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:

(Who is here because they couldn't get into Romero's talk?)

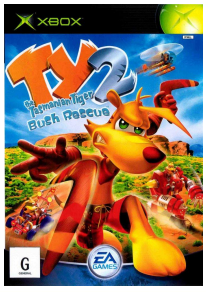
(Yeah, totally. I mean, I released a Doom mapset last year that was a Cacoward runner up, and which itself was a follow-up to a Cacoward winning map I released in 2003. But instead of watching his talk, I'm here presenting mine. Sacrifices have been made.)

[FORWARD]

Ethan Watson

AUSTRALIAN MADE

14 Years
11 Platforms



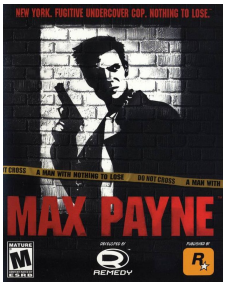
So a bit about myself to begin with. As the title slide gave away, I'm Ethan Watson.

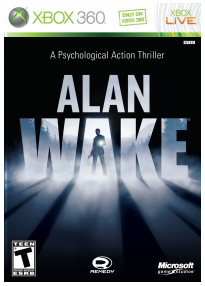
[FORWARD] And I come from the fictional land of Australia.

[FORWARD] I've been in the industry for almost 14 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; and, most recently, Quantum Break.

[FORWARD]

REMEDY





As you may have gathered, I work for an independent studio known as Remedy where I am currently a senior engine programmer.

[FORWARD] Over our 21 year history we've released games like Max Payne and Alan Wake.

[FORWARD]

Of course, the reason we're all here today is because of our most recent release - Quantum Break.

Also known to my girlfriend as the reason she didn't see me for a few months… and the reason why I missed her 40th birthday (it's cool, I'm sure she'll have another one).

BUT it's done, it's out, it got a good reception. And it means I can talk about our usage of the programming language D in the game, which made us the first studio to ship a AAA game using the language.

A bunch of the work I'm presenting here today was started before I got to Remedy by another programmer, Manu Evans. I've known him since 2002, back before we started in the industry, so when he decided to go back to Australia a month after I joined Remedy I was the natural choice to take over his work. Which lead to very quickly needing to learn some in-depth concepts with little more than an explanation of the code from Manu and the code itself as an example. Today, I aim to present everything in a human-understandable format throughout this talk.

But before we get into that, we need to talk about why we decided to try D in the first place.

[FORWARD]

Rapid prototyping

- Programmers wanted a "scripting" system
- Settled on compiling native code in a DLL
- "Why not D?"
- Yeah, why not D?

Our usage 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. Compiling and linking C++ code is slow and painful, so something needed to be done

[FORWARD] Various scripting solutions and things like Mono were discussed before the idea of compiling native code into a DLL was settled on. One of the programmers at the time, Tristan Williams, had previously worked at Splash Damage and would talk often about how nice rapid prototyping in idTech 4 was, which compiled C code to DLLs.

[FORWARD] Rather than being stuck with C in a DLL, Manu, who is a D enthusiast, said "Why not D?"

And to his surprise, the response from everyone was…

[FORWARD] yeah, why not?

Well, that's a lie. Not everyone said that. There was another

common response:

[FORWARD]

(Image source: http://medicalstate.tumblr.com/post/81104919425/why-not-zoidberg-by-barry-doyon-march-28th-was)

What is D?

- Statically typed, compiled language
- Designed to succeed C++
- Interoperates with C and C++
- Modern language features
- Fast compile times
- Ridiculously powerful compile-time features

What is D? That's a good question.

[FORWARD] It started life as a statically typed, compiled language…

[FORWARD] that was designed to succeed C++.

[FORWARD] To that end, it interoperates with C seamlessly, and C++ to varying degrees thanks to C++ not having a single unified application binary interface.

[FORWARD] The big winning factor is that it is a completely modern language with all the modern bells and whistles like lambdas and properties and all that jazz, so it's appealing to programmers.

[FORWARD] And appealing to everyone is its fast compile times.

[FORWARD] But, of particular interest, is its ridiculously powerful compile time features that sets it in a league of its own.

Googling for D is tricky though. Interestingly, it wasn't originally called D. It was originally called Mars, which led to the D standard library receiving the name "Phobos". It was the community which demanded it be called D.

[FORWARD]

He LOVES ~~the D~~

the **dlang**

http://www.dlang.org

This leads to some rather unfortunate jokes.

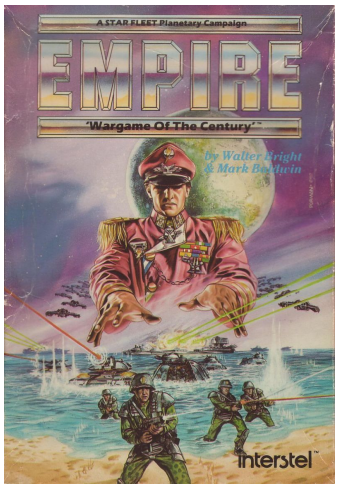
[FORWARD]

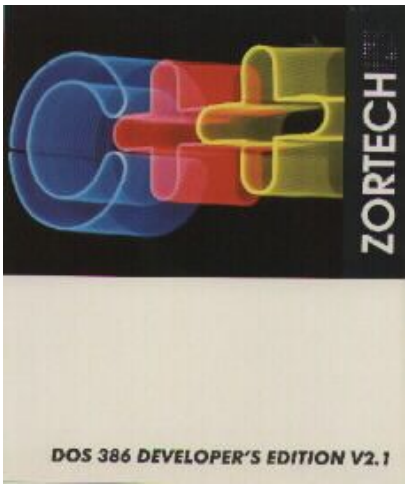
But there is something of a solution for it.

[FORWARD] The official way to google for the language is to use “dlang”.

[FORWARD] The homepage for D is in fact dlang.org, and I find limiting google search results to that domain usually gets you what you’re after thanks to some very active newsgroups.

