Binderoo - Wait, What Do You Mean You Quit Your Job?

Ethan Watson
professional loafer

Alright Munich, let's kick this off. Back for my third DConf presentation. And today, I've got some important stuff to get through. So, let's get started. [FORWARD]

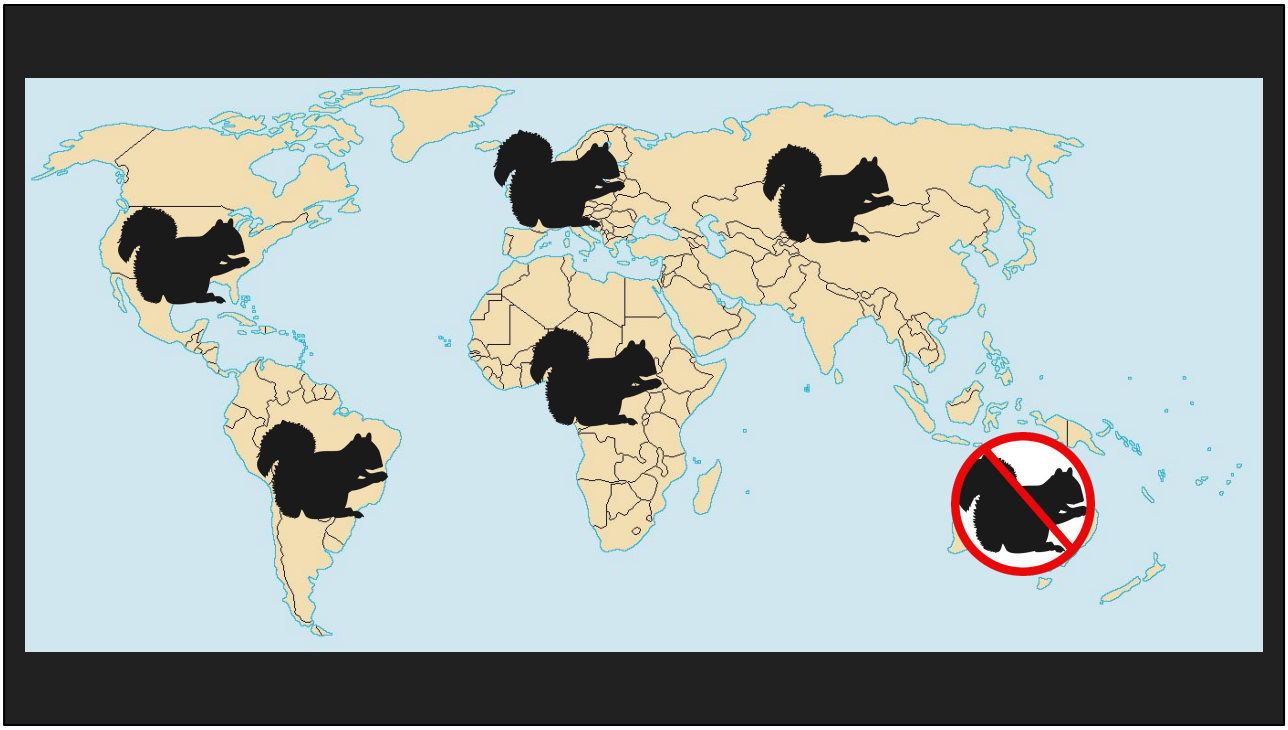
Squirrels

Squirrels. [FORWARD]

Yes, squirrels. Squirrels are quite important, and especially when it comes to the emerging completely legitimate scientific field of fictogeography. For instance…
[FORWARD]

You may have heard that the so-called land of Australia is a fictional place and entirely fabricated. I can prove this. See, in Australia [FORWARD]

they have no squirrels. And this seems like a glaring oversight in the creation of the Australian mythology. [FORWARD]



I mean, just think about it. Europe? [FORWARD]

Squirrels. Africa? Asia? [FORWARD]

Squirrels. North and South America? [FORWARD]

Yeah they have squirrels. But Australia? [FORWARD]

No squirrels. Clearly if such a place were to exist it would have squirrels just like everywhere else on earth. Whoever invented Australia really didn't think things through. [FORWARD]

Another place where a lot of fictogeographical research is currently focused on is the fictional country of Finland. And again, the squirrel comes in handy for proving that it is actually a made up place. [FORWARD]

In Finland, they have a fairly long and dark winter. Lots of snow. Or so they claim. In fact, their winters are supposedly so bad that only 15% of squirrels survive their first winter. Have you ever heard anything so preposterous? Squirrels are far more resilient than that. [FORWARD]

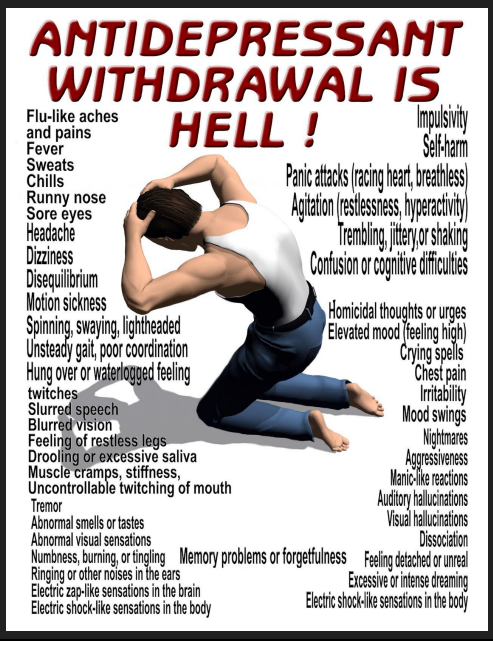
Of course, I am something of an expert on the matter. After all, [FORWARD]

I was born in Australia and [FORWARD]

have lived in Finland for almost six years so I have quite an extensive amount [FORWARD TO NEXT SLIDE] of experience to draw on here.

But what does all this even have to do with D?
I mean jeez you're presenting at DConf.
Not a flat earthers meeting.

Oh, uh, yeah, okay. So, um, fictogeography has nothing to do with D. You're right. But here's the thing. See, I quit my job in December last year. I've been on my own time since then. And one of the things I've spent my own time on recently… [FORWARD]



...is dealing with antidepressant withdrawal. And it is awful. The last couple of months of my life have essentially been a write-off. I still haven't gotten over it all, for example when drinking at the end of each day here at BeerConf alcohol is at least twice as effective as it used to be. But you're all here to hear about what D work I've been doing. So. You know how Binderoo is an open sourced project, right? [FOWARD]

Binderoo

Still being developed even though I quit my job!

https://github.com/GooberMan/binderoo
My development branch

https://github.com/Remedy-Entertainment/binderoo
Official home of Binderoo

Well, thanks to that I've been able to continue working on it. I've been out working in my own branch, and when it's all stable enough the pull request will be made and it'll be available on the main branch.

