happy with the new macro system
- D?
  - Language features allow me to do some ***really*** interesting things that are understandable at a glance
  - How interesting? Well, that's why I'm here today

You Don’t Need To Be An Expert Here

This talk is designed to introduce compile time programming to someone unfamiliar with it, and to also show off its capabilities at the same time.

And yes, this will be in the exam.

Template Parameter Parsing

A Common Code Problem

```
void SomeFunc( bool ImportantParam, bool AnotherImportantParam );

SomeFunc( false, true );
```

A Common Solution

```
enum ImportantParam          { Off, On }
enum AnotherImportantParam   { Off, On }

void SomeFunc( ImportantParam p1, AnotherImportantParam p2 );

SomeFunc( ImportantParam.Off, AnotherImportantParam.On );
```

D Has A Library Solution

```d
import std.typecons;

alias ImportantParam = Flag!"ImportantParam";
alias AnotherImportantParam = Flag!"AnotherImportantParam";

SomeFunc( ImportantParam.no, AnotherImportantParam.yes );
```

But What About Templates?

```
struct SomeStruct( ImportantParam p1, AnotherImportantParam p2 )
{
    static if( p1 == ImportantParam.yes )
    {
        int SomeImportantInt;
    }
    static if( p2 == AnotherImportantParam.yes )
    {
        float SomeImportantFloat;
    }
}
```

But What About Templates?

```
struct SomeStruct( ImportantParam p1 = ImportantParam.no,
           AnotherImportantParam p2 = AnotherImportantParam.no )
{
    static if( p1 == ImportantParam.yes )
    {
        int SomeImportantInt;
    }
    static if( p2 == AnotherImportantParam.yes )
    {
        float SomeImportantFloat;
    }
}
```

Template Parameter Parsing

```
struct SomeStruct( ParameterList... )
{
}

alias DodgyStruct = SomeStruct!( SomeType );
alias DodgyStruct2 = SomeStruct!( 42.0f );
alias DodgyStruct3 = SomeStruct!( SomeFunc );

alias ReallyDodgyStruct = SomeStruct!( int, 42.0f, SomeFunc );
```

Template Parameter Parsing

```
struct SomeStruct( DefaultOverrides... )
   if( DefaultOverrides.length >= 0 )
{
   // Parsing the parameters goes here!
   // But how...?
   static foreach( Override; DefaultOverrides )
   {
       // Pretty neat!
   }
}
```

Template Parameter Parsing

```
import std.typecons;

alias ImportantParam = Flag!"ImportantParam";
alias AnotherImportantParam = Flag!"AnotherImportantParam";

alias DodgyStruct = SomeStruct!( ImportantParam.no,
                                 AnotherImportantParam.yes );
```

Template Parameter Parsing

```
enum Params : string
{
    Important = "Adds an int to the struct",
    AnotherImportant = "Adds a float to the struct",
}

// alias DodgyStruct = SomeStruct!( ImportantParam.yes,
//                                  AnotherImportantParam.no );

alias DodgyStruct = SomeStruct!( Params.AnotherImportantParam );
```

Template Parameter Parsing

```
enum Params : string
{
   Important = "Adds an int to the struct",
   AnotherImportant = "Adds a float to the struct",
}

bool[ Params ] ParamSet;
```

Protip: Making Bitfields From Enums

```
enum Params : string
{
    Important = "Adds an int to the struct",
    AnotherImportant = "Adds a float to the struct",
}

alias AllParams = __traits( allMembers, Params );

pragma( msg, AllParams.stringof );

// tuple("Important", "AnotherImportant")
```

Protip: Making Bitfields From Enums

```
bool[ Params.length ] SetParams;

foreach( ParamIndex, ParamName; AllParams )
{
    // We'll come back to the "mixin" keyword later...
    mixin( "enum ThisParam = Params." ~ ParamName ~ ";" );
    if( DefaultOverrides.contains( ThisParam ) )
    {
        SetParams[ ParamIndex ] = true;
    }
}
```