[FORWARD]





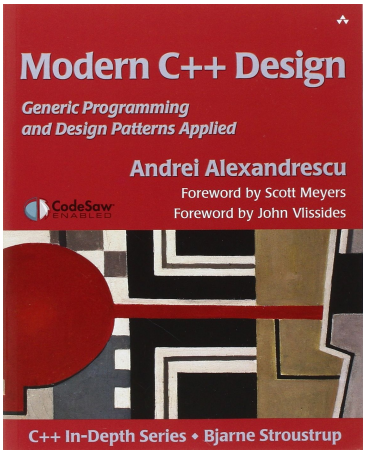
D was started by Walter Bright, who writes and maintains the reference compiler for the language called DMD.

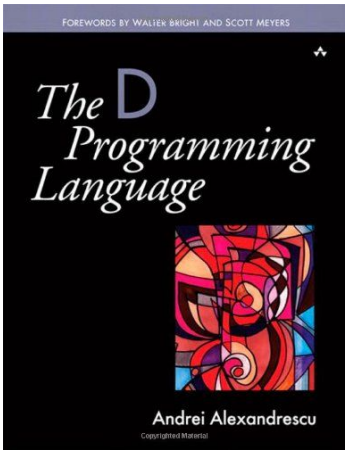
[FORWARD] Now, this guy is actually an OG vidya game developer. Back in the 70s, he wrote a game for mainframes called Empire, which eventually made its way to every major home computer with a keyboard in the mid-to-late 80s. My father played the bejesus out of the Amiga version.

But by that point in time, Walter had already found his calling.

[FORWARD] He had written and released the first native C++ compiler, Zortech C++. Anyone that's been around long enough may be familiar with it. D started as a re-engineering of C++ but has become something much more than just that.

[FORWARD]





A few years later, he was joined by a guy called Andrei Alexandrescu.

[FORWARD] Andrei has the distinction of having made C++ relevant in this century thanks to a little book he released in 1999 called Modern C++ Design, which popularised template metaprogramming. If you've ever done more with templates than just changing the storage type inside a class, then chances are you've done something popularised by this book.

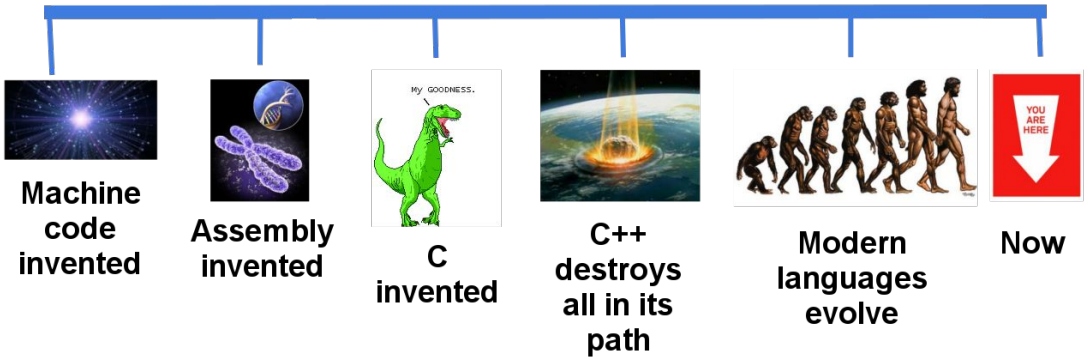
[FORWARD] He has also written the seminal book on D. If you want to learn the language, this book is generally the first stop.

So that's two big names in C++ powering the language. But why are they off making a new language instead of trying to improve C++? I can't speak for them, but I can give you my own take on C++.

[FORWARD]

Die C++, Die: Part One

TIMELINE OF THE UNIVERSE



(That's not German for "The C++, The")

Can I have a show of hands here of people that like C++?

Okay, now those with your hands up, lower them if you like another language better than C++ and would rather program in that.

Now, those of you remaining: Are you familiar with the term "Stockholm Syndrome?"

Sure, C++ has been paying my bills for about 14 years now. But you know what? Maybe it's time to move on.

[FORWARD] I mean, if you look at this timeline of the universe, look at what role C++ plays in the grand scheme of things. New languages and programming paradigms have evolved since C++ was new and fancy, and honestly, the additions to the language in C++11/14/17 just aren't enough or are far more obtuse than they need to be.

[FORWARD]

Die C++, Die: Part One

```
template< typename T >
inline T lerp( const T& from, const T& to, float ratio )
{
    return from + ( to - from ) * ratio;
}
```

Let's just take a really simple example - an interpolation function. Here's one we've written in a generic manner. And using a fairly standard interpolation formula. from plus brackets to minus from close brackets times ratio. Templated means that anything that can add, subtract, and multiply will Just Work™ with this code.

[FORWARD]

Die C++, Die: Part One

```
float from = 0.1f, to = 0.95f;
float result = lerp( from, to, 0.5f );
```

```
unsigned char from = 230, to = 110;
unsigned char result = lerp( from, to, 0.5f );
```

And for for something like interpolating between two floats, it will work just fine. The result from that example will be 0.525.

[FORWARD] But then we have a problem here. That lerp function, when interpolating from a higher to a lower value on unsigned values, will trigger the integer wraparound. Which results in incorrect values. The result when trying to interpolate half way from 230 and 110 when both values are 8-bit unsigned integers is in fact...

[FORWARD] 42, not the expected 170.

[FORWARD] But don't panic. We can fix this.

[FORWARD]

Die C++, Die: Part One

```
template< typename T >
inline T lerp( const T& from, const T& to, float ratio )
{
    return from * ( 1.0f - ratio ) + to * ratio;
}
```

There's another form of an interpolation function that we can use.

[FORWARD] The second one has an additional multiply with one minus the desired ratio, and both raw values are multiplied with. The result is that it handles unsigned values correctly. Hooray! And with a bit of template specialisation…

[FORWARD]

Die C++, Die: Part One

```
template< typename T >
inline _ty lerp( const T& from, const T& to, float ratio )
{
    return SignSensitiveMath< T, std::is_signed< T >::value >
                ::lerp( from, to, ratio );
}

template<> class std::is_signed< Vector3 > : std::true_type { };
// Not necessary for false, but required for habitual reasons
template<> class std::is_signed< ColourRGBA > : std::false_type { };
```

...we can rewrite our lerp to call a specialised container based on the result of std::is_signed to get us the correct lerp function.