[TURN ON AUDIENCE Q&A FEATURE] And now that we're getting in to the presentation proper, I'll do the same trick I did last year. There's a URL up on top of these slides. Whether you're here or watching the live stream, enter that in to your browser of choice and ask Twitter-length questions. Anyone can also upvote questions. Anyone that's seen my talks previously knows that I usually have a lot to talk about, so this allows me to schedule questions appropriately to ensure my talk doesn't go longer than it should.

Alright, so let's get on with it. I've expanded Binderoo in a few ways. [FORWARD]

C/C++ Header Parsing

- Major bugbear at Remedy
  - Write a duplicate definition of the function
    - It's extra work and thus bad
  - Requires extra care for any API change
    - Publish new build
    - *THEN* deploy updated D definitions
- Automation is clearly the correct solution
- Also, I wanted it for my own reasons

The first thing that I want to talk about is C header parsing. [FORWARD]

This is something that came about because of a frustration that programmers had at Remedy. [FORWARD]

Having a function exposed to D essentially meant writing a duplicate definition of that function in a separate file to the declaration. [FORWARD]

Now, this is the exact kind of work that is required for binding a function to a scripting language. But programmers are lazy by definition - it's literally our job to tell a computer what to do so that we don't have to it manually. Thus, this being extra work, is a bad thing. [FORWARD]

Of course, if your C++ APIs are unstable then this means that extra care needs to be taken. Every time your interface changes, [FORWARD]

you need to publish a new build for everyone to download [FORWARD]

and then you would deploy the updated D definitions. [FORWARD]

It wasn't a very automated process for Quantum Break, and clearly complete automation is the correct solution. [FORWARD]

But the main reason I decided to implement C header parsing is because I want it for

a project I’m working on. There’s some C and C++ libraries that I want to use, and since rapid iteration in D is a great way to work I want to write D code with Binderoo. [FORWARD]

C/C++ Header Parsing

- It should work at compile time
  - Minimal template requirements
  - CTFE compatible function
- Need to write a parser from scratch then

I had one goal in particular that I wanted to meet when determining a solution. [FORWARD]

Namely that it should work at compile time, so that you could mixin a header if you so desire. [FORWARD]

I didn't want to rely on templates at all, since templates are a bit slow for the compiler. There'd be a convenience mixin, but that would be about it. [FORWARD]

So a CTFE compatible function would be required, which is really easy to do. Of course, since CTFE functions must be written in D and can't be called from, for example, a pre-built C processing library, [FORWARD]

I'd need to write my own header parser from scratch. [FORWARD]

This is in every way a bad idea, as I would soon discover. [FORWARD]

C/C++ Header Parsing At Compile Time

- Look for structs/classes
  - Generate D structs
- Look for functions
  - Generate @BindImport declarations
- Look for enums
  - Duplicate them
- Look for #defines that declare constants
  - Define them as enums

The work I needed to do seemed fairly simple to begin with [FORWARD]

Scan for structs and classes [FORWARD]

And generate D structs that binary match them. [FORWARD]

Any functions that we find [FORWARD]

Get @BindImport declarations that expose it to Binderoo.[FORWARD]

Enums are pretty easy in C++, they're all integral types [FORWARD]

So duplicate them whenever we find them. [FORWARD]

And finally, thanks to C preprocessor macros being used to declare constants [FORWARD]

We just identify them and declare them as enums. Simple, right? We shouldn't even need to write a complete parser and lexer since we're only going to be looking for very specific things in header files. Easy, right? [FORWARD]

C/C++ Header Parsing At Compile Time

- Need wxWidgets, so let's use that as a testbed
- `mixin ImportCHeader!( "wx/defs.h" );`
  - Several minutes to process
  - >10GB used
- Binderoo lets us easily run the function through a command line
  - Apply @BindExport to the function
  - binderoo_util -c <function_name> -p <parameter>

Now, the reason I'm going to all this effort [FORWARD]

Is because I want to use wxWidgets. There's an existing D binding for it, sure, but since we're working through Binderoo I'd need to re-bind the entire thing myself with Binderoo imports. So rather than write test code, let's just use wxWidgets as a test bed. [FORWARD]

So we start off simple. Defs.h contains a bunch of enumerations and other things that are required by other files, so that sounds like the perfect place to start. [FORWARD]

And it would take several minutes to process… [FORWARD]

And use over 10GB of RAM to compile. Wow. That's kinda nuts. [FORWARD]

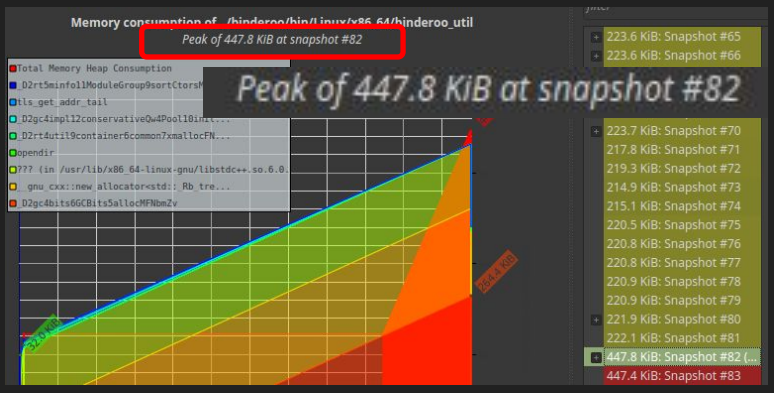
Of course, being something in Binderoo, there's a very easy way to test performance differences between the exact same functions running at compile time and at runtime. [FORWARD]

We can export that function with the BindExport UDA [FORWARD]

And then invoke it with the binderoo_util program. [FORWARD]

C/C++ Header Parsing At Compile Time

- valgrind --tool=massif binderoo_util …
- massif-visualizer



That's over twenty thousand times more memory efficient than running at compile time!