But… There's A Small Problem…

```
alias DodgyStruct = SomeStruct!( Params.ImportantParam,
                                 Params.AnotherImportantParam );

// Generates the same code, but a completely different type!
alias DodgyStruct = SomeStruct!( Params.AnotherImportantParam,
                                 Params.ImportantParam );
```

Protip: Reducing Template Bloat

```
struct BitfieldFrom( T ) if( is( T == enum ) )
{
    alias Members = __traits( allMembers, T );
    static foreach( Index, Val; Members )
    {
        mixin( "@property bool " ~ Val ~ "() const"
        ~ "{ return IsSet[" ~ Index.stringof ~ "]; }" );
    }

    bool[ Members.length ] IsSet; // You can do further binary logic here

    this( T[] SetThese... );
}
```

Protip: Reducing Template Bloat

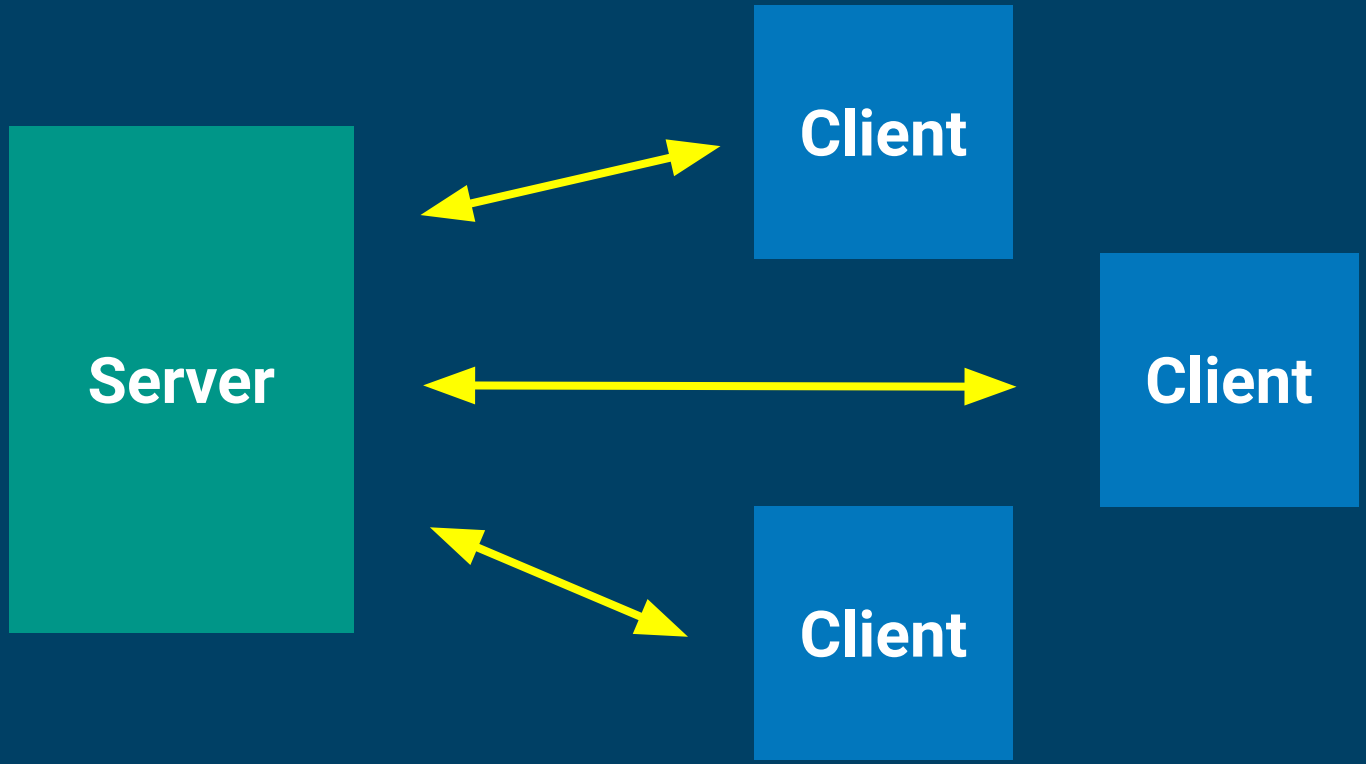
```
alias ThisBitfield = BitfieldFrom!Params;

// All of the following get transformed in to Params[]
ThisBitfield someBits       = ThisBitfield( Params.Important,
                                            Params.AnotherImportant,
                                            Params.ReallyImportant );
ThisBitfield someOtherBits  = ThisBitfield( Params.SomethingImportant );
```

