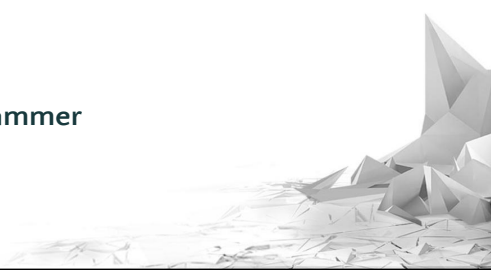
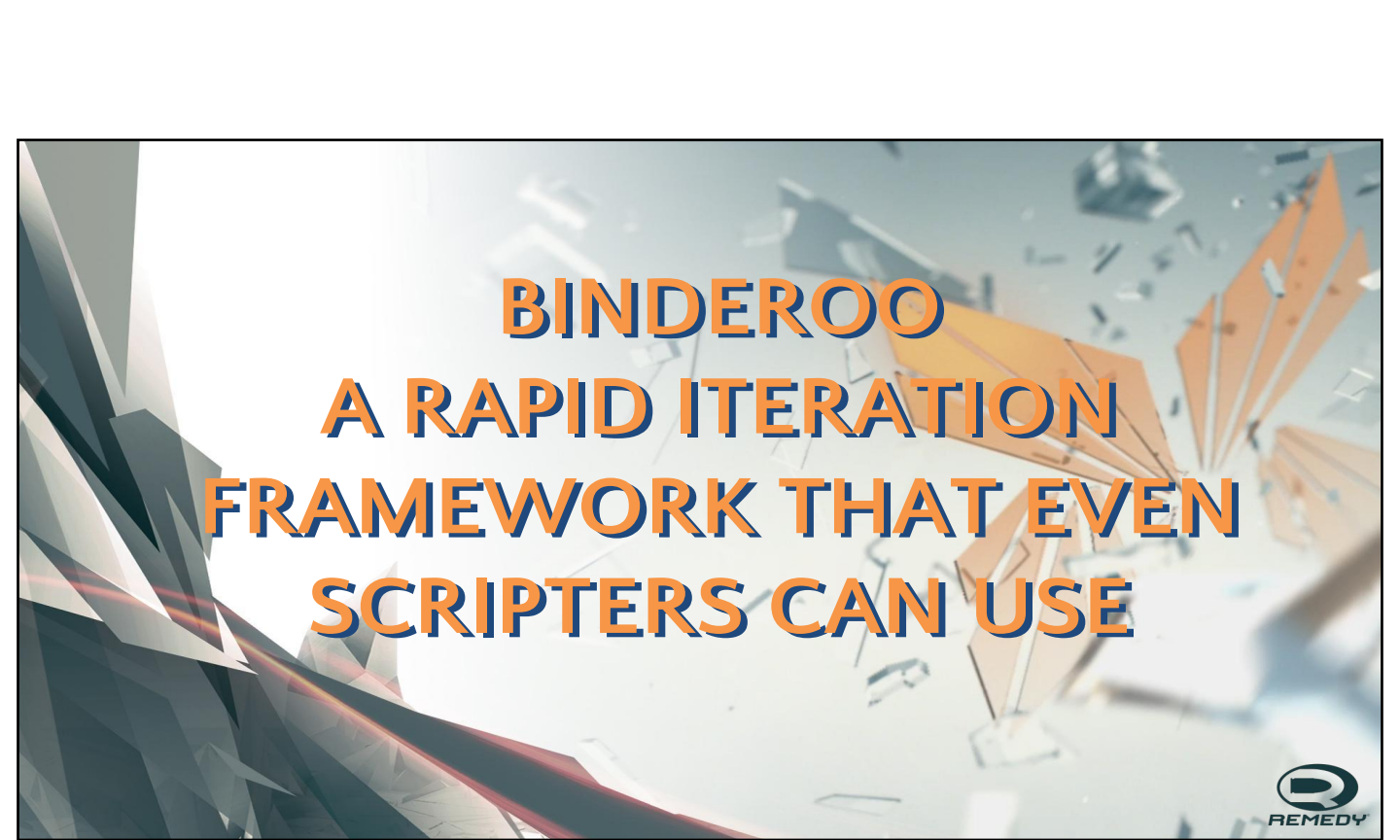




DConf 2017
Berlin, Germany
Ethan Watson, Senior Engine Programmer





BINDEROO A RAPID ITERATION FRAMEWORK THAT EVEN SCRIPTERS CAN USE

Before we get started, I'd like to point out that I'll take questions during the talk via this URL. You can go on there and ask a question, and you can even upvote questions. I'll check periodically throughout the talk and answer the most voted questions. In fact, there's one here right now:

[SWITCH]

[FORWARD]



Ethan Watson



14.5 Years

11 Platforms



So a bit about myself to begin with for those that don't know me. As the title slide gave away, I'm Ethan Watson.

[FORWARD] And I come from the fictional land of Australia.

[FORWARD] I've been in the industry for 14 and a half years, during which time I've shipped games on 11 different platforms. I've shipped a number of titles as both an engine programmer and a game programmer, some of which include the TY the Tasmanian Tiger games; Star Wars: The Force Unleashed; Game Room. I'm currently a senior engine programmer at Remedy Entertainment, and have most recently shipped Quantum Break.

[FORWARD]

Previously...

- Myself at DConf 2016, Manu at DConf 2013
- GDC Europe 2016
 - Very well received talk
- Reboot Develop Croatia 2017
 - Also well received



I'm no stranger here to DConf, and neither is the company I work for, Remedy Entertainment.

[FORWARD] I presented here last year about how we shipped Quantum Break with D Code, and Manu Evans presented here in 2013 about the initial work to support D.

[FORWARD] I've been a bit busy since then. I presented a talk at the Game Developer Conference Europe held in Cologne last year.

[FORWARD] The talk was very well received. Unfortunately, it was scheduled opposite John Romero though, who co-created the original Doom back in 1993, so the audience was a bit smaller than I would have liked.

[FORWARD] I also presented at Reboot Develop a couple of weeks ago in Croatia.

[FORWARD] The talk was also quite well received.

The interest in using D in game development is certainly increasing. Of course, getting out there and talking to other

industry professionals, I get to hear some of the reasons why people are hesitant to pick it up. There's one in particular that keeps on cropping up:

[FORWARD]

“D needs a large corporate sponsor”



This is something that keeps coming up. D needs a large corporate sponsor. Just the other week I had a guy from Io Interactive, who makes the Hitman games, say they're interested in what we're doing of D but skeptical because of the lack of a big corporate sponsor. Swift has Apple; Rust has Mozilla; Go has Google. But D doesn't have anyone that big. Even on a historical level, C and C++ had Bell; Java had Sun; C# has Microsoft; and so on.