So we're done, right? Well, no.

This solution solves lerping for basic types. But what about aggregate types, like a three-dimensional vector or a RGBA colour?

We need to know if our complex type has signed values, and with this code…

[FORWARD] the solution is to specialise std::is_signed for every single object passed into the function. And as this can be quite easy to forget, that results in making something as simple as a lerp function a buggy, unmaintainable mess.

[FORWARD]

Die C++, Die

```
template< typename T
inline _ty lerp( con                    ratio )
{
    return SignSensi                    >::value >
                ::le
}

template<> class std                    rue_type { };
template<> class std                    ::false_type { };
```

So, fine. To keep the generic function generic and maintenance free, the solution here is to use the sub-optimal two-multiply version for the generic version.

But that doesn't mean things have to be the same in D code.

[FORWARD]

Die C++, Die: Part One

```
auto lerp( T, F )( ref const( T ) from, F ratio, ref const( T ) to )
    if( isFloating!( F ) )
{
    static if( HasUnsignedMembers!( T ) )
        return from * ( 1.0 - ratio ) + to * ratio;
    else
        return from + ( to - from ) * ratio;
}
```

Here's the exact same function written in D.

The auto return type should be familiar to C++11 programmers.

[FORWARD] But notice how templates are defined - a first set of bracketed parameters. Instantiating a template, to avoid confusion, requires you to use the exclamation mark followed by those parameters. You don't get the double angular bracket problem you get in C++ templates as a result.

[FORWARD] But there's two extra very important concepts here. The first one is template constraining. You can constrain the template parameters inside their declaration, but far more useful in this case is to use an if constraint *after* the declaration. This will check if F is any kind of a floating type - D has built in support for several types of floating point types, so rather than writing several specialisations to handle each type constraint we simply use the if constraint to get it all in one.

[FORWARD] The next one there is static if. The preprocessor in C is like baby's first compile time branching compared to

static if. Static if will resolve every unique time it is encountered. Such as in this template, it will resolve for every template instantiation. Using static if, we can check if type T has unsigned members. If it does, then it will compile the unsigned form of the function. If it doesn’t, then it will compile the signed form.

[FORWARD]

Die C+

```
auto lerp(                                    ) to )
    if( is
{
    static
        re
    else
        re
}
```

And that's it. We've written the function once, constrained it to correct parameter types, and inspected types using built-in language features. It'll now handle any type we can throw at it, generate optimal code in each case, allow you to specialise for micro-optimisations if you so desire, and assuming your linker is reasonable and can do identical function folding your template code won't blow out when used on very similar types.

[FORWARD]

Die C++, Die: Part One

	Template parameter constraints	Template if constraints	static if
Swift	**Yes**	**where**	**No**
Rust	**Yes**	**where**	**No**
C#	**where**	**where**	**No**

Of course, there are other, trendier modern languages out there. And they're all nice enough replacements for C++ in many use cases. Swift and C# are immediately relevant to gamedev. And if D wasn't a thing, I might even be aboard the Rust bandwagon (and to be clear, I'm comparing core Rust functionality, not Rust plus your favourite compiler plugin).

But sometimes, certain features are either lacking or are incomplete to do it as elegantly and as effortlessly as you can do it in D.

In this case, many languages will have "where" constraints that allow you to check types only and require you to either specify complex chains or to change the makeup of your type to fit it. Rust can get away with a lot thanks to traits, but it's still hasn't gone as far as it should go.

An if constraint is far more powerful in that it works just like an if statement, so if the declaration inside it resolves to true then you're good to go. I'll get into why this is *very* powerful later.

And when it comes to static if? None of these languages are

doing it. (Rust can kinda do some very limited things with its config macros). Walter, Andrei, and Herb Sutter actually submitted a proposal for static_if to the C++ standards boards. To put it politely, Bjarne Stroustrup is not a fan (look up the papers online). So if it does come to C++, chances are it'll be a pain to use like lambdas are.

(No, seriously, who decided the lambda syntax was a good idea???)

[FORWARD]

Die C++, Die: Part One

Problem: C++ is a pain.

Solution: Use a different language.

There will be other examples in this talk, but in each case the moral of the story is simple: If C++ is a pain, then it's time to use a different language.

[15 MINUTE MARK?]

[CHECK QUESTIONS HERE]

[FORWARD]

D on Our Target Platform

- Situation wasn't great a few years back
  - LDC
  - GDC
  - DMD

So, let's get back to integrating D.

Back when we started with D, there was a very good question of whether we could ship it on Xbox One. At that point, the Xbox One hadn't even been announced yet, and information surrounding it was under a very heavy lock with a very sharp key. But we knew it would be a 64-bit Windows OS, so we should be able to get it generating reasonable code.

[FORWARD] Compilers, though, weren't quite there a few years back.

[FORWARD] LDC was in a dire state for Windows. So it was out of the question. It is very nearly 100% usable these days, debugging symbols are still an issue on Windows at least.

[FORWARD] GDC had working 64-bit output, but it lagged behind mainline features and bugs found had a two-or-more week turnaround from report to fix. This wasn't really good enough for our needs.

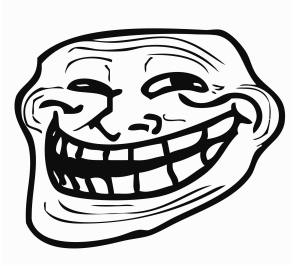
[FORWARD] DMD was always up to date. Only problem? It only generated 32-bit binaries. That made it, at that point

unsuitable for our use. But being up to date on the bleeding edge was going to be highly advantageous.

[FORWARD]

Go with DMD

- Get 64-bit binary support
  - Output to mslink compatible formats
- Got it! It works!
- And Xbox One works!



So we made the decision to go with DMD.

[FORWARD] Of course, we're in the future now. 32 bit is so passe. We needed 64 bit support, and DMD did not have it.

[FORWARD] So we asked Walter for 64 bit support that we could link with Visual Studio's linker. This required Walter to write output in the Common Object File Format.