Protip: Reducing Template Bloat

```
struct SomeDodgyStruct( BitfieldFrom!Params TheseParams )
{
    static if( TheseParams.ImportantParam )
    {
        int SomeImportantInt;
    }
    static if( TheseParams.AnotherImportantParam )
    {
        float SomeImportantFloat;
    }
}
```

Code Generation With External Introspection

A Simple Client-Server Application



Messaging In A Client-Server Application

What is a Message?

- An action with associated parameters

What else is an action with associated parameters?

- A function
  - Implemented across a network? Remote procedure call

What is a set of parameters?

- A meaningful aggregate
- Oh hey, just like a struct!
  - The type explicitly indicates the message
  - The members are simply the parameters
  - (Defaults don’t muck up readability like in function prototypes!)

So Let’s Make A Message

```
module some.module;

struct ImportantMessage
{
   int      param;
}
```

Automated Message Gathering

```d
// As seen earlier:
alias Members = __traits( allMembers, Params );

// Now let's get everything for an entire module!
alias ModuleMembers = __traits( allMembers, Module!"some.module" );


pragma( msg, ModuleMembers.stringof );

// tuple("ImportantMessage")
```

But Very Soon We Have A Problem...

```
module some.module;

struct ImportantMessage
{
   int      param;
}

// Ignore this struct! We use it elsewhere, but it's not a message
struct SomeImportantStruct
{
   string   arcaneknowledge;
}
```

User Defined Attributes!

```
// Compile-time expressions that can be attached to a declaration. These
// attributes can then be queried, extracted, and manipulated at compile time.

// Just add a @ in front of a symbol before your definition!

@SomeType  struct SomeStruct {}
@( 42.0f ) struct SomeStruct {}
@SomeFunc  struct SomeStruct {}
```

Message Type UDAs

```
struct ServerMessage { }

struct ClientMessage { }
```

Tagging Up Messages

```
@ServerMessage
struct ImportantMessage { bool bParam; }

@ClientMessage
struct AnotherImportantMessage { bool bParam; }

@ServerMessage @ClientMessage
struct Ping { }

@ServerMessage @ClientMessage
struct Pong { }
```

Protip: A UDA To Denote a UDA

```
struct UDA { }

@UDA struct ClientMessage { }
@UDA struct ServerMessage { }
```

Checking For Message Type

```
alias AllAttributes = __traits( getAttributes, MessageType );
```

Checking For Message Type

```
import std.traits : hasUDA;

enum PingIsClient = hasUDA!( Ping, ClientMessage );

enum IsClientMessage( T ) = hasUDA!( T, ClientMessage );
enum IsServerMessage( T ) = hasUDA!( T, ServerMessage );
```

Getting All Messages In One Line

```
import std.meta : Filter;

alias ClientMessages = Filter!( IsClientMessage, ModuleMembers );
alias ServerMessages = Filter!( IsServerMessage, ModuleMembers );
```

Protip: Unique IDs Via Hashing

```
ulong MakeHash( string val );

enum UniqueID( T ) = T.stringof.MakeHash; // Don't do this

enum UniqueID( T ) = fullyQualifiedName!( T ).MakeHash;

enum SomeStructID = UniqueID!SomeStruct; // HashOf( "some.module.SomeStruct" )
```

Code Generation With External Introspection

```
MessageCheck: switch( bytestream.ConsumeMessageID )
{
    static foreach( Message; FromClientMessages )
    {
        case UniqueID!Message:
            onReceived( bytestream.Deserialise!Message );
            break MessageCheck;
    }
    default:
        assert( false, "Invalid message found!" );
        break;
}
```