I need more information than just running it though. I want to know exactly why it's using that 10 gigabytes. Rather than being a classic game developer and writing my own tool for that, I'll accept that we're in a connected world and use a utility built for the exact purpose. [FORWARD]

So let's see what Valgrind has to say about it. We'll use the massif tool... [FORWARD]

And then open it in the visualizer and see what the problem is. [FORWARD]

And it turns out there is no problem. Peak memory usage is 447.8 kilobytes. [FORWARD]

Which is over twenty thousand times more memory efficient than running at compile time. Wow. [FORWARD]

No C/C++ Header Parsing At Compile Time :-(

- CTFE memory and performance is bad
- #include impossible to process
  - Can't do it one at a time and import resulting modules
  - #defines change thanks to #undef
  - #include order matters
    - import("wx/defs.h") is not nearly enough
    - Runtime code requires std.stdio
- I hear newCTFE addresses these problems…

Doing C header parsing at compile time was - only now - starting to sound like a bad idea. [FORWARD]

The memory usage at compile time as well as performance is prohibitively bad. I've been told that it's mostly because of the way the CTFE engine handles strings. But while I was writing the initial code, something else occured to me. [FORWARD]

The #include directive is essentially impossible to process. [FORWARD]

In simple cases, processing the headers one at a time and replacing #includes with D module imports works. But C/C++ is rarely simple. [FORWARD]

Common usage includes #defines that change thanks to undefining them and redefining them. But more importantly, [FORWARD]

#include order quite often matters in C headers, and each #include essentially needs to be reprocessed [FOWARD]

The import keyword isn't nearly enough in these situations [FORWARD]

And it also doesn't work for running the code after it's been compiled. You need to go through the stdio library to access files in runtime code, and those functions are not available at compile time. [FORWARD]

I hear newCTFE should address these problems, so the sooner that’s in main the better in my opinion. [FORWARD]

Also, C/C++ Is Actually Two Languages

- I don't mean C and C++ are two languages…
- The Preprocessor
- wxWidgets relies heavily on it (of course)
  - Symbol replacement
  - Mixin-style expansions
- Which means you need to write a Preprocessor evaluation engine

It also became clear to me around this time that my naive approach to doing this ignored one very important fact - that parsing C and C++ is actually parsing two languages [FORWARD]

And I don't mean the obvious that C and C++ are two divergent languages at this point. [FORWARD]

I mean the preprocessor. [FORWARD]

Getting accurate results out of my parsing code was proving difficult since wxWidgets relies heavily on preprocessor macros [FORWARD]

Both as a symbol replacement solution [FORWARD]

And more complicated mixin-style expansions. [FORWARD]

All this boils down to needing to write a preprocessor evaluation engine before you even think of scanning your files for functions and classes and enumerations and anything else you're interested in. [FORWARD]

Yeesh. Never look at the code for all of this. It’s a frankenstein’s monster at the moment. [FORWARD]

But It Is Getting There!

```
public import wx.event;
public import wx.list;
public import wx.cursor;
public import wx.font;
public import wx.colour;
public import wx.region;
public import wx.utils;
public import wx.intl;
```

I'm still working on getting all the little niggles out, but it's getting there. [FORWARD]

Any #include it runs across gets added to an import list. [FORWARD]

But It Is Getting There!

```
alias wxSortFuncFor_wxWindowList = extern( C++ ) int
function( const( wxWindow ), const( wxWindow ) );
```

It can understand typedefs and generate aliases for them. [FORWARD]

But It Is Getting There!

```
@CTypeName( "wxWindowVariant", "wx/window.h" )
enum wxWindowVariant : int
{
    wxWINDOW_VARIANT_NORMAL,
    wxWINDOW_VARIANT_SMALL,
    wxWINDOW_VARIANT_MINI,
    wxWINDOW_VARIANT_LARGE,
}
```

Enums get translated with their correct underlying type, and have the correct Binderoo attributes applied to them. [FORWARD]

But It Is Getting There!

```
@CTypeName( "wxWindowListNode", "wx/window.h" )
struct wxWindowListNode
{
    mixin CStructInherits!( wxNodeBase );
    mixin( GenerateBindings!( typeof( this ) ) );

    @BindVirtual( 1 )
    protected void DeleteData(  );
```

This particular object that is being bound is generated with the WX_DECLARE_LIST_3 macro, and it knows when to insert inheritance info and to preserve the order of the virtual function table. [FORWARD]

But It Is Getting There!

- Integrate it in to your build process
- Publish the resulting files with your build

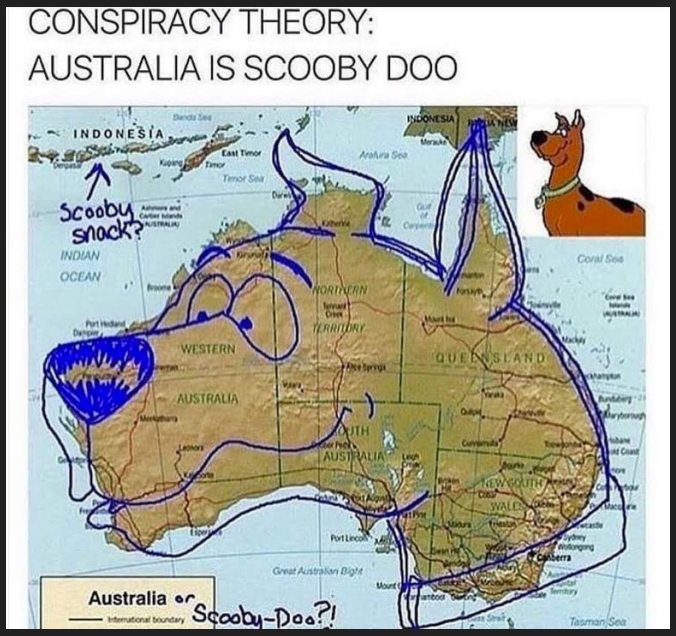
It still needs a bit more work to bind up everything perfectly, but it very definitely is going to save users a lot of time. [FORWARD]

Especially in terms of automation. Integrate the header parser in to your build process, [FORWARD]

And publish the resulting files with your build. And sweet. Now you've got an automated automatica-binding solution for as much of your codebase as you want.

[QUESTION TIME ON NEXT SLIDE]

Let’s check some questions...



[FORWARD]

.di Files

- Templates and CTFE is slow
- Precompiling Binderoo files into a library seems sensible
- Only problem with DMD's .di generation?
  - Mixin templates are not expanded
  - Neither is `mixin( CTFEFunction() );`
- End result - .di re-executes these
- But Binderoo knows everything it needs to generate these .di files itself...