I mentioned this to Andrei yesterday, and he's confident that this will be solved. That will certainly make getting more support in gaming for D easier.

[FORWARD]

Rapid Prototyping

- Programmers wanted a “scripting” system
- Code as data
- Quantum Break’s solution was prototype quality
 - Let’s do it right then...



Remedy’s usage of D, though, came about because of a desire to have rapid prototyping capabilities.

[FORWARD] Before I joined Remedy, the programmers decided that they wanted a system that enabled rapid prototyping. After some discussion, the decision was made to give D a shot but compiled into a DLL.

[FORWARD] This leads to a system in which code is not treated as code, but it is in fact treated as data. Anyone familiar with Unity and Unreal Engine is familiar with this paradigm - engine code is precompiled, your code sits in with the rest of your source data, and everyone’s happy.

[FORWARD] The solution we had in Quantum Break was of prototype quality. It was started by Manu Evans before I joined Remedy, and after he left Remedy I took over his work and took it to a functional state.

I never got much of a chance to take it beyond that functional prototype state, and as such there were many problems with the system that were screaming out for solutions.

[FORWARD] Now that Remedy is supporting multi-project development with a more traditional engine team, this means that we now have the chance to do it right.

[FORWARD]

Binderoo

C++ binding layer
Rapid iteration development

<https://github.com/Remedy-Entertainment/binderoo>



And we're doing it open-sourced to boot!

On our Github is the in-development version of all the back-end code used in this presentation. The version up there currently only supports Windows Desktop. We have some Xbox One support internally. Remedy announced last week that we're also branching out into Playstation 4 support. So while we'll get to it at some point before we release a PS4 game, if anyone out here is interested and would like to join in on getting this beast running on a PS4 hit me up after the presentation.

[FORWARD]

Binderoo Goals

- Perfect C++ binding
- Rapid iteration
- Minimal maintenance



There's three important goals that I'm aiming for with Binderoo:

[FORWARD] Perfect C++ binding is paramount. It's unreasonable to say to people "here's this great new thing! oh, also, you have to throw away all your old code." This is quite important in video games in the "AAA" space, since C++ is our de-facto standard language. Binderoo will serve as an interoperation layer between C++ and, well, any language you can compile in to a DLL. But D is the primary focus to begin with.

[FORWARD] Rapid iteration of code is also paramount. It's becoming unreasonable for people to stop program execution, recompile for a change, and load it back up with the size of data we use this day. We need to fill gigabytes of memory these days before games can start. Anything that removes that loading time from the equation will be invaluable going forward.

[FORWARD] And as a lesson learnt from doing it all before on Quantum Break, it needs to have minimal maintenance. Programmers are lazy by definition - it's literally our job to tell

a computer what to do so that we don't have to - so making a system a programmer doesn't have to think about will lead to it being more widely used.

Perfect C++ Binding

```
void someCPPFunction( )  
{  
    SomeSimpleClass aNewInstance;  
    printf( "%d\n", aNewInstance.aMemberInteger );  
}
```



Perfect C++ binding is a tricky one. D has quite a few features for binding to C++ But things in C++ land are rarely simple.

If you were to write code like this in C++, you'd expect that to just work with whatever the default constructed value of aMemberInteger is. Cool.

[FORWARD]

Perfect C++ Binding

```
void someEquivalentDFunction( )  
{  
    SomeSimpleClass aNewInstance;= new SomeSimpleClass;  
    writeln( aNewInstance.aMemberInteger.toString );  
}
```



But if you were to write an equivalent in D? You'd immediately get a null pointer error.

Of course, that's because classes are in fact reference types. You have to explicitly instantiate them because as far as your concerned, their storage is a pointer.

If we wanted our code to not throw a null pointer exception...

[FORWARD] we'd have to allocate a new instance of SomeSimpleClass. This is an important thing to keep in mind...

[FORWARD]

Perfect C++ Binding

```
extern( C++ ) class SomeOtherType
{
    final void SomeOperation();
    // Surprise! You also get a vtable pointer here
    SomeSimpleClass aNewInstance; // Surprise! This is a pointer
}
```



...because it gets trickier with more complicated types.

So you've embedded an instance of SomeSimpleClass in your object called SomeOtherType. This would be just peachy in C++ code.

[FORWARD] But surprise! Because a D class is a reference type, you've actually embedded a pointer in your class.

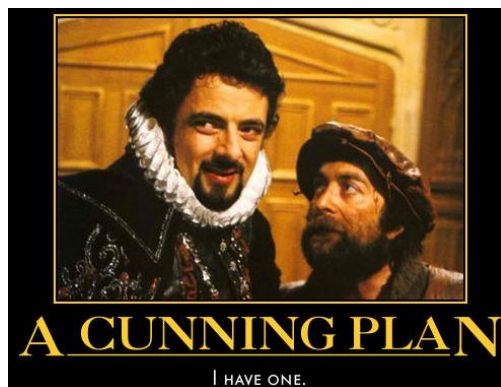
And there's one other surprise you'll find too:

[FORWARD] Regardless of if your class has virtual functions or not, a vtable entry will be added.

[FORWARD]

Value and Reference Types

- In C++
 - struct - value type
 - class - value type
- In D
 - struct - value type
 - class - reference type



REMEDY 

This seems to be just an inescapable piece of language design.

[FORWARD] In C++ there is essentially no difference between a struct and a class.

[FORWARD] Both a struct

[FORWARD] and a class are value types where storage location is defined by the user.

[FORWARD] D, however, has taken that modern approach

[FORWARD] where a struct is a value type, with storage location defined by the user

[FORWARD] and a class is a reference type with storage defined by the language itself. There's ways around that with classes, but they're all varying levels of broken and they tend to be fairly unintuitive. We're going for intuitive and "It Just Works™" here.

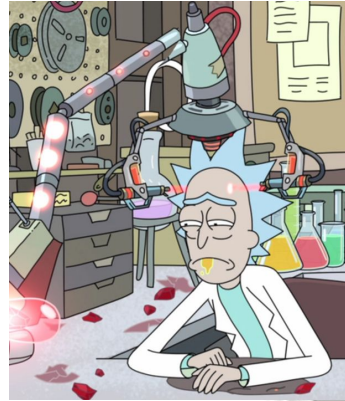
[FORWARD] Alright, so thinking of it that way, maybe the way to go is to make D structs do all the C++ heavy lifting, right?