[FORWARD] With the correctly formatted object files, we could then link with the XDK's tools. End result, Walter provided us with Windows 64-bit support for the compiler frontend. We could now compile and link and load our D DLLs in our 64-bit codebase on our work machines, all running Windows 7 64-bit at that point in time.

[FORWARD] And since we got 64-bit support via Microsoft's Visual C linker backend, all we needed to do was plug in the XDK's linker to our toolchain and the end result was that we got Xbox One support basically for free, and Walter was none-the-wiser that we were able to do this.

[FORWARD to trollface]

[FORWARD]

Development Environment

Visual D

Visual Studio 2005-2015 support

http://rainers.github.io/visuald/

Having a good development environment is essential for game programming these days. Specifically, on Windows, if it doesn't have Visual Studio support then it's basically not usable.

[FORWARD] Thankfully, a man called Rainer Schütze was there with a package for Visual Studio called Visual D, which provides integration of D into Visual Studio with all the bells and whistles you'd normally expect.

It's safe to say, and I've told him this in person, that Remedy would not have used D had this not been a Thing™. It still has some flaws in it, but one of the results of the annual D Conference this year in Berlin was that the D Foundation has decided to put money into Visual D. So it's going to be getting even better in the future.

[FORWARD]

Code as Data

- Exports to DLL in global block
  - Back door for programmers
- Implementation was sssllllllooooowwwwww
  - No platform distinction for data sets
  - Debug + Release for Win7 + XBO + UWP
- Being improved right now

Alright, so we've got a compiler and we've got a development environment. Sweet. But while that's enough to get us writing Windows applications, we need to do more work to support rapid prototyping. Which meant that we treat D code as data. This is essentially the same idea that powers Unity and Unreal Engine 4, but as we shipped it in Quantum Break it was a fair bit clunkier.

[FORWARD] We compiled our D code in our global export block.

[FORWARD] Programmers had a back door via a plugin for Visual Studio that would kick off an export for a single module, printing the output to Visual Studio so that you could do all the normal things like go straight to the problem lines.

[FORWARD] The implementation, however, ended up being slow by the end of the project. The major reason?

[FORWARD] A decision was made very early on that we would just use a single set of data for all platforms. It's not something I ever agreed with and it did bite us towards the end of the project in other areas, such as pre-tiling textures

and using XMA audio for Xbox while still having a dataset we could use on our work PCs.

[FORWARD] But specifically for our D plugins, it meant that exporting your D code required compiling both debug and release for our work platform, Windows 7, and our release platforms, Xbox One and UWP, at the same time. This was a minute long process for modules, whereas just compiling release OR debug for the work platform was down in the two or three second mark.

[FORWARD] This system is currently being significantly improved, so that it is quite a lot more like a Unity-style workflow and not what we had. But I'll get to that later on.

[FORWARD]

Die C++, Die: Part Two

```
template isFloating( T )
{
    enum isFloating =  is( T == float )
                    || is( T == double )
                    || is( T == real );
}
```

We'll take a slight detour now, so that I can explain in greater depth some code I introduced earlier. Specifically, the template isFloating. This is a template in the standard library that would look a little something like this.

It makes use of a feature known as "eponymous templates", where naming a symbol inside of a template the same as the template's name means that the template resolves to this new symbol. In this case, a boolean.

[FORWARD] It calculates that boolean with the is operator. The is operator is a handy little thing that checks types. It has far more uses than what is on display here, but this is the bare basic use case for it.

[FORWARD]

Die C++, Die: Part Two

```
template HasUnsignedMembers( T )
{
    enum HasUnsignedMembers = hasUnsignedMembers!( T )( );
}
```

HasUnsignedMembers is a custom template, and does a similar thing- wait, hold on...

[FORWARD] That looks like it's assigning the results of a function call to the enumeration.

This is another one of D's killer features - Compile Time Function Evaluation. If you can write your function to follow functional programming rules (ie pure, no exceptions, memory safe), then you can call and evaluate it at compile time. One of D's philosophies is that the more you can do at compile time, the faster your runtime code will be.

[FORWARD]

Die C++, Die: Part Two

```
static if( isBasicType!T ) bResult = isUnsigned!T;
else foreach( Member; __traits( allMembers, T ) )
{
    static if( is( typeof( __traits( getMember, T, Member ) ) ) )
    {
        alias MemberType = typeof( __traits( getMember, T, Member ) );
        bResult |= hasUnsignedMembers!MemberType( );
    }
}
```

So let's take a look at the implementation of hasUnsignedMembers.

Before I explain all this, I must point out that while this code is correct, it will only work on simple types and aggregate types with no private members. This is cut down for brevity.

[FORWARD] So we see a static if to begin with. Pretty simple security, if you're already a basic type then you only care if it is signed or unsigned - isUnsigned there is another standard library template. But what if it's an aggregate type? Well, that's where D has another little magic trick.

[FORWARD] It's the compile time traits system! allMembers of T gives us a tuple of strings, which we can then foreach over. Now, this will give you ALL members of your object. Variables, functions, sub-objects, uninstantiated templates, constants, everything. So we need to go in and check whether the symbol we're parsing is actually a variable.

[FORWARD] So we static-if based on whether we can get the type of the member. But to do that, we need to get the member first. What good is a string, after all? So we use the

getMember functionality of the traits system, which does the symbol resolution for us. We then simply get the type of it and do an is check as mentioned.

[FORWARD] From there, we simply recursively call the hasUnsignedMembers function and merge the result with our current cached one. If anything is unsigned, that will be true and we can continue on as normal.

Of course, as I've pointed out, this only works for very simple structs. If you were to write this in a truly generic fashion…

[FORWARD]

Die C++, Die: Part Two

```
static if( isBasicType!T ) bResult = isUnsigned!T;
else foreach( MemberType; VariableTypesOf!T )
{
    bResult |= hasUnsignedMembers!MemberType( );
}
```

You'd go ahead and wrap that up in a little template, which I'll call VariableTypesOf, that goes through and parses the symbol and sticks them in a type tuple for you to iterate over at your leisure. Any errors in logic can be fixed in your template, and now you can just get the variable types of any given aggregate type whenever you want.