One thing I've been wanting to do for a while is add .di files to the Binderoo chain. Why? [FORWARD]

Binderoo operates with a lot of template and CTFE code. And as we've seen in this presentation, that's not the fastest way to do things. [FORWARD]

So to facilitate fast rapid iteration, precompiling all Binderoo files - as in, the core functionality and all bindings generated - into a library that gets published with your build seems like a very sensible idea. [FORWARD]

Of course, there's one little problem with DMD's .di generation. [FORWARD]

Mixins templates are not expanded [FORWARD]

And CTFE results are also not expanded inside mixins. And Binderoo requires both of these things. [FORWARD]

The end result is that whenever you use a .di generated from Binderoo objects, these all get re-executed. And since we're trying to compile to a library to avoid that cost, that's far from ideal. [FORWARD]

Now, since Binderoo does a ton of reflection work, it actually knows everything it needs to generate these .di files itself, with every mixin and CTFE function it cares about expanded since it operates on objects after the compiler has done that work.

[FORWARD]

.di Files - By Binderoo

```
@CTypeName("TestObject", null, 15487458745312390138LU)
struct TestObject
{
    @BindVirtual(1, -1)  int someVirtual(float someFloat)
const;
    @BindVirtual(1, -1) ref int someOtherVirtual();
```

One artefact of doing this is that when I do a stringof on the UDAs that I find [FORWARD]

I get every variable expanded. Oh well. The important thing is that the UDAs are preserved… [FORWARD]

.di Files - By Binderoo

```
    // Variables
    @BindNoSerialise public void* _vtablePointer;
    public int iFoo;
    public float fFoo;
    public short sFoo;
    public double dFoo;
}
```

And things that are normally hidden for you, such as the insertion of the virtual function table pointer, are expanded correctly. [FORWARD]

.di Files - By Binderoo

```
@CTypeName("AnotherTestObject", null, 235548085149669629 5LU)
struct AnotherTestObject
{
    // Variables
    @InheritanceBase public TestObject base;
    alias base this;
```

As well as things like inheritance. This is all done with mixins when you write the code, but all gets expanded correctly with .di generation.

So that's pretty sweet. Some future work will be to expand the Binderoo Service module to handle pre-compiled libraries and automatically provide .di generation, but for now it's a bit of a manual process. [FORWARD]

Multiplatform Support

- Console support locked away behind the scenes
  - But there are other platforms…

- Binderoo now works on Linux!
- Now also compiles with LDC and Clang
  - Opens up PS4 and OSX support

Another thing I've done is expanded the multiplatform support. [FORWARD]

Console support is locked away behind the scenes. I actually don't have the Xbox One specific support files any more, that exists purely in Remedy's internal servers. [FORWARD]

But there are other platforms that I can support that everyone here would actually find useful. [FORWARD]

Binderoo now runs on Linux, for example. Sweet. I'm actually presenting here from Linux installed on a USB stick, my Windows install is still completely intact on this laptop so switching between to ensure multiplatform compatibility is quite simple. [FORWARD]

Fairly important as well is that Binderoo compiles with LDC and Clang. [FORWARD]

Which is a clear path forward for supporting PS4 and OSX. I don't imagine OSX will be hard at all from this point, but I currently lack the hardware to do the work and test it. [FORWARD]

Porting To Linux And Cleaning Up Old Problems

- Replace WinAPI calls with Posix calls. Simple!
- dlopen/dlclose/dlsym for library handling
- Linux doesn't have the Windows DLL attach messages
  - D runtime, subsequently, wasn't initialised
- Standardised with `binderoo_startup` function
- This reintroduced a Quantum Break problem…
  - DLL_THREAD_ATTACH before DRT initialised

Porting to Linux was actually quite easy. I've been a multi-platform engineer my entire professional career, so writing Binderoo in a manner that was conducive to cross-platform operations is just something I do from the start. [FORWARD]

And it really just came down to replacing the Windows API calls with Posix calls in the right locations. [FORWARD]

Mainly related to dynamic library handling. Dlopen, dlclose, and dlsym served me well. One one niggle that I came across though was that [FORWARD]

Linux doesn't have the Windows DLL attach messages. Surprise, right? [FORWARD]

But that's how the D runtime initialises on Windows when a DLL is loaded. Now, remember, the Binderoo Host is a C++ library that handles loading D dynamic libraries, so the D runtime was never initialised to begin with. This is an easy fix though [FORWARD]

All Binderoo libraries now implement a binderoo_startup function that intialises its copy of the D runtime. Easy. [FORWARD]

Unfortunately, this reintroduced a Quantum Break problem that I never solved properly back then. [FORWARD]

Essentially, in a heavily threaded environment the other threads in your process will

call the D runtime thread attach code via the thread attach message before the D runtime has initialised. This leads to the program crashes you would expect. Of course, Linux doesn't have this problem. You load the library, and now it's just code in memory. What you do with it from there is entirely up to you.

Now that I've had at least five minutes to think about it, the solution was simple enough. [FORWARD]

Porting To Linux And Cleaning Up Old Problems

```
version( Windows ):

__gshared CRITICAL_SECTION mThisMutex;
// core.sync.mutex.Mutex mThisMutex;   // Nope, requires GC


case DLL_THREAD_ATTACH:
    acquireMutex();
    dll_thread_attach( true, true );
```

We just use a good old critical selection and mutex lock the thread attach and detach routines behind it. [FORWARD]

Why not one of the D runtime build in mutexes? Simple. Being a class, and not a struct, means it is a reference type that needs to be allocated before usage. [FORWARD]

And that requires the garbage collector, which gets initialised with the D runtime. And since the D runtime not being initialised when our thread attaches is the problem, you immediately get memory errors when you try to allocate a mutex to perform the lock. The CRITICAL_SECTION, being a Windows value type, does not require allocation and can live on the stack. So it clearly is the correct call here.

Now, one other thing of note [FORWARD]

Is that we now introduce a mutex lock on every thread attach. Which means if you have a ton of threads in your code, you're going to notice a clear pause whenever a Binderoo library gets loaded or reloaded. On the other hand, rapid iteration is designed to be a development tool and not for retail code. If you initialise Binderoo in your program when you only have your main thread, you won't really notice the hang. [FORWARD]

But Still To Be Solved...