They're the closest to C++ classes and structs after all.

[FORWARD]

A D Struct Can't Have...

- Default constructors
- Virtual methods
- Inheritance



REMEDY 

Weeeellllll, if it wasn't for the fact that D structs can't have
[FORWARD] default constructors,
[FORWARD] virtual methods,
[FORWARD] or inheritance, they'd be perfect.
[FORWARD] But don't despair. All is not lost.

Rapid Iteration

- Code compiled in dynamic library
 - No need to link executable every time
 - Also it allows hot reloading
- Can't hard link functions

- There's a common solution here...



We're making a rapid iteration system, after all!

[FORWARD] With code being treated as data, that means our D code is going to get compiled into dynamic libraries. This is for two practical reasons:

[FORWARD] If they were compiled to static libraries, we'd have to link all our code into our main executable every time; and

[FORWARD] being a dynamic library, that means we can unload it and replace it with a new version at runtime, thus supporting hot reloading of our code.

[FORWARD] Windows DLLs can be used in a static manner thanks to a static lib that does all the function linking for you on program initialisation. But we can't do that since we need to support hot reloading. We also have multiple platforms to consider at some point, and locking yourself into one inflexible solution is usually a bad idea there.

There's a common solution in here that solves our C++ interoperational woes *and* handles the function linking

automatically without user intervention. Which I call...

[FORWARD]

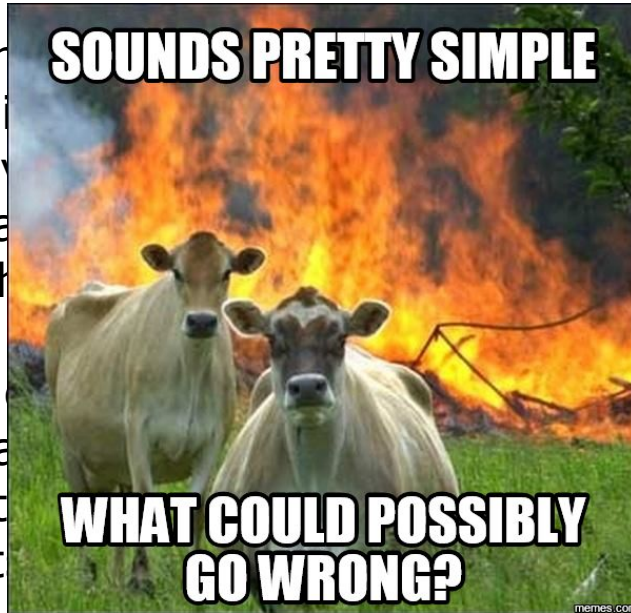
Let's implement C++ virtual tables from scratch inside D structs and also let's make sure non-virtual methods work and inheritance works and let's do it in a way that we can collect that information at compile time and use that to link functions on library load and support hot reloading through automated type serialisation and also while we're at it let's support version numbers on functions and objects to also solve Windows DLL Hell!



Let's implement C++ virtual tables from scratch inside D structs and also let's make sure non-virtual methods work and inheritance works and let's do it in a way that we can collect that information at compile time and use that to link functions on library load and support hot reloading through automated type serialisation and also while we're at it let's support version numbers on functions and objects to also solve Windows DLL Hell!

[FORWARD]

Let's in
scratch
sure non-
works a
collect th
use that
support h
serialisa
support
object



s from
t's make
nheritance
we can
time and
load and
ated type
t it let's
ons and
L Hell!

REMEDY 

[CHECK QUESTIONS HERE]

[FORWARD]

Binderoo's Three Components

- service
 - Handles compiling and reload notifications
- host
 - Loads D code, handles reloads
- client
 - Where you write your D code



To handle our rapid iteration requirements, Binderoo is split into three major components:

[FORWARD] There's the service

[FORWARD] Which is embedded into our content creation tool. It's responsible for monitoring our source data directories for code changes and recompiling the code into new DLLs. It also needs to communicate to

[FORWARD] a host

[FORWARD] which is the component you embed inside your game executable. It handles loading up those new DLLs, and the subsequent function linking and object lifetime concerns.

[FORWARD] The DLLs themselves need to include the binderoo client code

[FORWARD] and serves to communicate directly with the host. It is this client code that we're going to focus on for the next few slides.

Give It C Type Information

```
@CTypeName( "SomeSimpleClass", "core/src/SomeSimpleClass.h" )
```

```
struct SomeSimpleClass  
{  
}
```



So, you're writing a client DLL, and you want to know how to use your C++ code. There's a few things you'll need to do, but they're all quite simple.

Start off with a struct that is meant to represent the class...

[FORWARD] and apply the CTypeName User Defined Attribute to the entire object.

This particular UDA has several uses. When trying to bind functions, for example, we don't use the name of the object in D since that will not include all the C namespacing information we require - among other things.

[FORWARD] The first parameter, as such, is the name of your object as accessible from the global namespace.

[FORWARD] The second parameter is a bit more interesting - we give it the location of the matching C++ header. This comes in handy later.

[FORWARD]

Mark Up Bound Functions

```
struct SomeSimpleClass
{
    @BindVirtual( 1 ) void SomeMethod();
    @BindVirtual( 1 ) void DoThis( int anIntParam );
    @BindMethod( 1 ) void DoThat( float aFloatParam );
}
```



What's our C++ class without some functions?

[FORWARD] We tag our virtual methods with a BindVirtual UDA. It has a numerical parameter here. This is simply the version number that this function has been introduced in. It secretly takes a second parameter that defines the version that function was removed in.

[FORWARD] Non-virtual functions get tagged in the exact same way, except with the BindMethod UDA.

[FORWARD]

Generate Code To Make It Work

```
struct SomeSimpleClass
{
    mixin CPPStructBase!( BeginsVirtual.Yes );
    mixin( GenerateBindings!( typeof( this ) ) );

    int aMemberInteger = 0;
}
```



And finally, a bit of magic to make it all work. We have two mixins that do our workL

[FORWARD] A mixin template that will set up constructors and, if the BeginsVirtual parameter is defined as true, it will also put a pointer in the object to store the virtual function table in.

[FORWARD] The second mixin is a fair bit more interesting. Provide it with a string, and it will be inserted directly into the code and evaluated as if you had written the code yourself. In this particular case, the string is being provided to us by the GenerateBindings function.