[FORWARD]

Die C++, Die: Part Two

```
static if( isBasicType!T ) bResult = isUnsigned!T;
else foreach( MemberType; VariableTypesOf!T )
{
    bResult |= hasUnsignedMembers!MemberType( );
}
```

Handy.

[FORWARD]

Die C++, Die: Part Two

	Eponymous templates	Compile time reflection	CTFE
Swift	**No**	**No**	**No**
Rust	**No**	**No**	**Macros**
C#	**No**	**No**	**No**

Now, this table looks a lot more interesting than the last one. The other languages here have similar features in some cases. C# and Swift have runtime reflection capabilities, not compile time. Rust doesn't have anything built in to the base language, you need a compiler plugin to get that working (which in itself gets silly, it results in endless dialects of Rust).

Eponymous templates are a nice-to-have, but CTFE and compile time reflection are critically important features for our purposes. The realistic alternatives are not comparable.

[FORWARD]

Die C++, Die: Part Two

What is the one thing you can do in D that you can't do in C++?

Save time.

But you know, tables comparing features is one thing. You can list features out, and people will come up with ways to do something similar in C++ (or, to be fair, their favourite language, it's not solely a C++ phenomenon). Either way, when someone asks me, "Alright, so what can you do in D that you can't do in C++?" I always have a very simple answer for them:

[FORWARD] Save time. Everything you can do in C++, I can do quicker and more elegantly in D. Saving time in your work shouldn't just be about better pipelines, or hot reloading. Programmers spend a lot of their time writing code, so why not speed that part of the process up by using a different language?

[30 MINUTE MARK?]

[CHECK QUESTIONS HERE]

[FORWARD]

Code as Data - Binding DLLs

- Functions
  - Any namespace
  - Static member functions
- Classes
  - Member functions
  - Virtual tables

So now we get down to the meat of things. Our D code will export to DLLs which we can load in and interact with at runtime. Now, there's two ways to work with DLLs. The simpler way these days is to statically link a little lib generated by the compiler that will automatically load up and bind functions for you on program start. We can't use that. We need to be able to unload and reload new DLLs on the fly. So we'll need to roll our own binding solution.

[FORWARD] The bare basics of what the binding system will need is to bind functions, both functions in any namespace and static class members.

Entire classes themselves also needed binding including all member functions and any virtual tables necessary.

Back when we started, D had support for these things. But the support is not really the kind you need for making a rapid prototyping system. It works out for the best to roll your own solution.

Match vtables

```
class SomeClassInterface
{
    virtual void SomeMethod() = 0;
    virtual void DoThis( int anIntParam ) = 0;
    virtual void DoThis( float aFloatParam ) = 0;
};

BIND_INTERFACE( 1, SomeClassInterface );
```

One of the built in mechanisms D provides for C++ interoperation is virtual function table matching. We used that in a few places. The mechanics we settled on were basically to define an abstract interface that a given class would inherit. And to ensure the system knows about it, you also have to use the BIND_INTERFACE macro.

So this is all well and dandy. We've got our normal methods, we've got overloads..

[FORWARD] Both the C++ and D compilers will do all the work for us and we can just get on with life.

[LOOK AT SCREEN]

Uh, Visual Studio? What are you doing there? And why have you got those angry eyebrows?

[FORWARD]

Match vtables

```
class SomeClassInterface
{
};

BIND_INTERFACE( 1, SomeClassInterface );
```

Your vtable is wrong!
It needs fixing!

My vtable is wrong? You what, mate?

[FORWARD]

Match vtables

```
class SomeClassInterface
{
    virtual void SomeMethod() = 0;
    virtual void DoThis( int anInt, float aFloat ) = 0;
    virtual void DoThis( float aFloat, int anInt ) = 0;
};

BIND_INTERFACE( 1, SomeClassInterface );
```

Wait, hold on. Did you just reverse my virtual function order for overloaded functions?

[FORWARD]

Match vtables

```
class SomeClassInterface
{

};

BIND_INTERFACE( 1, SomeClassInterface );
```

Much better! Is good now! kthxbai!

What? Better? How? Wait, get back here, Visual Studio!

[FORWARD]

So it turns out that visual studio, when it generates its vtables, reverses the order of any overloaded functions it finds. And since D compilers will generate vtables in the order it encounters those functions, you have a problem of the wrong function being called at the wrong time.

[FORWARD]

Match vtables

```
class SomeClassInterface
{
    virtual void SomeMethod() = 0;
    virtual void DoThisInt( int anIntParam ) = 0;
    virtual void DoThisFloat( float aFloatParam ) = 0;
};

BIND_INTERFACE( 1, SomeClassInterface );
```

To avoid this being a poorly-understood problem, I put out a blanket ban on overloading functions. Far from an elegant solution, but it meant that I could ensure consistent results across compilers.

[FORWARD]

Match vtables

```
@BindExport( 1 )
extern( C++ ) interface SomeClassInterface
{
    void SomeMethod();
    void DoThisInt( int anIntParam );
    void DoThisFloat( float aFloatParam );
};
```

This is how the same declaration would look in our D code. Nice and neat.

[FORWARD] And, importantly, we need to declare the class as extern(C++) or else it will generate the virtual function table in a D manner, which is not compatible with the C++ manner.

[FORWARD] Making it visible to the binding system is also done not with a macro, but by tagging the class with the BindExport user defined attribute.

User defined attributes are something I haven't really talked about yet. Sounds like a great time to do so.

Die C++, Die: Part Three

```
struct CustomUDA { }
struct ComplexUDA { int someValue; string someString = “Hi!”; }
@CustomUDA struct SomeNewStruct { … }
@ComplexUDA( 5, “Bye!” ) int someInt = 42;
@CustomUDA void imMisterMeeseeks() { writeln( “Look at me!” ); }
@FunctionStub void someFunctionStub();
```

User defined attributes are actually something that Manu requested and got added to the language. Everyone is using them extensively now. So there’s a good point - if you need something added to D and can make a good case to Walter and Andrei, then there’s a chance you’ll get it. There’s a proper process for these things today - D Improvement Proposals - but speaking to them face-to-face is still very much worthwhile.

But back to the business at hand. A user defined attribute is a pretty simple thing. How simple?