- Each Binderoo library has its own D runtime
- Has always been the case on Windows
  - Copying this pattern on Linux
- Normal .so usage is fine for average use case, but Binderoo needs to:
  - Load dynamic libraries, module initialisation
  - Deinit modules in one library, unload, load new code
- Why not betterC? My original users want a betterC#

There is a very big thing I still need to solve however. [FORWARD]

Each Binderoo library still embeds its own copy of the D runtime. [FORWARD]

This has always been the case on Windows, since the D runtime was never able to live in its own DLL. [FORWARD]

But I'm also copying this pattern on Linux. Linux has had the D runtime available as a .so (shared object) for a while. [FORWARD]

But that's mostly because it works just fine in that architecture for the average use case. But Binderoo is not a normal use case. [FORWARD]

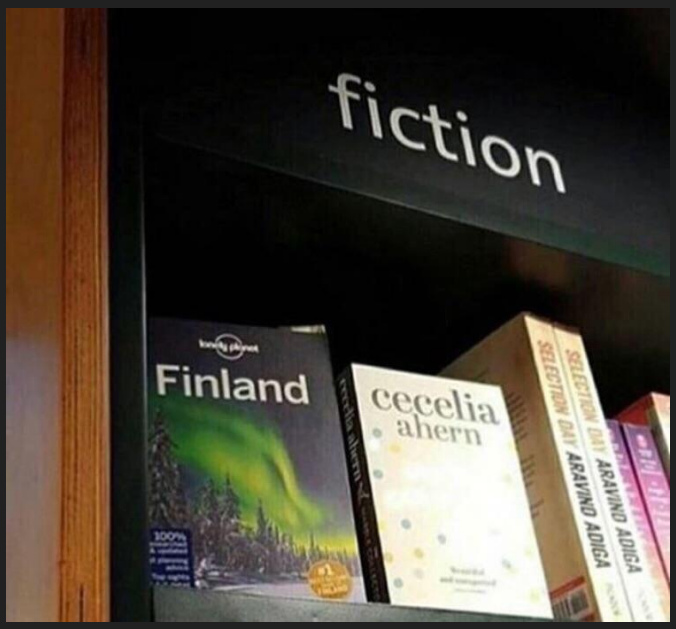
We need to load our libraries dynamically, which means modules need to be initialised. [FORWARD]

And then with rapid iteration, we need tear down just those modules in a single library, unload the library from memory, and then load new code and go through the module initialisation routines again.[FORWARD]

Of course, if you use better C, this won't be a problem. But my original users want a better C#, so that's not an option:

[QUESTION TIME ON NEXT SLIDE]

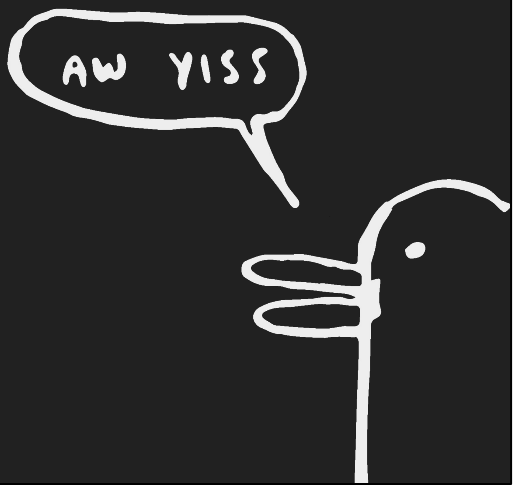
Let’s check some questions...



[FORWARD]

And Now, A “Never Planned” Feature

- Binderoo now has .NET support



There has been one feature requested for Binderoo that was never ever planned. So, here’s to the power of open source: [FORWARD]

Binderoo now has .NET support. [FORWARD]

This greatly increases the number of applications that can use Binderoo in the real world, so this is certainly a Good Thing™. [FORWARD]

binderoo_host_dotnet

- Original plan - just compile binderoo_host with C++/CLI
- This requires the full MSVC toolchain
  - ie the win32/win64 binaries
  - Means messing around to compile on Linux, OSX
- Scratch that
- binderoo_host now has a minimal C API that .NET wraps
  - (Don't use this API yourself, kids)

There's now a binderoo_host_dotnet subsection of the Host component. [FORWARD]

This is a bit divergent from the original plan of compiling binderoo_host with C++/CLI. Which actually wasn't too hard given the manner in which I wrote the code to begin with. But then I came across a small stumbling block. [FORWARD]

C++/CLI is not supported by the main .NET compilers, and instead is a component of the Microsoft Visual C toolchain [FORWARD]

And these are only distributed for win32 and win64 systems. [FORWARD]

Of course, there's nothing stopping you running them on Linux and OSX with VMs or WINE or whatever, but that's a bunch of work that shouldn't be necessary. [FORWARD]

So I didn't do that. [FORWARD]

Instead, I wrote a minimal C API that wraps up the binderoo_host functionality behind the API, and the .NET assembly wraps around that. [FORWARD]

But I really don't recommend using the API yourself. It exists solely to make my life easier when supporting .NET and is prone to change without warning. Don't, for example, D++ it and link it to your D executable.[FORWARD]

Time To Make Binderoo Better In General

```
@BindExport( 1 )
extern( C++ ) void someFunction( int someParam );

extern( C++ ) struct SomeDObject
{
    int someObjectFunction( int someParam );
}
```

So getting Binderoo running in a .NET environment wasn't that much of a problem. But then it comes time to writing example code showing how it works. And I come across a bit of an annoyance. [FORWARD]

If I wanted to export **any** function to C++ with Binderoo, [FORWARD]

I'd need to declare it as extern(C++). [FORWARD]

And that would lead to entire objects being declared as extern(C++) just to ensure the object would default to C++ calling conventions. This is an annoying bit of work that just doesn't need doing. And also, it's problematic. [FORWARD]

Time To Make Binderoo Better In General

```
module test1;

extern( C++ ) void someFunction( int someParam );


module test2;

extern( C++ ) void someFunction( int someParam );


test1.someFunction.mangleof == test2.someFunction.mangleof
```

See, if you have a function named someFunction in one module, and have another function called someFunction in another one, that gives you a compile error. Why? [FORWARD]

Thanks to C and C++ mangling rules, these two functions generate exactly the same mangled string. So it's a duplicate definition problem. Now, of course, you can get around it by specifying a namespace in the extern(C++) declaration, but again that's extra work. And you know what? We can do better with Binderoo. [FORWARD]

Time To Make Binderoo Better In General

- Generate a wrapper for every non-C++ @BindExport
- This is a funky chain for .NET interop
  - Marshall function call to native/C++
  - C++ wrapper in to D function
  - D code executes, returns
  - C++ wrapper the return value
  - Un-marshall the return value

We can ditch the requirement for declaring a function as extern(C++) entirely. Binderoo already does a bunch of compile-time code generation. So what's a bit more? [FORWARD]

Whenever we find a function with a @BindExport tag, now we just go through and generate an anonymous wrapper function for it and export a function pointer for that function instead of the actual function. Any code calling that function from D code uses the native calling convention, and code from outside is none-the-wiser. [FORWARD]

This does make for a bit of a funky chain for .NET interop [FORWARD]