[FORWARD]

GenerateBindings Internals

```
foreach( Member; __traits( allMembers, ThisType ) )  
{  
    static if( __traits( getOverloads, ThisType, Member ).length > 0 )  
    {  
        // Now we're dealing with functions  
    }  
}
```



How does GenerateBindings work? It's actually a conceptually simple piece of code.

We iterate over each member, and from there

[FORWARD] we use the getOverloads trait. This is the simplest way by far I've found to determine if I'm dealing with a function. If it's a function, the tuple length will be a minimum of 1.

[FORWARD]

Generate Bindings Internals

```
void DoThis( int anIntParam );
```

```
alias FnType = extern( C++ ) void function( SomeSimpleClass* pThis, int  
anIntParam );
```

```
alias FnType = extern( C++ ) ref int function( SomeSimpleClass* pThis,  
int anIntParam ); // ERROR!
```



For each function we encounter, we need to do a few things.

The first is to define a function pointer type. For each of our encountered functions, we need to rewrite the parameters of the function to support the thiscall function calling convention. Essentially, we need to make the first parameter of each function a this pointer.

[FORWARD] And then build up a new type based on all that. For a C++ calling convention of a method, the this parameter comes first. So before we insert all our previous parameters, we just stick our own this pointer in there.

[FORWARD] Of course, when dealing with typical game engine C++ functions, I come across one of my biggest bugbears with declaring function pointer types - the inability to specify ref return types.

```
module functionrewriter;
```

```
template FnRewritePtr( alias fn )  
{  
    // Do some analysis stuff to generate...  
    static extern( C++ ) ref int prototype( SomeSimpleClass* pThis,  
                                             ref SomeOtherClass param1,  
                                             ref YetAnotherClass param2 );  
  
    alias FnRewritePtr = typeof( &prototype );  
}
```



The way around this can be solved with templates. The template takes an alias to a function, we do some analysis of the function, and we generate a new function prototype that we then take the typeof for our eponymous template. Unfortunately, this is not problem solved.

[FORWARD] Because our template will live in a module somewhere.

[FORWARD] And it will deal with types that don't live inside this module. This has only become a problem with the template visibility changes over the last year or so.

[FORWARD]

```
alias ParamType = ParameterTypeTuple!fn;
foreach( Type; ParamTypes )
{
    alias UnqualType = Unqualified!( Type );
    static if( IsUserType!( UnqualType ) )
    {
        importString ~= "import " ~ ModuleName!( UnqualType ) ~ ";";
    }
}
mixin( importString );
```

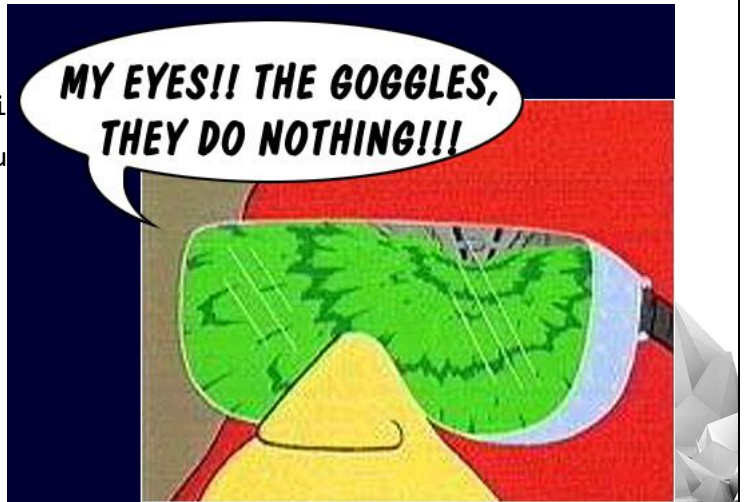


And it leads to some truly awful code. Essentially, I have to analyse each and every single type, from the return type to each parameter type, of the function that we're rewriting. If it's a user type, then I need to get the module name and

[FORWARD] mix it in inside my template. Otherwise, it errors out complaining that this type is not visible to the module the Function Pointer template lives in.

[FORWARD]

```
alias ParamType = ParameterTypeTuple!fn;  
foreach( Type; ParamTypes )  
{  
  alias UnqualType = Unqualifi  
  static if( IsUserType!( Unqu  
  {  
    importString ~= "import "  
  }  
}  
mixin( importString );
```



REMEDY 

Any kind of template that deals with types in a non-trivial manner now needs to do this. Reducing the functionality of templates so that I need to do this makes me not want to use templates.

Buuuuut at least I have something that works. So I can continue.

[FORWARD]

GenerateBindings Internals

```
// Into a vtable you go!  
@BindRawImport( ... ) FnType function0;  
  
@BindVirtual( 1 ) void DoThis( int anIntParam )  
{  
    _vtable.function0( &this, anIntParam );  
}
```



The second thing - now that we've got a function pointer type, we need to stick an instance of it somewhere

[FORWARD] and mark it up with a new, secret UDA that our binding system will be looking for. This BindRawImport UDA will contain all the information required for it to look up the correct function pointer.

[FORWARD] The final thing - well, we need to provide a definition for the DoThis function. By silently putting code in that wraps the D function call in to the C++ function pointer, this has the end result of making the object precisely as usable as the native C++ interoperation support. No special treatment is required, simply get an instance of your object and use it like any other instance of an object.

Versioning also means that we can support loading new D code in to an old engine version, or vice versa. We can build up a version of the vtable according to version number matching. Which is great if your interfaces are unstable.

[FORWARD]

GenerateBindings Internals

```
@BindMethod( 1 ) void DoThat( float aFloatParam )  
{  
    _mtable.function0( &this, aFloatParam );  
}
```



Methods are also handled in essentially the same manner - the difference being that they get a separate table so that virtual tables aren't interfered with.

Derived Types Are Easy

```
struct SomeDerivedClass
{
    mixin CPPStructInherits!( SomeSimpleClass );
    mixin( GenerateBindings!( typeof( this ) ) );

    @BindVirtual void aNewMethod( int aNewParam );
}
```



Derived types are also quite easy to implement. We just need a slightly different mixin template

[FORWARD] specifically, the CPPStructInherits template. It must take a parameter of the base type. The CPPStructInherits template is responsible for iterating through the inheritance chain and ensuring this new type contains a complete representation of the object.

