C++ Introspection

DigiPen Game Engine Architecture Club
Sean Middleditch

About the Speaker

Sean Middleditch <sean@seanmiddleditch.com>

Engine Architecture Nerd

DigiPen BSCSRTIS Senior: Subsonic, Core

Professional: SONAR, Gas Powered Games

About the Talk

C++ introspection mechanics geared toward understanding the sample code posted online

Advanced topic geared for expert C++ users (sorry underclassmen), not expected to be clear after one talk

Feel free to ask questions, I expect (and welcome) them in this talk

Sample Code URL

http://tinyurl.com/cootluj

Or search GitHub for 'dpengineclub' on user 'seanmiddleditch', the code is under the 2013/Introspection-2 folder.

Introspection Recap

Previously on Game Engine Architecture...

What It Is

Type introspection is the ability of a program to examine the type or properties of an object at runtime. - WikiPedia

Let's code make decisions at runtime that would normally only be doable at compile time, such as "call method Foo() on object Bar"

Implementation

Database (map) of type names to type descriptors

Database of names to properties and methods on those type descriptors

In C, based almost entirely on the memory offset of a struct member variable from the base of that struct

Method Binding in C

Previously, we avoided it

No support in C for deducing parameter/return types or automatically generating thunks (wrappers) for methods to a runtime-friendly calling convention

C++ solves both problems

Advantages of C++

... relevant to introspection

Type Deduction

With C, it was necessary to respecify the type of a bound member variable

With C++, meta-programming magic can figure out the type of any member variable specified by name or even the type of return value from a getter function

Template Code Generation

Templated functions can be automatically generated to wrap up any C++ function, allowing the automatic conversion of runtime-friendly object wrappers to native function parameter types and wrapping up the return value

Only necessary to specify a function by name, no need to respecify the whole signature

Meta-Programming

Meta-programming means writing code that changes behavior at compile time based on types passed in, allowing a single piece of code to be written that can handle both a string or a SIMD vector

Slightly Better Inheritance

The C-based object inheritance of the last talk are not necessary in C++ since it natively supports inheritance, albeit a model that's harder to work with than you might think

With some work the introspection system can handle single inheritance, multiple inheritance, even virtual inheritance (though this talk won't cover virtual inheritance because it's awful)

Bigger Standard Library

The example introspection system is more fully featured simply because C++ include `std::string`, `std::vector`, and other useful types that ease implementation

Template Type Specialization

Types can be made which change their behavior based on *template specialization*

Allows `GetMeta<Foo>()` to do something different than `GetMeta<Bar>()` at compile time

Namely, return a pointer to a different type info object

C++ Introspection Features & Overview

Examples of what this is all about

Example Introspection Declaration

```
// Enemy.h -- C++ class modified for introspection
class Enemy : public Actor {
  META_DECLARE(Foo);
  int m_Health;
public:
  int GetHealth() const { return m_Health; }
  void SetHealth(int hp) { m_Health = hp; }
  void Damage(int dmg)
    { m_Health = std::max(0, m_Health - dmg); }
};
```

Example Introspection Declaration

```
// Enemy.cpp
#include "Enemy.h"

META_DEFINE(Enemy)
.base_class<Actor>()
.member("health", &Enemy::GetHealth, &Enemy::SetHealth)
.method("Damage", &Enemy::Damage);
```

You could use macros to remove the redundant type specifiers if you want; they help with Intellisense, though

Type Info

Just like in C, every type must have a `TypeInfo` object that describes it

This object is not a template since we want a single well-known type/interface for querying object capabilities

It holds a type name, base types, member variable descriptors, and method descriptors

Member Variable Info

Member variables can come in three types

Templates and specialization must be used; the `MemberInfo` type for a string cannot be used for an integer

Likewise, the `MemberInfo` for a variable accessed directly can't be used for variables accessed by getter/setter methods