Since you have to marshal your parameters to native format and then invoke a C++ function [FORWARD]

Which then rewrites the parameters in to the D ABI format in order to call a D function [FORWARD]

Which executes, and can return a value in the D ABI format [FORWARD]

Which the C++ wrapper then returns in the C++ ABI format [FORWARD]

Which then finally gets un-marshalled so that you can use it in .NET. Unfortunately, short of the .NET runtime and C compilers supporting D calling conventions there's

not much that can be done here. But hey, it works! [FORWARD]

But Suddenly, LLVM

- Every symbol is generated and exported
- C++ function inside a template with string parameter? HAH! YOU FOOL!
- Solution: Cheat

```
pragma( mangle, "wrapper_test1_somefunction0" )
```

And just when it looked great, I compiled with LDC and hit an LLVM specific problem. [FORWARD]

Every symbol is generated and exported. Including my anonymous functions. And, even better, [FORWARD]

A C++ function inside a template that uses a string parameter is just right out of the question. You immediately get a compiler error saying that C++ types can't be templated with strings, or any D specific type really. So you know what I did? [FORWARD]

I cheated. [FORWARD]

I generate a new mangle for it and generate a pragma mangle. So there's no hope of ever automatically resolving this symbol the traditional way, but hey, we have basic unique mangles for our function and thus there's no more symbol clash. LLVM can continue doing LLVM things and our code will continue to work. [FORWARD]

Binding In C#

- Start by creating an ImportedFunction< RT > object
  - ImportedFunction< RT, P0 >
  - ImportedFunction< RT, P0, P1 >
  - ImportedFunction< RT, P0, P1, etc etc etc >

Now that we export our functions just fine, we need to import them in to .NET. Knowing that C# objects use the concept of “generics” similarly to how templates work, I decided to be a nice modern programmer and [FORWARD]

I started off creating an ImportedFunction generic object. C# generics aren’t that powerful though, so to support multiple parameters with your function, [FORWARD] [FORWARD] [FORWARD]

You need to go old school C++ style and define each permutation you want manually. Okay, sure, this work only needs doing once, it shouldn’t be that much of a problem. [FORWARD]

Binding In C#

```
private delegate RT DT( P0 p0, P1 p1 );

private IntPtr FP;

public RT call( P0 p0, P1 p1 )

{

    DT del = (DT)Marshal.GetDelegateForFunctionPointer( FP,
typeof( DT ) );

    return del( p0, p1 );

}
```

NO GENERICS FOR YOU!

It's fairly simple code here. Nothing too tricky. Our function pointer gets retrieved from Binderoo [FORWARD]

And stored in the FP value on construction and whenever it gets reloaded. Cool. Compile it, and it works fine. Run it though? Well. [FORWARD]

The .NET runtime will throw an exception on this line. The reason? [FORWARD]

The typeof parameter refuses to accept a generic type - DT being a delegate definition inside a generic object means that it is a generic type. I have no idea what the reason for this is, and it is really an utterly stupid restriction from a user point of view. [FORWARD]

Binding In C#

```
ImportedFunction< bool,
                  [illegible](UnmanagedType.LPStr) string,
                  int > func;
```

NO ATTRIBUTES AS GENERIC TYPE MODIFIERS FOR YOU!

There's also something else annoying with generics. When calling a native function, [FORWARD]

you can use the MarshalAs attribute to automatically convert a .NET managed type in to a native object. Convenient. Except [FORWARD]

It doesn't work like that. It can only be applied directly to those function definitions and delegate types. You can't add attributes to generic parameters. So even if the GetDelegateForFunctionPointer were fixed to accept generic types, being unable to apply attributes easily to the generic type means you'll need to manually convert your C# types to D types before calling it. And that's essentially unusable in my opinion. I want to have a very long talk with whoever created C# generics. [FORWARD]

Binding In C# (If You Actually Want It To Work)

- Binderoo can already generate C++ to automatically export functions…
- …so make Binderoo generate C# code to automatically import functions

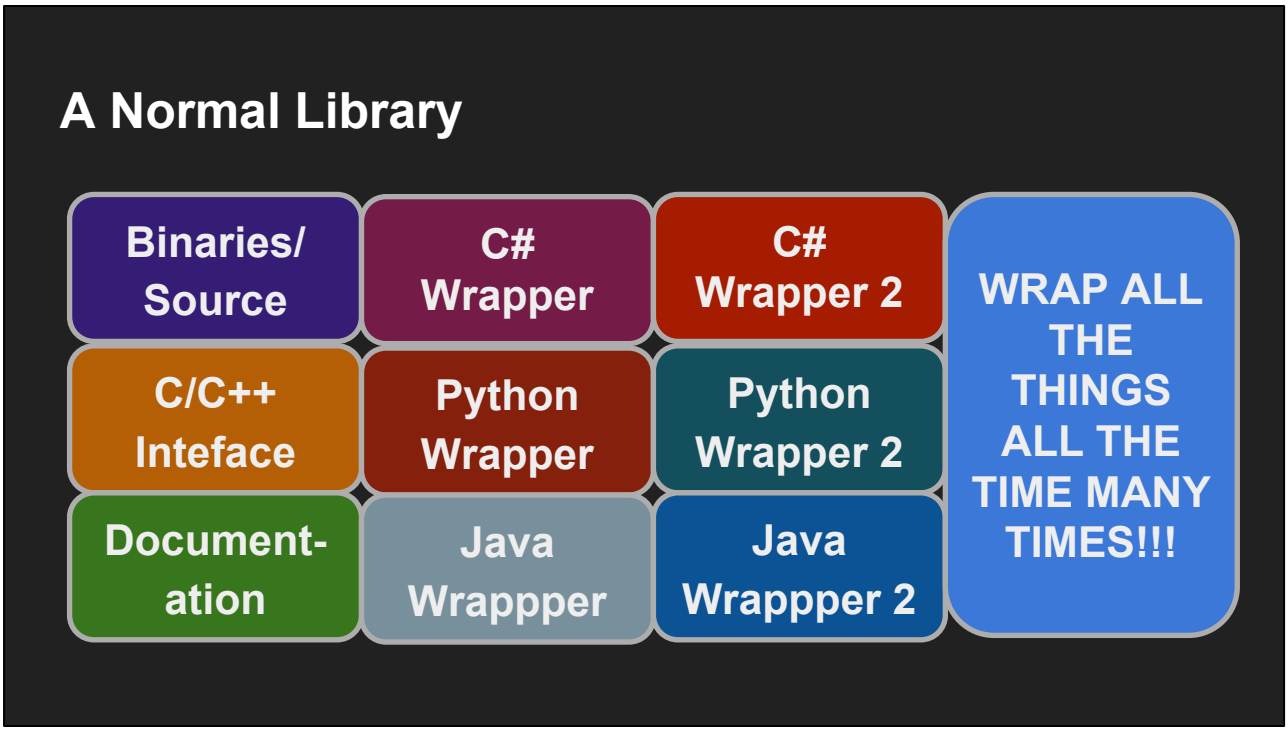
- Binderoo - Doing the work the C# compiler really should be doing itself since 2018