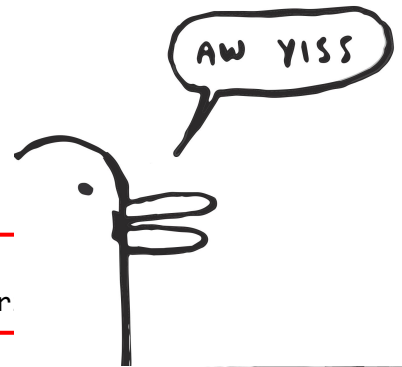
There is, of course, one more derived type we need to care about:

[FORWARD]

Also, One Last Derived Type

```
struct ANewDTypeDerivedFromACPPType
{
    mixin DStructInherits!( SomeDerivedClass );
    mixin( GenerateBindings!( typeof( this ) ) );

    @BindVirtual void aNewMethod( int aNewParam )
    { writeln( "I'm in D! With ", aNewParam.to!str
}
}
```



REMEDY 

Exposing your code from C++ is all well and dandy. But we're writing a rapid iteration system. That means that we would want to write new code and have it Just Work™.

[FORWARD] Enter the DStructInherits mixin. It operates in essentially the same manner as CPPStructInherits, with one important distinction:

[FORWARD] each BindVirtual method encountered is now matched to previous declarations in the inheritance chain and inserted into the vtable in place of the C++ function pointer.

With perfect binary matching and authoring our own virtual tables, this means we can sit on top of C++ code, write D code,

[FORWARD] and get on with enjoying life.

[FORWARD]

Exposing C++ Code

- This can be automated
- Compile your D definitions to a DLL
- `binderoo_util -gA`



To tie it all together though, you need to expose all your C++ functions. The system we had in place for Quantum Break was a sisyphian task of doing all that work by hand.

[FORWARD] But we're in the future. This can be automated. I spend an awful lot of time generating D code, which gave me the idea - I should be able to generate C++ code as well, right?

The process is a fair bit more manual than the D code, but with a few simple steps you can automate the binding process too.

[FORWARD] The first step is to compile your D definitions into a DLL. This is the first part of the project build chain in our own projects internally.

[FORWARD] As a post-build process, invoke a program I distribute with binderoo called `binderoo_util` and provide the `-gA` parameter. This util will load up your generated DLL and ask it to generate some C++ bindings.

[FORWARD]

```
File Edit View Visual D Project Build Debug Team Tools Test Analyze Window Help Full Screen
Miscellaneous Files (Global Scope)
15
16 #include "binderoo/defs.h"
17 #include "binderoo/exports.h"
18 //-----
19
20 #include "coregame/src/EntityComponent.h"
21
22 static const int iExportVersion_coregame_EntityComponentState = 1;
23 static size_t numExportedMethods_coregame_EntityComponentState = 61;
24 static void constructor_coregame_EntityComponentState( coregame::EntityComponentState* pObj ) { new( pObj ) coregame::EntityComponentState; }
25 static void destructor_coregame_EntityComponentState( coregame::EntityComponentState* pObj ) { pObj->~EntityComponentState(); }
26 static void** getVTable_coregame_EntityComponentState() { coregame::EntityComponentState thisInstance; return *(void***)&thisInstance; }
27 static void** vtableOf_coregame_EntityComponentState = getVTable_coregame_EntityComponentState();
28 static binderoo::ExportedMethod exportedMethods_coregame_EntityComponentState[] =
29 {
30     binderoo::ExportedMethod( "coregame::EntityComponentState::EntityComponentState", "void()", ( void(*) ( coregame::EntityComponentState* ) )&constructor_coregame_EntityCom
31     binderoo::ExportedMethod( "coregame::EntityComponentState::~EntityComponentState", "void()", ( void(*) ( coregame::EntityComponentState* ) )&destructor_coregame_EntityCom
32     binderoo::ExportedMethod( "coregame::EntityComponentState::addChild", "void(r::ComponentStateBase*)", ( void( coregame::EntityComponentState::*( r::ComponentStateBase* ) )
33     binderoo::ExportedMethod( "coregame::EntityComponentState::isHidden", "bool(unsigned int) const", vtableOf_coregame_EntityComponentState[ 1 ] ),
34     binderoo::ExportedMethod( "coregame::EntityComponentState::removeChild", "void(r::ComponentStateBase*)", ( void( coregame::EntityComponentState::*( r::ComponentStateBase*
35     binderoo::ExportedMethod( "coregame::EntityComponentState::isHidden", "bool() const", vtableOf_coregame_EntityComponentState[ 2 ] ),
36     binderoo::ExportedMethod( "coregame::EntityComponentState::getChildByOSPType", "r::ComponentStateBase*( unsigned int )", ( r::ComponentStateBase*( coregame::EntityComponent
37     binderoo::ExportedMethod( "coregame::EntityComponentState::hide", "void( bool, unsigned int )", vtableOf_coregame_EntityComponentState[ 3 ] ),
38     binderoo::ExportedMethod( "coregame::EntityComponentState::getHideReason", "unsigned int() const", vtableOf_coregame_EntityComponentState[ 4 ] ),
39     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( asset::OSP_PropertyContainerBuilder8, bool )", vtableOf_coregame_Entity
40     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_SimpleKeyValueRead8, bool )", vtableOf_coregame_EntityComponent
41     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_SimpleKeyValueWrite8, bool )", vtableOf_coregame_EntityComponent
42     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_GetJSONSchema8, bool )", vtableOf_coregame_EntityComponentState[
43     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_ValueExtractor8, bool )", vtableOf_coregame_EntityComponentState
44     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_ResourceFetcher8, bool )", vtableOf_coregame_EntityComponentState
45     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_PointerFixer8, bool )", vtableOf_coregame_EntityComponentState[
46     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_PointerGatherer8, bool )", vtableOf_coregame_EntityComponentState[
47     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_MemberCounter8, bool )", vtableOf_coregame_EntityComponentState[
48     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::ObjectStreamProcessor_CompareSave8, bool )", vtableOf_coregame_Entit
49     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::OSP_WriteJSON8, bool )", vtableOf_coregame_EntityComponentState[ 15
50     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::ObjectStreamProcessor_WriteSave8, bool )", vtableOf_coregame_EntityC
51     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::ObjectStreamProcessor_WriteNoVersions8, bool )", vtableOf_coregame
52     binderoo::ExportedMethod( "coregame::EntityComponentState::virtualProcessVariables", "unsigned int( r::ObjectStreamProcessor_Write8, bool )", vtableOf_coregame_EntityComp

```

And generate it will. There will be a massive amount of code generated, all of it taking the heavy lifting of doing the binding away from you.