Three Supported Member Variable Types in the Example

First are those accessed directly in memory, like C struct members

Second are properties accessed by a getter method and setter method

Third are read-only properties that have a getter but no setter

Member Functions

Member functions are also supported

Need a difference `MethodInfo` template for each number of arguments (until we all have good C++11 variadic template support) and also to easily handle void vs non-void return values

Base Types

Inheritance is complicated

Multiple base types allowed in C++

Converting a pointer from a derived type to a base type might involve adjusting the pointer value

Not technically legal the way I do it, but meh

Type Builder

The syntax presented earlier is done via a type builder template that allows *method chaining* (each method call returns a reference to the original object)

Some trickery allows you to bind *private member variables and function* - I'm undecided if that's a good thing or not, but it's a cool trick

Properties vs Member Variables

By allowing the use of getter and setter methods, *virtual* properties can be defined

That means named properties that don't actually exist in memory but are the result of a calculation

Example might be a `textureName` property that when set loads a new texture rather than just changing some string in memory

Member Variable & Function Binding

Now the fun stuff

Member Variable Pointers

Recall that a pointer can be specified to a member variable on a particular type

```
struct Foo { int i; };
int Foo::*pi; // pointer named pi to int on class Enemy
pi = &Foo::i; // take address of member variable
Foo foo;
foo.*i = 17; // dereference pi
```

Template Deduction

Functions can use a template with *template type deduction* to make any pointer-to-member work

```
template <typename ClassType, typename MemberType>
void bind(MemberType ClassType::*member)
{ /* ... */ }

struct Foo { int i; }

void test() {
  bind(&Foo::i); // ClassType = Foo, MemberType = int
}
```

Member Function Pointers

Very similar to member variable pointers

Parameter types, return value, and type qualifiers (like constness) all matter!

```
struct Foo { int bar(float) const { return 3; } };

// yes this is ugly, should look similar-ish
// to C function pointer syntax
float (Foo::*method)(float) const = &Foo::bar;

Foo foo;
// extra parenthesis needed or this won't compile!
int ret = (foo.*method)(17.5f);
```

Template Type Deduction Again

You can also use template type deduction for methods (or any function), but only for a set *arity* (number of arguments) until C++11

```
template <typename ClassType, typename RetType,
  typename Arg0Type, typename Arg1Type>
void bind(RetType (ClassType::*method)(Arg0Type,
Arg1Type))
{ /* ... */ }

struct Foo { int bar(float, char) { return 3; } };

void test() {
  bind(&Foo::bar); // ClassType = Foo, MemberType = int,
                   // RetType = int, Arg0Type = float
                   // Arg1Type = char
}
```

Method Explosion

You need a different `bind` for each arity of method

You need a different `bind` for `const` methods, `volatile` methods, `const-volatile` methods, and regular methods (you probably don't need to worry about `volatile` though)

C++11 makes this better, but Visual Studio 2012 SP1 doesn't support them

More Method Explosion

Generated wrapper code will want to return values, likely with code similar to this:

```
return (obj->*method)(args);
```

Illegal if the bound method returns void

Specializations are needed for `void` vs non-`void` return values for every method binder, too

Method Mega-splosion

Const and non-const, void and non-void (4 variations) needed for every arity (you'll probably want to support arity of 3-5, maybe higher)

That's 12-20 templates that must be written

Thankfully they're all small and simple, but still a pain

Member Variable Getter/Setter

Getters and setters for properties are easier than general methods

Mandate that all getters are `const`

All setters must be non-`const` and return values are ignored

Intermediate Templates

Simpler templates can be used for the public API which call into more complicated templates for implementation

For example, binding a member getter has to manipulate the getter slightly in most cases, so have a public API binder that calls into some specialized helpers