[FORWARD] It’s literally just a struct definition. If you know how to write a struct, you know how to make a user defined attribute.

[FORWARD] There’s no restriction on struct complexity. You can have default values and everything in it.

[FORWARD] Applying it to your type is simply a matter of pre-pending the declaration with an at sign followed by the name of your UDA.

[FORWARD] You can also apply UDAs to variables. This is quite handy.

[FORWARD] And also, you can apply it to functions. Also quite handy.

[FORWARD] Especially if you start tagging up function declarations with the intention of providing a definition later. Why is this useful? Well, let’s get down to business.

[FORWARD]

Die C++, Die: Part Three

```
foreach( MemberType; VariableTypesOf!T )

foreach( Variable; VariablesOf!T )

foreach( Function; FunctionsOf!T )
```

Remember in the last example how I illustrated going through the variable types of an object, and how it was possible to wrap that all up in a template? Well that's not all!

[FORWARD] We can also get all the actual variables!

[FORWARD] And all the declared functions!

And you can get carried away and collect subtypes of a given type, but that's going too far for our illustrative purposes.

[FORWARD]

Die C++, Die: Part Three

```
foreach( Function; FunctionsOf!T )
{
    static if( hasUDA!( Function, FunctionStub ) )
    {
        enum FunctionDef = generateStubFor!Function();

    }
}
```

What we're interested in here is getting all the declared functions of an object.

[FORWARD] And once we have them, we can iterate over them and check if the function has a FunctionStub user defined attribute.

The hasUDA template lives inside the standard library's traits module, and works on any symbol or type. If our function has the FunctionStub UDA attached with it, then sweet, our if check passes and we continue on.

[FORWARD] So here's an interesting one. generateStubFor is a templated function. And by assigning it to an enum, we're forcing it to run at compile time.

Die C++, Die: Part Three

```
string generateStubFor( alias Function )()
{
    return FunctionDeclaration!( Function )
        ~ " { writeln( \"I'm a function stub!\" ); }\n";
}

// Output will look like:
// void someFunction() { writeln( \"I'm a function stub!\" ); }\n
```

So let's have a quick look at the generateStubFor function. Its template parameter is an alias. Alias is a pretty powerful keyword in D. It can be used in a simple manner just like you'd use typedef in C or C++. In this case, we're using it to take a direct reference to a symbol.

The only other weird thing here is the FunctionDeclaration template we're invoking. The purpose of it will be to generate a string representation of the function as you'd see it in source code, and to do that it uses a few things from the standard library.

The end result will look just like a plain old ordinary function, but represented by a string. Cool. But now what do we do with it?

Die C++, Die: Part Three

```
foreach( Function; FunctionsOf!T )
{
    static if( hasUDA!( Function, FunctionStub ) )
    {
        enum FunctionDef = generateFromStub!Function();
        mixin( FunctionDef );
    }
}
```

Back over here, we'll have our function string calculated at compile time. And now, here's one of D's greatest magic tricks. Nothing up my sleeve, aaaaand:

[FORWARD] Ta-da! The mixin keyword. Now, everyone knows how to write C macros. And when you use them, you know that the preprocessor replaces the invokation of the macro with the expanded form. This mixin keyword? It's essentially the same functionality. Except it takes in a string. Any string. It can be a string literal. Or, in this case, it's a string we're generating at compile time to fill out a function stub. Let that sink in for a moment. Because there's more to mixins than just code generation.

Die C++, Die: Part Three

```
mixin template GenerateStubsFor( T )
{
    foreach( Function; FunctionsOf!T )
    {
        static if( hasUDA!( Function, FunctionStub ) )
            mixin( generateFromStub!Function() );
    }
}
```

There's another kind of template in D that you can write called a mixin template. Unlike a normal template, because it is a mixin it will act in a simiilar manner the C preprocessor and replace the invocation of the mixin with the contents.

In this case, our function iteration and generation code has in fact been sitting inside a mixin template this entire time.

[FORWARD] And yes, notice that you can go all Inception in here and have mixins within mixins.

But how is it all used? Well, that's quite simple really.

[FORWARD]

Die C++, Die: Part Three

```
struct SomeIncompleteStruct
{
    @FunctionStub void someFunction();
    @FunctionStub void someOtherFunction();

    mixin GenerateStubsFor!( typeof( this ) );
}
```

Here's an incomplete struct…

[FORWARD] And here's the invokation of our mixin.

The compiler will mixin the contents of the GenerateStubsFor template, which itself is mixing in compile-time generated code definitions for each function it finds marked with the FunctionStub user defined attribute.

[FORWARD]

Die C++, Die: Part Three

```
struct SomeIncompleteStruct
{
    @FunctionStub void someFunction();
    @FunctionStub void someOtherFunction();

    void someFunction() { writeln( “I’m a function stub!” ); }
    void someOtherFunction() { writeln( “I’m a function stub!” ); }
}
```

[FORWARD] And once the compiler has expanded everything, this is how your struct will look to the compiler. Everyting is expanded, and your stub functions will gladly print out to the screen that they’re function stubs when invoked. How nice of them!

[FORWARD]

Die C++, Die: Part 3

	User defined attributes	Deep function inspection	Mixins
Swift	**Runtime**	**Obj-C**	**No***
Rust	**Not yet**	**Maybe?**	**Macros**
C#	**Yes**	**Runtime**	**No**

The other, trendier languages have similar features for UDAs, but none of them are accessible at compile time in the same way as D. Rust allows you to define attributes from a pre-defined list.

Swift has mixins via traits and protocol extensions. While it's the same name, it's a different end result.

I'm certain I've seen how to inspect function types in Rust, but I can't find that info any more. Might have been in a plugin.

Anyone one of these features alone that I've shown over the course of this talk is a powerful thing for your language to have. In fact, if D wasn't a thing, maybe I'd be here talking about Rust now instead. D has all the features I've talked about out of the box, there's no need to install an additional library or build step or compiler plugin or anything to get access to these features. And having them all there, combined, waiting for you to use it? It's some next level shit.

Die C++, Die: Part Three

Programmers are lazy. It's literally our job description to tell a computer what to do so that we don't have to.

So let's write less code to do the same work.