[FORWARD]

Exposing C++ Code

- This can be automated
- Compile your D definitions to a DLL
- `binderoo_util -gA`
- Add output to your C++/host project
- Compile C++ project, run
- (Future work - parse .h files automatically)



But at the very least, the next few steps should be pretty obvious:

[FORWARD] Add the output to your C++ project,

[FORWARD] and then compile your C++ project that embeds the binderoo host and run it. Simple. If everything has been set up correctly, you'll now have D code running in your project that itself is operating on top of your C++ code.

[FORWARD] There's a major feature I have plans for in the future, though - the ability to parse C++ header files and generate both the D representation and the C++ function exposure automatically. But that's a massive undertaking and is a bit off in the future.

[FORWARD]

“I want to just write code”



At Reboot Develop a few weeks back, I had a few good conversations Jonathan Blow. Video game fans would know his games Braid and The Witness. Language nerds would know him for working on his own programming language, Jai.

There's actually quite a fair bit of common ground to talk about. We've both had enough of C++ and want a better way for example. And Jai's coming along well. It's got three things I use quite a bit already - compile time code evaluation, compile time code generation, and complete introspection.

His philosophy as a programmer can essentially be summed up with that sentence up on the screen - "I just want to write code". This is something I can understand. Every time I have to battle with a language to get it to do what I want to do, it's frustrating.

[FORWARD]

Battling Compile Time Code

- Type evaluation order and finalisation
 - `__traits(allMembers)`
 - Template parameters
- Template engine is slow
 - Rewrote it to use CTFE and mixin code generation
- Also compile time debugging is awful



To get Binderoo to this point, I've had to battle D's compile time functionality.

[FORWARD] The bindings i just explained are actually a rewrite of the original form. I had it down to one mixin originally, but thanks to evaluation order I've had to split it out and be explicit about whether I'm starting a virtual function table.

[FORWARD] Invoking `__traits allmembers` will in fact "finalise" a class so to speak. I want to iterate over my type to see if it has virtuals so that I know whether to insert a vtable pointer. But to iterate over that type, I have to finalise the type making it impossible to insert that vtable pointer.

[FORWARD] Supplying a type to a template parameter at any point also "finalises" it, thus meaning further mixins won't work.

[FORWARD] On top of that, I've been working with Stefan providing him sample code using Binderoo. And as a result, I found out that the template engine is slow. I wrote my original implementation of Binderoo using templates in order to be

readable by humans.

[FORWARD] But to get performance out of it, after Stefan's recommendations I've had to rewrite it to not rely on templates but to generate string mixins to get the same functionality. It's now harder for a random person to read and understand the code, which I'm not too pleased with, but it does give performance wins.

[FORWARD] It'd also be remiss of me not to mention that compile time debugging is rather awful at the moment, especially when it comes to generated code with mixins. Pragma msg is my only tool here, I spit out the entire code I generated and then try to match up the problems with the generated code manually. This is a known problem, but it's worth mentioning.

[FORWARD]

DIPs I Need To Write

- rvalue reference parameters



I also need to write some D Improvement Proposals to help facilitate both perfect C++ binding, and allowing programmers to just write code without having to worry too much about implementation details.

[FORWARD] The first one I need to write is rvalue reference parameters. This is something we've heavily relied on in the games industry for a very long time.

```
extern( C++ ) Matrix create( ref const( Vector3 ) va11,  
                             ref const( Vector3 ) va12,  
                             ref const( Vector3 ) va13,  
                             ref const( Vector3 ) va14 );
```

```
Matrix mat = create( Vector3( 1, 0, 0 ),  
                    Vector3( 0, 1, 0 ),  
                    Vector3( 0, 0, 1 ),  
                    Vector3( 0, 0, 0 ) );
```



For example. A large chunk of our math is multi-dimensional, be it 3-dimensional vectors or 3x4/4x4 dimensional matrices. Copying those values to the stack just to call a function is quite inefficient, so we pass our values around by reference where possible.

[FORWARD] This also leads to shortcuts, where people will create new objects inline to call common function. They never expect to use them again, so storing them in an explicitly declared variable is entirely pointless. With something like a matrix create function, we could do the old "create a wrapper function that takes arguments by value and wrap into the base function" but for multiple ref parameters that becomes a permutation problem. Getting this to be something the compiler handles is much nicer for the end user.

[FORWARD]

DIPs I Need To Write

- rvalue reference parameters
- struct default constructors



Further on top of that though

[FORWARD] is that we need default constructors for structs.

[FORWARD]

```
int[] generateRandomValues( size_t count)
{
    CPPMutex aMutex = CPPMutex();
    int[] someArray;
    void doSomething( int val )
    {
        aMutex.lock(); someArray ~= randomInt() ^ val; aMutex.unlock();
    }
    Threading.parallelForEach( &doSomething, iota( 0, count ) );
    return someArray;
}
```



Our programmers expect to be able to instantiate their objects on the stack. This particular example is an easy-to-break case without constructors - a mutex that binds to our C++ code. On Windows, this C++ code will call InitializeCriticalSection in the Windows API on construction.

[FORWARD] We have a kind-of-awful workaround for this - the static brackets operator will call the C++ constructors for bound objects. This works when you write code like this dodgy example that no one in their right mind would ever write in the real world where it puts a mutex on the stack.

[FORWARD]

```
extern( C++ ) struct SomeCPPObject
{
    int foo;
    int bar;
    CPPMutex mutex;
    int reallyAMutexRightThereAreYouSerious;
    int yesTotallyItsQuiteNormal;
    int andItNeedsUniqueConstructionWithEachInstance;
}
```



However, this really becomes a problem when embedding types in other types. Perfect C++ support means that we need to be able to instantiate objects anywhere correctly. And things like

[FORWARD] embedding a mutex in another object need special consideration when struct default constructors don't exist. There is a few instances of this in Remedy's code where an object has a mutex sitting in there, and you can't post-blit a mutex and expect it to just work.

[FORWARD]

DIPs I Need To Write

- rvalue reference parameters
- struct default constructors
- Not exactly a DIP, but...
 - core.simd



[FORWARD] Also, it's not exactly a DIP, but

[FORWARD] core.simd is effectively unusable by LDC - or any platform that is not an x86 platform - which means I need to write an alternate implementation for each compiler or platform. Our engine uses SIMD datatypes quite a bit. Manu's tried to solve this with his std.simd module he was working on, but as it stands right now there's not a SIMD solution anyone can just pick up and use and expect to just write portable code with.