So there's nothing for it. That work gets deleted, never to make a compiler cry again. But we need a solution somehow. And this is where Binderoo's native functionality points towards a solution. [FORWARD]

Using binderoo_util, you can load up any Binderoo DLL and ask it to generate a C++ code file that, when included in your host program's build, automatically exports all required functions to Binderoo. It's not much of a leap from there [FORWARD]

To get Binderoo to generate C# code that does the reverse - import functions from a Binderoo DLL. [FORWARD]

Which really is something that I think the .C# compiler should handle for me. But on the other hand. It highlights a very cool paradigm that Binderoo is stumbling in to. [FORWARD]



Alright, so if you were to release a normal library on the internet [FORWARD]

You'd release binaries, or if you're an open source project you'd release pre-packaged binaries and the source code. [FORWARD]

And you generally also want to ship a C or C++ interface with your library, especially if it's a binary only release. [FORWARD]

And documentation. Documentation is certainly important. With everything packaged and release, you congratulate yourself on a job well done. But that's not where things end. [FORWARD]

Because if your library is useful, someone on the Internet will have to make a C# wrapper [FORWARD]

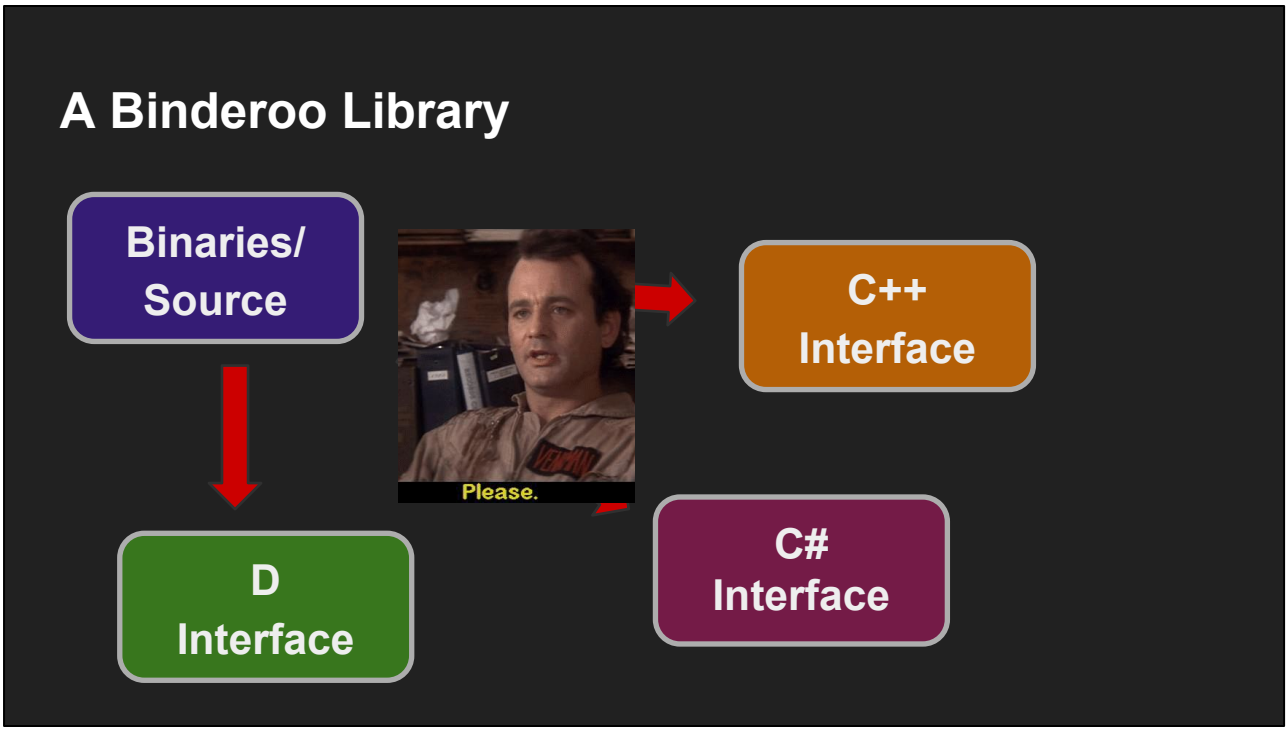
And a Python wrapper [FORWARD]

And a Java wrapper. And then they won't ever be maintained so there'll be [FORWARD]

another set of wrappers after a year that people need to investigate and choose between. [FORWARD]

And before you know it, the internet has created wrapper hell. Seriously, just look in to

OpenGL some time for an example of this. [FORWARD]



A Binderoo library has one key difference. [FORWARD]

It starts off in the same way - upload binaries and/or source. And immediately without doing anything special [FORWARD]

you can get a D interface [FORWARD]

A C# interface. And, with a little bit of effort on my part [FORWARD]

it will even be able to generate a C++ interface. And all of this is achieved essentially [FORWARD]

By asking it nicely. Load it up with binderoo_util and get all the information you need out of it. This is essentially the same approach as .NET assemblies. It doesn't matter what .NET language you use to author an assembly, each language knows how use it without needing wrappers. That's pretty cool and far more preferable to the old way in my opinion. [FORWARD]

Binding In C# (If You Actually Want It To Work)

```
namespace testmodule
{
   private static ImportedFunction func0;
   private delegate void d_func0( int foo, float bar );
   public static void doAThing( int foo, float bar )
   {
      if( func0 == null ) func = new ImportedFunction(
"testmodule.doAThing", "void(*)(float, int32_t)" );
      d_func0 call = (d_func0)GetDelegateForFunctionPointer(
func0.FuncPtr, typeof( d_func0 ) );
      call( bar, foo );
   }
}
```

NOW DO IT PROPERLY!