Now, someone might be able to bang their head up against the C++ wall long enough and persistently enough to get something that might do what we've done here at compile time. But honestly: Why go to all that effort? Programmers are lazy by definition. If you can write less code to do the same job and have it still perform better than C++ compile times and not incur a runtime cost, **and** have it be readable to ordinary humans, then do that.

[45 MINUTE MARK?]

[CHECK QUESTIONS HERE]

[FORWARD]

Export C++ functions

```
BIND_CLASS( 1, namespace::SomeClass );
BIND_METHOD( 1, namespace::SomeClass, SomeMethod );
BIND_METHOD_OVERLOAD( 1, namespace::SomeClass, DoThis, void, int );
BIND_METHOD_OVERLOAD( 1, namespace::SomeClass, DoThis, void, float );
```

Alright, so getting back to the binding system.

Another critical thing D provides for C++ interoperations is function calling conventions. It knows all about standard calling conventions used in C, C++, Windows, and even Pascal. We can actually use this to give greater flexibility than built-in virtual function table matching.

[FORWARD] The first thing we needed to do was expose our C++ functions.

This was the method that we shipped Quantum Break with for the C++ code. Every time you wanted functionality, you'd go in and make these declarations yourself.

[FORWARD] And it's a bit of a Sisyphean Task.

Especially when it comes to exposing new functions and classes that you need. Getting non-polymorphic functions is easy enough, but then when you do encounter an overloaded function you need to specify the return types and all parameter types for the compiler to pick up the correct function you need.

[FORWARD]

Import C++ Functions

```
@CTypeName( “namespace::SomeClass” ) @BindClass( 1 ) struct SomeClass
{
    @BindImport( 1 ) void SomeMethod();
    @BindImport( 1 ) void SomeOtherMethod();
    @BindImport( 1 ) void DoSomething( int foo );
    @BindImport( 1 ) void DoSomething( float bar );
    mixin BindAllImports;
}
```

Now we get to the fun part.

D has the same distinction between struct and class that C# has - struct is a value type, class is a reference type. So if we want to replicate a C++ class as close as possible, we define a struct.

The struct, we mark up with some UDAs.

[FORWARD] The BindClass UDA is a lot like the C++ macro.

[FORWARD] The interesting one is the CTypeName UDA - we declare the fully qualified C++ name inside it. The why of this will come later.

[FORWARD] Now, if you remember the example with the FunctionStub UDA, you’ll see I’ve done something very similar here. We’re declaring functions and parameters, and marking them up with UDAs. It’s then up to this mixin down here to do all the heavy lifting we’re interested in.

[FORWARD]

Syntactic sugar

```
void namespace::SomeClass::DoSomething();


void namespace_someclass_DoSomething( namespace::SomeClass *const this );
```

Before I continue on the D side, there's something that needs highlighting on the C++ side. We have been exposing class methods to the binding system. The fully qualified name for the DoSomething method looks something like this. It looks just like a member function with no parameters, right?

Wrong.

[FORWARD] There's a secret parameter in that function call. The compiler, in fact, sees the function a little something like this.

"this" is actually the first parameter to each and every method function. So when we take the address of a function in C++, if we just jam it into a function pointer matching that invisible signature things will Just Work™.

[FORWARD]

Import C++ Functions

```
@BindImport( 1 ) void DoSomething( int foo );


void DoSomething( int foo ) { DoSomething_ptr( this, foo ); }


@BindRawImport( “namespace::SomeClass::DoSomething”, “void(int)”, 1 )
__gshared RawMethodPtr!( DoSomething ) DoSomething_ptr;
```

Subsequently, when we parse our class and generate bindings for our functions, that means we need to store the function pointer in a format that is correct for C++ but allows us to call it from D.

[FORWARD] The BindAllImports mixin will go and grab every BindImport tagged function and do thigns with it, and we’ll use this function here for illustrative purpose.

[FORWARD] The easiest thing it does is to implement the function. It inlines it since it is doing little more than calling a function pointer with all parameters and this as the first parameter.

[FORWARD] The pointer itself is stored as a globally shared variable. In D, static only has thread-local storage. So if you want something to be accessable across all threads, it needs to use gshared storage.

[FORWARD] Notable here is that we have attached a BindRawImport user defined attribute to our pointer. Everything up to this point has served to provide a human-friendly abstraction. When we import function pointers

from C++, we import them directly in to function pointers marked with BindRawImport. And inside each BindRawImport is all the information it needs to find the correct function with the correct prototype.

[FORWARD]

Import C++ Functions

```
import std.traits;
import std.typetuple;

alias ParamTypes = TypeTuple!( ThisType*, Parameters!( Method ) );
alias ReturnType = ReturnType!( Method );
alias RawMethodPtr = ReturnType function( ParamTypes );

pragma( msg, RawMethodPtr.stringof ); // void function(SomeClass*, int)
```

Getting the RawMethodPtr template to work is really quite simple thanks to a couple of things in the standard library.

[FORWARD] The Parameters template there will return a type tuple of the parameter types of our function, and we can make a new one by feeding a type tuple another type tuple.

[FORWARD] Getting the return type is also quite easy, and self explanatory.

[FORWARD] We then make an eponymous template definition to a function pointer defined with the determined return type and our newly constructed paramters.

[FORWARD] And finally, for a bit of sanity checking, if we were to print out the type of our function pointer - each type in D has a stringof member that you can query for a string representation of it...

[FORWARD] Our output would look exactly like this in the compile log.

[FORWARD]

Putting the Pieces Together

- Load a D DLL from C++ code
- Call a function with a collection of exported objects
- Stick all those pointers in D equivalents
- Get on with life

And there's really not much more to do from this point.

[FORWARD] It really is just as simple as loading a DLL with LoadLibrary, resolving a function and calling it with a collection of exported objects, stick all those pointers you now have in to your D equivalent function pointers, and get on with your life programming in a nicer language.

Instantiate from C++...

```
DClass< SomeDClassInterface > pInstance;
pInstance.instantiate( “remedy.testobjects.SomeDClass” );

pInstance->doATask( pSomeBoundCPPObject );
```

And that’s the basics of the system we shipped Quantum Break with. It meant that in C++ code, we could instantiate an object in D with a C++ interface and speak to it.