[CHECK QUESTIONS HERE]

Scripting

- Programmers wanted “scripting”...
- How about everyone else?
- Why have a script VM when you have code as data?



Back at the start of this presentation, I mentioned

[FORWARD] that our usage of D came about as a result of wanting “scripting” or rapid iteration capabilities.

[FORWARD] I started thinking - why limit this to just the programming team?

[FORWARD] Just about every other team involved in gameplay requires scripting capabilities of some kind. Why go to all the trouble of running a script VM for some language when we have code that is treated as data?

Old Script

- Been around since Max Payne
- Message based, string lookups in code
- No debugger
- Lacks modern features - like loops



We already had a scripting solution in place.

[FORWARD] It's been around since Max Payne. So it's certainly battle tested. There are, however, many things that aren't ideal with it

[FORWARD] Executing code-provided functions is message based. One function receives the message, and then works out which bit of code to execute by if-else-ing string comparisons.

[FORWARD] A big problem with it is that there's no debugger. It's not an insurmountable problem, but it certainly stacks the chips up against it

But the big red mark against it

[FORWARD] is that it lacks modern features - like loops.

[FORWARD]

Old

- Bec
- Me
- No
- Lac



REMEDY 

We're totally in the future now though. If we want the old scripting language to be usable, we'd have to invest quite a bit of effort into getting it all up to modern standards.

[FORWARD]

Replace It With D!

- Level team keen for a real programming language
 - Several have CompSci degrees
 - Or are otherwise familiar with programming
- Also keen to try multithreading



After much theorising that it would be possible, we're finally trying out replacing the scripting system with D.

[FORWARD] Our level team is even keen to be let loose with a real programming language. Although saying "let loose" is a bit misleading.

[FORWARD] Several of the team have computer science degrees. They already know how to program. And in fact, the hacks they put into the scripts previously to get around loops is something no one should ever have to see.

[FORWARD] Even if they don't have a degree, many already know how to program in a real language.

Our level team these days is highly skilled, and thus the old assumption of not giving them too much rope to hang themselves with in a scripting language is just plain wrong these days.

[FORWARD] After discussing with the team as well, they're also keen to try multithreading. We've been replacing our old, monolithic game objects with component-based game objects.

Each of these components is being designed to run asynchronously in a task system. Rather than have a monolithic script VM execute during some execution phase, the D scripting code will just be function calls during any ordinary task in our system.

After asking around, I'm not aware of any other game studio giving multithreaded capabilities to scripters. This solution could very well end up being a litmus test for the industry as a whole.

[FORWARD]

Game Dev Terminology

- Component
 - Self-contained object wrapping up singular functionality
 - Executes as a task
- Blueprint
 - Collection of components defining an entire object



Of course, I'm talking to a bunch of not-game-developers here, so I'll clear up a few terms I'm about to use before I get to them.

[FORWARD] We've been slowly moving over to what is known as a "component" model. This has been around in the games industry for many years. I first used components in 2004 for example.

[FORWARD] And they're essentially just a self-contained object wrapping up a single piece of functionality. A car engine, for example, would be one component. A wheel would be another component.

[FORWARD] More specifically for our system, they execute as tasks. This essentially means arbitrary execution on any thread in a job system. No assumptions can be made about your environment, you absolutely need to work in a self contained manner. This also opens up the door to distributed execution, but we're not too concerned about that right now.

[FORWARD] We also have this concept of a blueprint in our engine.

[FORWARD] A blueprint simply defines a collection of components that represent an entire object. That car example I mentioned would be represented by a blueprint that pulls in engines, wheels, a chassis, and all those other things. It also defines the data relationship between those components so that we can schedule component execution in our task system correctly.

[FORWARD]

Scripting Environment

- Keep it familiar to old environment
 - Hierarchical level structure
 - Independently scriptable objects
 - Globally scriptable objects



Of course, to serve as a replacement for our scripting language, we kind of need to keep the scripters away from all that.

[FORWARD] The environment needs to seem familiar to users of the previous scripting system. That doesn't mean the language needs to be similar, just the constructs that they work in.

[FORWARD] For example, all objects in our levels live inside a hierarchy in the old system and can be accessed by name. This one's fairly critical for their usage.

[FORWARD] Each one of these objects can also be independently scriptable, regardless of their blueprint type. So essentially we need unique logic per instance of an object.

[FORWARD] There's also a concept of globally scriptable objects - these are essentially a functional approach to scripting, you write a function that takes an object as input and you perform the operations required on it. This is usually used for effect spawning and the like.

World Interface

- Generate D code from data
 - Blueprints become one object



This does mean, however, that we need to give them their own self-contained environment to play in. And thanks to trickery with D and function execution through `binderoo_util`,

[FORWARD] we generate quite a bit of D code from source data as well as other pieces of data we have.

[FORWARD] Our blueprint definitions live in a JSON file. In a Unity-esque environment, you'd have to resolve the components you want in an object manually, either by calling a function or storing a pointer in a variable. Rather than make people look up these components on an object every time they want to do something, we can go one better - and create a single object representing a single blueprint.

[FORWARD]

```
struct TransformComponentState
{
    @ComponentScriptFunction
    @BindMethod( 1 ) Vector  getWorldPosition() const;

    @ComponentScriptFunction
    @BindMethod( 1 ) void setWorldPosition( ref const( Vector ) vPos );

    @ComponentScriptFunction
    @BindMethod( 1 ) Quaternion getWorldRotation() const;
```



Our components can be augmented with a new user defined attribute,

[FORWARD] ComponentScriptFunction. This instructs our code generator to only consider these functions when generating the scripting interface. And because us programmers are a nice people, these are functions we've written internally ensuring that they're safe to call in a threaded environment.

[FORWARD]


```
struct HierarchyTransform
{
    private TransformComponentState* m_pTransform;
    public Vector getWorldPosition( ) const
    {
        return m_pTransform.getWorldPosition( );
    }
    public void setWorldPosition( ref const( Vector ) vPos )
    {
        m_pTransform.setWorldPosition( vPos );
    }
}
```



The object itself simply comprises of pointers to the relevant components, and wrapper functions that call the component functions with the correct parameters. We ensure all those pointers are set correctly, and a scripter needs to do nothing more than write their logic and call those functions. Nice and easy.

[FORWARD]

World Interface

- Generate D code from data
 - Blueprints become one object
 - Level reflection



Alright, so the net effect is that we're auto-generating scripting interfaces from definitions that the programmers use themselves, and blueprint data that entities need to use anyway. Sweet. What's next?