Alright, so we're in C#. To preserve our D module names, we put everything in namespaces [FORWARD]

And whoops we've already encountered our first problem. You can't declare static variables inside a namespace. Or static functions. Nope. That's not how C# works at all. [FORWARD]

So we have to do something really nasty - even though C# is going to tell us this is the proper way to do it. [FORWARD]

(Error message if you try to compile this: `Expected class, delegate, enum, interface, or struct`)

Binding In C# (If You Actually Want It To Work)

```
public static class testmodule
{
    private static ImportedFunction func0;
    private delegate void d_func0( int foo, float bar );
    public static void doAThing( int foo, float bar )
    {
        if( func0 == null ) func = new ImportedFunction(
"testmodule.doAThing", "void(*)(float, int32_t)" );
        d_func0 call = (d_func0)GetDelegateForFunctionPointer(
func0.FuncPtr, typeof( d_func0 ) );
        call( bar, foo );
    }
}
```

We're going to use public static classes **everywhere** instead of namespaces. Which allows us to preserve D module names and avoid symbol conflict. I'll get to why this is nasty in a second. Now, there's a whole bunch of stuff that is hidden down the bottom of the generated code file wiithin a #region block, but right here it's all on display to show you how it works. [FORWARD]

The ImportedFunction object has survived, but it is no longer a generic object. Rather, it is a .NET object that wraps everything Binderoo does. It's fairly useless by itself, and it absolutely needs [FORWARD]

The generated support code to work. But as far as anyone programming in C# needs to know… [FORWARD]

Binding In C# (If You Actually Want It To Work)

```
public class SomeCSharpClass
{
    public void someFunction
    {
        testModule.doAThing( 42, 2.71828 );
    }
}
```

They write their code naturally and everything just works. Although, when I say naturally, [FORWARD]

That is a bit of a lie. See, because everything needs to be in public static classes to work, you need to fully qualify your D function every time you call it. No **using** declarations for you. This is really not ideal as I see it, but hey it works! [FORWARD]

Binding In C# (If You Actually Want It To Work)

```
public static class testModule
{
    [ StructLayout( LayoutKind.Explicit, Pack = 8 ) ]
    public struct SomeTestData
    {
        [ FieldOffset( 0 ) ] public int foo;
        [ FieldOffset( 8 ) ] public double someDouble;
        [ FieldOffset( 16 ) ] public int bar;
        [ FieldOffset( 20 ) ] public short dwarf;
        [ FieldOffset( 24 ) ] public float someFloat;
        [ FieldOffset( 28 ) ] public short hobbit;
    }
}
```

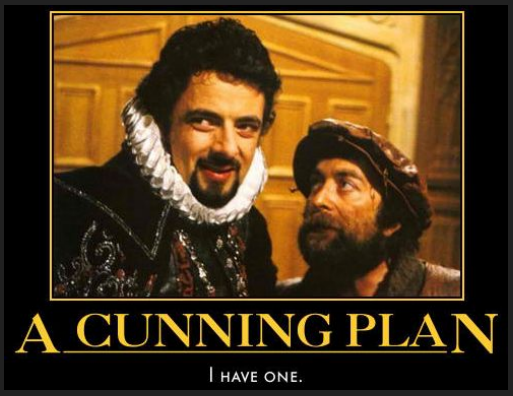
Now, just calling D functions is great. But we need more. Especially as a game developer, you really don't get far without aggregate types. So we need to export compatible aggregate types to C# that can work interchangeably with D. [FORWARD]

C# already can match the C ABI's definition of aggregate layouts, so that's cool. Value types, ie structs, in D match the C ABI so that's easy. Just generate C# duplicates of any aggregates that Binderoo knows about. But what about classes? Well... [FORWARD]

Previously At DConf 2017...

Value And Reference Types

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



Last year when I presented my talk here, [FORWARD]

I had a slide on value and reference types, and went in to my rationale for binding all C++ objects with D structs since their ABIs matched, whereas a D class has its own custom ABI. The reason I'm bringing this up [FORWARD]

C# Also Has A Reference Type

- class
- The ABI is technically unpublished
  - Which means, yes, class binary layouts ***have actually changed*** between CLR versions
- Binderoo operates as “allocate once, use anywhere”
  - Won’t wrap D classes as C# structs
  - But C# classes won’t “allocate once, use everywhere”
    - Unless we hide everything...

Is that C# also has a reference type. [FORWARD]

And it is also called a class [FORWARD]

But this is where things get tricky. Because while Microsoft has been officially open sourcing .NET, the ABI is technically unpublished. This is really advantageous for Microsoft [FORWARD]

Because by not publishing it, and telling programmers that they don’t need to know about it, it means that they can change the ABI behind the scenes. And they have done this. Most recently, CLR 4.0 introduced a new class layout. [FORWARD]

Now, Binderoo operates on a “allocate once, use anywhere” principle. Essentially, interoperation works because each object operates just as well regardless of which language you allocated it in. [FORWARD]

Wrapping a D class as a struct though means treating a D class as a value type, and that’s full of its own set of problems. [FORWARD]

And since we can’t trust the binary layout of C# objects, attempting to recreate the object in C# and allocating it there absolutely will violate the “allocate once, use anywhere” principle. [FORWARD]

But it turns out I have a cunning plan after all: just hide everything. [FORWARD]

Binding Reference Types In C#

```
public static class testModule
{
   public class SomeTestClass
   {
      ImportedClass pObj;
      …
      public void someTestFunction( int someParam )
      {
         …
         call( pObj.Ptr, someParam );
      }
   }
}
```

Rather than trying to copy the binary layout [FORWARD]

We just make D allocate it. Allocate a SomeTestClass in C#, and it will make the appropriate Binderoo calls to create an object through the C++ API and store it in the ImportedObject. [FORWARD]

And whenever the object needs to be passed to a function, since we generate all this code anyway then we simply retrieve the pointer and forward it along. Sweet. C# users are now free to use objects written entirely in D! Pretty pretty pretty sweet. [FORWARD]

If you’re reading this at home, I will launch a dodgy calculator with a .NET frontend and a D backend using Binderoo interop. [FORWARD]

It’s almost the end of the talk!

And with that, we’re just about at the end of my talk. Upgrading Binderoo has been one of the things I’ve been spending time on. Some of those modifications I’ve talked about have been the result of a consultancy, in fact, so I can put “D Consultant” on the resume. But I’ve also been spending a bit of time on something else that’s fairly significant. [FORWARD]

I quit Remedy because it was time for me to get out and do my own thing. I've been putting together a pitch to get investment for a company, which will write the kind of tech I need to make the kind of games I want. And yes, I've already started writing that tech in D.

That's about all I'll say about it for now, but that up there will be our logo. This is its first public appearance. And when I'm in a position to tell y'all more, well, I'll make an announcement.

[FINAL QUESTION TIME ON NEXT SLIDE]

Final questions time!