Code Generation With External Introspection

```
final void onReceive( ref const( Ping ) msg )
{
   Pong pong;
   sendMessage( pong );
}

final void onReceive( ref const( Pong ) msg )
{
   writeln( "Pong!" );
}
```

Eponymous Mixin Templates

Duplicating Behavior

```
final void onReceive( ref const( Ping ) msg );
final void onReceive( ref const( Pong ) msg );
```

Duplicating Behavior

```
class ClientServerCommon
{
   final void onReceive( ref const( Ping ) msg );
   final void onReceive( ref const( Pong ) msg );
}

class ClientCommon : ClientServerCommon { }

class EditorClient : ClientCommon { }
class DebugClient : ClientCommon { }
```

Duplicating Behavior

```
interface ClientServerCommon
{
    final void onReceive( ref const( Ping ) msg );
    final void onReceive( ref const( Pong ) msg );
}

class ClientCommon : ClientServerCommon, ClientCommon { }

class EditorClient : ClientServerCommon, ClientCommon { }
class DebugClient : ClientServerCommon, ClientCommon { }
```

Duplicating Behavior

```
mixin template PingPong()
{
   final void onReceive( ref Pong msg );
   final void onReceive( ref Ping msg );
   int SomeTrackingVal;
}

class Client
{
   mixin PingPong!();
}
```

Formalising Partial Functionality

Behavior rules:

- One per module
- Module name must be lower case of behavior name

Formalising Partial Functionality

```
module behaviors;

@UDA struct Behavior { }

mixin template Behaviors( Names... )
{
    static foreach( Name; Names )
    {
        mixin( "import behaviors." ~ Name.toLower ~ " : " ~ Name ~ ";" );
        mixin( "mixin behaviors." ~ Name.toLower ~ "." ~ Name ~ "!();" );
    }
}
```

Formalising Partial Functionality

```
module behaviors.pingpong;

@Behavior
mixin template PingPong()
{
   final void onReceive( ref Pong msg );
   final void onReceive( ref Ping msg );
}
```

Behaviors!

```
class Server
{
   // Behaviors are essentially partial classes at this point
   mixin Behaviors!( "PingPong",
                     "HandShake",
                     "Terminate" );
}

class EditorClient
{
   mixin Behaviors!( "PingPong" );
}
```

Specialising Behaviors

```
final void onReceive( ref Pong msg )
{
    static if( IsServer )
    {
        // Do stuff
    }
    else static if( IsClient )
    {
        // Do different stuff
    }
}
```

Specialising Behaviors

```
module behaviors.pingpong;

@Behavior
mixin template PingPong( Mode ThisMode )
{
   final void onReceive( ref Pong msg );
   final void onReceive( ref Ping msg );
}
```

Specialising Behaviors

```
class Client
{
   // Error! The behavior mixin will try to instantiate your mixed-in mixin
   mixin Behaviors!( "PingPong!( Mode.Client )",
                     "HandShake",
                     "Terminate" );
}
```

Eponymous Templates

```
template SomeTemplate( T )
{
   enum SomeTemplate = T.init;
}

T val = SomeTemplate!T;
```

Eponymous Templates

```
template SomeTemplate( T )
{
    T SomeTemplate( string SaySomething )
    {
        writeln( SaySomething );
        return T.init;
    }
}

T val = SomeTemplate!T( "Hurry up and get to the good stuff!" );
```

Protip: Reducing Template Bloat Redux

```
struct SomeDodgyStruct( BitfieldFrom!Params TheseParams )
{
    static if( TheseParams.ImportantParam )
    {
        int SomeImportantInt;
    }
    static if( TheseParams.AnotherImportantParam )
    {
        float SomeImportantFloat;
    }
}
```

Protip: Reducing Template Bloat Redux

```
enum IsParam( alias P ) = is( typeof( P ) == Params );

template SomeDodgyStruct( TheseParams... ) if( TheseParams.length >= 1 )
{
   alias ActualParams = Filter!( IsParam, TheseParams );
   alias ParamBits = BitfieldFrom!Params;

   alias SomeDodyStruct = SomeDodgyStruct!( ParamBits( [ ActualParams ] ) );
}
```