[FORWARD] We need to reflect the level hierarchy. And again, because this is just an abstract representation in the editor, we can go ahead and generate more code.

```

struct Entity001
{
    HierarchyTransform m_data;
    alias m_data this;

    private Entity000* m_pParent;
    @property Parent() { return m_pParent; }

    Entity002 Keyframed_Mesh_Entity_000;
    Entity003 Trigger_Entity_000;
}

```



Each entity in our level gets its own unique structure created for it. The name of the type is unimportant, since interaction with these entities will be exclusively through the hierarchy.

[FORWARD] Thus, the concept of its parent, accessible by - strangely enough - the Parent property is required. That gets

[FORWARD] As is the names of the children entities in the hierarchy.

This is far from a complete representation by the way - we also have helpers to get the root object of the level; iterate over all children, or even iterate over children by type - notice by the naming conventions that we have one keyframed mesh; and one trigger as our children entities.

[FORWARD] The interesting bit here is the HierarchyTransform definition. We're embedding a copy of our blueprint representation and calling it m_data. Then we use D's *alias this* functionality. This essentially imports the scope of the object named in the alias this declaration into this object's scope.

[FORWARD]

```
struct Entity001
{
    @Event OnUpdate()
    {
        Vector vPos = Vector( 0, 2 * Time.Delta, 0 );
        vPos += Parent.getWorldPosition();
        setWorldPosition( vPos );
    }
}
```



This does result in the user being able to write code as if it was the blueprint type, so it's an example of alias this working as intended and making everyone's life easier. This is also a case of the per-instance logic here - each object has an OnUpdate function that can be defined independently.

[FORWARD]

World Interface

- Generate D code from data
 - Blueprints become one object
 - Level reflection
- Scripters get their own utility library
- Also get their own components



With all of these in place, the scripters have most of what they need.

[FORWARD] But now that they have a proper programming language and disk space, they also get something they didn't have in the old scripting system - their own utility library. Previously, any utility functions would need to be written by a programmer and exposed through the messaging system. If they can write code, they can import it and use it wherever they want. Nifty!

[FORWARD] They also get their own components. Unlike code components, these are far more limited in scope - and a lot more flexible for a level builder's needs.

[FORWARD]

```

@PrettyName( "Rotation Component" )
struct RotationComponent
{
    @Event OnUpdate();

    @Settable( 1, "Pitch speed/second" ) Degrees pitchSpeed = 0.deg;
    @Settable( 1, "Yaw speed/second" ) Degrees yawSpeed = 0.deg;
    @Settable( 1, "Roll speed/second" ) Degrees rollSpeed = 0.deg;
}

```



This here is an example of a component that gives an entity the ability to rotate autonomously on the spot. Our previous solution required level builders to set up a full keyframed animation container to do something this simple.

[FORWARD] By giving it a PrettyName, it shows up in the level editor properties panel

[FORWARD] By giving it an OnUpdate event, it will update during the normal world step.

[FORWARD] And by marking some variables as settable, it allows you to set those variables per-instance from the level editor properties panel.

As a component, this means that a new blueprint needs to be made. But once that's done, having an object rotate by itself in the level means it's as simple as dropping the blueprint in the level and setting its parameters. Simple!

One little tidbit worth pointing out in here actually:

[FORWARD]

```
Degrees pitchSpeed = 0.deg;
```

```
Radians pitchSpeed = 0.rad;
```

```
Radians sin( Radians angle );
```

```
Degrees sin( Degrees angle ) { return sin( angle.rad ).deg; }
```



Level builders love their degrees. But programmers know all our math functions take radians. What do we do? Do we make them use radians? Probably a bad idea, they'd consider radians unusable.

So I stumbled across this pattern that I've been calling a type mimic.

[FORWARD] This Degrees object is essentially a float. The float data in fact uses alias this. We do give it some restrictions though. Like no multiplying angles together, that's undefined mathematically.

[FORWARD] Also a second type - radians. This one's important...

[FORWARD] because you make all your front-facing functions take radians. Zero confusion this way, and the UFCS deg and rad functions make sure you're explicit with your intentions.

[FORWARD] We do however need to also write wrapper functions to make life easier for everyone. Multiple alias this would remove the need to maintain wrappers, but right now

that's how things need to work.

World Interface

- Generate D code from data
 - Blueprints become one object
 - Level reflection
- Scripters get their own utility library
- Also get their own components
- Binderoo handles compile and reloading



All of this gives level builders quite a bit of power to do their thing while operating within our threaded environment. A lot of the functionality for them is essentially autogenerated, which means programmer maintenance comes down to whatever a programmer needs to do to expose normal C++ functionality to D.

[FORWARD] And let's not forget - binderoo handles compiling and reloading of these scripts. They get the exact same rapid iteration abilities that programmers do. Nifty!

[FORWARD]

Memory Management

- You what mate?
- Users expect a garbage collected environment
- We're not introducing a system-blocking monolithic collect phase



Opening scripters to native code leads in to the inevitable talk about memory management. To which any higher-level programmer will be all like

[FORWARD] you what mate?

[FORWARD] Higher level programmers expect a garbage collected environment. Scripting languages like Python and Lua, or even languages like C# and Java, don't make the user worry about object lifetime management.

[FORWARD] But we're going towards a massively parallel task-based execution environment. We're not going to introduce a system-blocking monolithic garbage collection phase.

[FORWARD]

ARC Garbage Collection

- Collect all the things
- Time to modify the front end
 - Pointer acquire and release
 - New function calls
- Not done yet
 - Ye olde "Cyclic Reference" to be solved



This calls for automatic reference counting garbage collection! ARC operates in a manner we expect - lose all references, automatically clean up your object - with a slight overhead to increment and decrement reference counters.

[FORWARD] It needs to be seamless to the users, and it needs to collect everything - pointers, references, slices, everything.

[FORWARD] So we need to go into the D frontend.

[FORWARD] We need to identify everywhere that a pointer is acquired and released

[FORWARD] and we need to insert new code or function calls to manage it. So things like `gc_obtainref` and `gc_releaseref` for example.

[FORWARD] We're still working on it. It's not quite ready yet

[FORWARD] Thanks to the old cyclic reference problem. But it'll get there.

The big thing here, of course, is that it does require branching

DMD to do this solution. There's been a few examples this conference already of people branching off DMD or LDC, and even talk about -betterc, but I'd rather not maintain my own compiler branch.

[FORWARD]

OUTATIME



Talk here went into questions to wrap it all up.