[FORWARD]

...And Use D Code

```
class SomeDClass : SomeDClassInterface
{
    void doATask( SomeClass* pClass )
    {
        pClass.SomeMethod();
        pClass.DoSomething( 42 );
    }
}
```

And inside D, because we'd gone to all the trouble of binding up methods, it meant we can pass instances of C++ classes straight to D and treat it exactly like it was a D object and write plain old ordinary D code with it. Sweet. We can write as much D code as we want, and as long as we've exposed our C++ functionality there's no technical disadvantage to doing so.

[FORWARD]

What About Hot Reloading?

- VariablesOf!(Type)
  - Write to JSON
- Delete, unload, load new code, recreate
  - Free choice of debug and release code :-D
- VariablesOf!(Type)
  - Locate and read from JSON

But what about hot reloading? You can't just reload a DLL and expect the data to be just fine, especially if you change the layout of your D objects. It has quite a simple solution actually.

[FORWARD] We already know how to get all the variables of a given type.

[FORWARD] So doing that, we write the name and a string representation of the value to JSON. The JSON parser in the standard library can do this for you, but we need to do things it doesn't handle like scanning for pointers so that we only serialise objects once.

[FORWARD] And then it's out with the old, in with the new ---- We delete all of our D objects after serialisation, unload the DLL, load in the new DLL, and recreate all our D objects

[FORWARD] And hey, since we're going to the trouble of reloading a DLL, do you want a release or a debug build? Now, it needs to be said, being able to selectively switch between debug and release on the fly like this changes the way you work.

[FORWARD] But whichever one you choose to load, the deserialisation is the same - with your recreated objects, do another pass over all the variables...

[FORWARD] And read them from JSON if they exist in the JSON document. Being a JSON representation means we don’t have to worry one iota about binary matching. If a variable is there, then it will deserialise. If it is not, then I hope you’ve put in meaningful default values.

It got use in a rather critical subsystem too. We used D to communicate between character inputs - be that AI or player driven character inputs - and our animation middleware, Morpheme. It was originally all C++ code, but that was terrible for workflow purposes as code and data needed to be heavily in sync for everything to just work. I ported all that code to D, and now if you try to use a character without that D code it just stands there in its idle loop.

Of course, there were issues with the binding system. Some of these, I've highlighted and said we're improving. In fact, what started as a simple clean up so that humans could read the code has turned into a full on reengineering of the system.

[FORWARD]

We're open sourcing!

Binderoo

C++ binding layer
Rapid iteration development

And we're open sourcing it. We're calling it binderoo. It's on github. It's currently a work-in-progress, but it's going to improve. Now that it's open, Manu will be working on it once again so there'll be two of us constantly updating and improving it.

[FORWARD]

Binderoo current/planned features

- Code edit and continue
- Code object binary layout modification
- Switch between debug/release on the fly
- Function versioning, mismatch mitigation
- Dynamic virtual function tables
- Unity-style code updating
- Auto-bind a C++ codebase

Some of these features are already in the code. Others still need varying degrees of work to get there. But I've already got some interesting stuff.

For example, did anyone wonder throughout the talk what all those numbers were doing in the binding declarations? They were actually version numbers. Quantum Break shipped with hard version checking, which solved DLL hell but also meant we had to follow a rather convoluted process to submit C++ changes that D code relied on. The new system requires you to define the version that the function was introduced in - and if necessary, the version the function was removed in. The binding system can then make decisions on what to import, and you can write branching code to handle it based on what version was imported.

The *really* fun part with the new way I'm handling versioning is that I also completely sidestep the compiler and rebuild virtual function tables for any given object based on the version that's being imported. This itself solves a very big problem we had with matching interfaces - the restriction on overloaded functions no longer exists, but it also means that since the vtable is provided by software and not the compiler that I can

build one up for the version you've actually loaded in.

The rapid iteration functionality will have quite a bit of attention focused on it. I'm currently working on showing that we can replace our level scripting language with D. That scripting language has served several games, but it lacks certain modern features that our scripters want these days. Like loops, for example. Our level builders are highly skilled these days - one of them releases award-nominated indie games in his spare time, another one was on the Oscar-winning visual effects team for the movie Gravity. They can handle and want a real language, so it's just a matter of making the workflow quick enough and easy enough for them (the goal is sub-second iteration times). And this will have knock-on effects for the core programmers too as their D code will be handled by the same backend system.

[FORWARD]

Binderoo

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

You can go get the in-progress work now. It's not really documented, but it's there. As it gets more feature complete, the documentation will be worked on so that anyone can use it in as close to a drag-and-drop deployment as I can possibly get it.

[QUESTIONS?]

Learning D

- The D Programming Language by Andrei Alexandrescu
- Learning D by Michael Parker
- Programming in D by Ali Çehreli
- The D Cookbook by Adam D Ruppe
- Ask on https://forum.dlang.org
- irc://irc.freenode.net/#d

Before everyone goes. If you want to learn D there's some very solid resources out there. I've already mentioned Andrei's book. Learning D is aimed at people familiar with C. Programming in D is aimed at people new to programming in general. The D Cookbook there has some fairly solid examples of more advanced stuff you can do in D. And there's always the newsgroups which can be accessed via forum.dlang.org. And if you're an old timey kind of person, the official IRC channel is always active.

On a personal note. The first time I programmed in D was for an internal 48 hour game jam a month after I got to Remedy. I found it very easy to pick up at a basic level thanks to its similarities to C#, but there are some key differences that will bite you if you don't pay attention. Everything I've presented in this talk about compile-time code, I had to pick up *very* quickly. One of the programmers I previously mentioned, Manu Evans, was in charge of the system before I got to Remedy. But not long after, he decided to leave. And as I've known him since 2002 and could quiz him quite easily if I needed help, and since I found it interesting, I took over and finished the prototype he started. All I had to go on was his explanation of the code, and the code itself. Everything I've

presented here, I had to learn the hard way. But hopefully it’s clear enough that you were able to follow what was going on.

We’re hiring!

http://www.remedygames.com/careers/

Also, come work with us. We actually have cookies!