Protip: Reducing Template Bloat Redux

```
alias DodgyStruct = SomeStruct!( Params.ImportantParam,
                                 Params.AnotherImportantParam );

// Generates the same code thanks to the eponymous template redirect!
alias DodgyStruct = SomeStruct!( Params.AnotherImportantParam,
                                 Params.ImportantParam );
```

Eponymous Mixin Templates

```
template PingPong( T )
{
    @Behavior
    mixin template PingPong()
    {
        // Manual instantiation becomes PingPong!( Mode.Client )!();
    }
}
```

Eponymous Mixin Templates

```
template PingPong( T )
{
    @Behavior
    mixin template PingPong()
    {
        // ERROR! Undefined symbol T
        // This mixin has zero knowledge of its surrounding template
        T SomeValue;
    }
}
```

Protip: String Mixins

```
int SomeVal;
```

Protip: String Mixins

```
int SomeVal;

// The following code generates the above statement
mixin( "int SomeVal;" );
```

Protip: String Mixins

```
int SomeVal;

// The following code generates the above statement
enum PartOne = "int ";
enum PartTwo = "SomeVal;";
mixin( PartOne ~ PartTwo );
```

Protip: String Mixins

```
int SomeVal;

// The following code generates the above statement
string MakeSomeVal( T )()
{
   return T.stringof ~ " SomeVal;";
}
mixin( MakeSomeVal!int );
```

Protip: String Mixins

Whether it’s a template mixin or a string mixin, you are ***always*** required to generate legal D code.

Which helps significantly with maintenance.

Compared to macro solutions from other languages, this is clearly the correct way to go about things for large scale codebases with multiple maintainers over its lifespan.

Eponymous Mixin Templates

```
// What the code needs to look like at the end of the day

// Instantiated with PingPong!( Mode.Client )
template EponymousMixin( Params... )
{
   // A new mixin that is little more than a wrapper for another mixin
    mixin template EponymousMixin( )
    {
        mixin ActualMixinTemplate!( Params );
    }
}
```

Eponymous Mixin Templates

```
string GenerateMixin( string DesiredName, string MixinName )
{
   return = "template " ~ DesiredName ~ "( Params... )"
   ~ "{"
   ~ "  enum MixinString = \"mixin template " ~ DesiredName ~ "()"
   ~ "  {"
   ~ "    mixin " ~ MixinName ~ "!( AliasSeq!( \" ~ ToString!Params ~ \" ) );"
   ~ "  };"
   ~ "  mixin( MixinString );"
   ~ "}";
}
```

Eponymous Mixin Templates

```
module behaviors.pingpong;

@Behavior
mixin template PingPongImpl( Mode ThisMode )
{
  final void onReceive( ref Pong msg );
  final void onReceive( ref Ping msg );
}

mixin( GenerateMixin( "PingPong", "PingPongImpl" ) );
```

Eponymous Mixin Templates

```
class Client
{
   // And now this works just fine! Readbility maintained, minimal maintenance
   mixin Behaviors!( "PingPong!( Mode.Client )",
                     "HandShake",
                     "Terminate" );
}
```

Eponymous Mixin Templates

```
@Behavior
mixin template SendReceive( alias CanSendCheck, ReceiveMessages... );

class Client
{
   mixin Behaviors!( "SendReceive!( CanClientSend, FromServerMessages )" );
}

class Server
{
   mixin Behaviors!( "SendReceive!( CanServerSend, FromClientMessages )" );
}
```

Templated UDAs

In An Earlier Slide...

```
MessageCheck: switch( MessageID )
{
    static foreach( Message; ClientMessages )
    {
        case UniqueID!Message:
            onReceived( bytestream.Deserialise!Message );
            Break MessageCheck;
    }
    default:
        assert( false, "Invalid message found!" );
        break;
}
```

Object Serialisation

```
auto Deserialise( T )( ref byte[] bytestream )
{
   // How does the magic work?
}
```

Object Serialisation

```
auto Deserialise( T )( ref byte[] bytestream ) if( is( T == struct ) )
{
    T output;
    static foreach( Index, Member; T.tupleof )
    {
        output.tupleof[ Index ] = bytestream.Deserialise!( typeof( Member ) );
    }
    return output;
}
```

Object Serialisation

```
@UDA struct NoSerialise { }

struct SomeStruct
{
    int SerialiseThis;
    @NoSerialise float IgnoreThis = 42.0f;
    double AlsoSerialiseThis = 0.0f;
}
```

Object Serialisation

```
auto Deserialise( T )( ref byte[] bytestream ) if( is( T == struct ) )
{
    T output;
    static foreach( Index, Member; T.tupleof )
    {
        static if( !hasUDA!( Member, NoSerialise ) )
        {
            output.tupleof[ Index ] = bytestream.Deserialise!( typeof( Member ) );
        }
    }
    return output;
}
```

UDAs Revisited

```
@SomeType  struct SomeStruct {}
@( 42.0f ) struct SomeStruct {}
@SomeFunc  struct SomeStruct{}

struct SomeTemplate( T )
{
   T SomeValue;
}

@SomeTemplate!int struct SomeStruct {}
```

Serialisation Hooks With Tempalted UDAs

```
@UDA struct OnSerialise( alias Func )          { alias Call = Func; }
@UDA struct OnDeserialise( alias Func )        { alias Call = Func; }

@UDA struct OnPreSerialise( alias Func )       { alias Call = Func; }
@UDA struct OnPreDeserialise( alias Func )     { alias Call = Func; }
@UDA struct OnPostSerialise( alias Func )      { alias Call = Func; }
@UDA struct OnPostDeserialise( alias Func )    { alias Call = Func; }
```

Serialisation Hooks With Templated UDAs

```
struct Distance
{
   @OnSerialise!( x => x.sqrt )
   @OnDeserialise!( x => x * x )
   double value = 0.0;

   alias value this;
}
```

Serialisation Hooks With Templated UDAs

```
static foreach( Index, Member; T.tupleof )
{
    static if( hasUDA!( Member, OnDeserialise ) )
    {
        output.tupleof[ Index ] = getUDAs!( Member, OnDeserialise )[ 0 ]
                                  .Call( bytestream.Deserialise!( typeof( Member ) ) );
    }
    else
    {
        output.tupleof[ Index ] = bytestream.Deserialise!( typeof( Member ) );
    }
}
```

Serialisation Hooks With Templated UDAs

```
struct Distance
{
   @OnSerialise!( x => x.sqrt )
   @OnDeserialise!( x => x * x )
   double value = 0.0;
   alias value this;
}
```

Things To Try At Home

Bitfield Type And Size Specifications

```
@UDA struct Storage( T, ulong BitCount );

enum Params : string
{
    @Storage!( int, 4 )
    Important = "Adds an int to the struct",
    @Storage!( uint, 2 )
    AnotherImportant = "Adds a float to the struct",
}
```

Command Line Parser With UDAs

```
@DefaultArgParser( "file", "Input files", ArgIs.Repeating )
bool ParseInputFileParam( string value, int priorcallcount, ref ProgramParams
params );

@ArgParser( "o", "output", "Target file", 1.MinCalls, 1.MaxCalls )
bool ParseInputFileParam( string value, int priorcallcount, ref ProgramParams
params );

@ArgParser( "i", "iwad", "IWAD to use as a base for the provided input file",
1.MaxCalls )
bool ParseIWADParam( string value, int priorcallcount, ref ProgramParams
params );
```

UFCS, But Templates!

```
struct Symbols( S... )
{
   alias S this;
   template opDispatch( string symbol ); // Magic goes here!
}

alias TheseSymbols = Symbols!( int, float, string, bool );
pragma( msg, Symbols[ 1 ].stringof );

// Currently doesn't work!
alias UTESFilterAlias = TheseSymbols.Filter!( IsFloat );
```

Questions?