Binding a Getter Part 1

```
template <typename ClassType, typename MemberType,
  typename ReturnType>
class TypedMemberInfo : public MemberInfo
{
  ReturnType (ClassType::*m_Getter)() const
  MemberInfo(const char* name,
    ReturnType (ClassType::*getter)() const) :
    MemberInfo(name), m_Getter(getter) { }

  virtual Any Get(const void* object) {
    return make_any<MemberType>(
      (static_cast<const ClassType*>(object)-
>*m_Getter)());
  }
};
```

Binding a Getter Part 2

```
template <typename ClassType, typename ReturnType>
MemberInfo* bind(const char* name,
  ReturnType (ClassType::*getter) const
{
  return new TypedMemberInfo<ClassType,
    typename std::remove_const<
      typename std::remove_reference<ReturnType>::type>
    >::type, ReturnType (ClassType::*)() const>
    (name, getter);
}
```

The `std::remove_*` turn `int const&` into just `int`, which is what you will want later

Type Trait Macros

Note that a more complete `MemberInfo` will also store a pointer to the `TypeInfo` to the member's type for lookup later if needed

The types of `int const&` and `int` are different, but what we really care about is that the type is `int`

Strip off reference, pointer, const, and so on when building member/method metadata

Template Specialization for Void

Methods that return `void` are a pain

Specializations allow a `void` version of method calls to be written

Only types can be specialized in C++03 (and MSVC 2012), so method calling should be moved into a helper struct to work around this limitation

Helper Struct Specialization

```
template <typename ClassType, typename RetType>
struct call0 { static Any call(void* object
    RetType (ClassType::*method)()) {
  return make_any<RetType>(
    (static_cast<ClassType*>(object)->method)());
} };

template <typename ClassType>
struct call0<ClassType, void> { static Any call(
    void* object, void (ClassType::*method()) {
  (static_cast<ClassType*>(object)->method)());
  return Any();
} };
```

Helper Structs

Remember, you will need a `call0`, `call1`, `call2`, etc.

Also, overrides for `const` vs non-`const` methods

Call these helpers from inside a `MemberInfo` class's overridden `Call` method

Simpler Deduction When Possible

In some of your intermediate code, you won't care about the specifics of a method signature, because the method pointer will be passed to a helper that specializes or overloads as necessary

Only spell out signature types when you actually need them

Simple Deduction Example

```
template <typename ClassType, typename RetType>
MethodInfo* _bind(RetType (ClassType::*method)());

template <typename ClassType, typename RetType,
  typename ArgType>
MethodInfo* _bind(RetType (ClassType::*method)(ArgType));

// notice this doesn't care about method signature
// expand to have any common code, like registration
template <typename Method>
MethodInfo* bind(Method method)
{ return _bind(method); }
```

Binding Your Privates For Public Abuse

Most immature slide title you'll see here

Private Context

Private (and protected) declarations are only available in some contexts

Inside of methods of the class are one place privates can be accessed

They can also be accessed in the *initialization* of the class or a nested member type!

Nested Pointer Initialization

```
class PrivateStuff {
  int my_eyes_only;
  struct holder {
    int PrivateStuff::*spi;
    holder(int PrivateStuff::*spi) : spi(spi) { }
  };
};

// illegal global declaration, invalid access level
int PrivateStuff::*gpi = &PrivateStuff::i;

// legal declaration!
PrivateStuff::holder leak(&PrivateStuff::my_eyes_only);
```

Abusing The Trick

If the special `TypeBuilder` type used to construct the `TypeInfo` for a class is declared as a static inner type of the class, then our pretty meta declaration syntax is allowed to bind private members!

This only works if you can modify the class to contain that special inner class, which sometimes you can't (or won't want to) do

Meta Lookup

How to actually find the TypeInfo for a class

Run-time Lookup

The runtime case is actually easy

After building a `TypeInfo`, insert it into a `std::map` or table, and then do a simple search on the type's name to find its `TypeInfo`

Once you have the `TypeBuilder` support working, this is basically free (ironically, I did not implement this in the sample code...)

Compile-time Lookup

There are actually multiple ways to do this... and we want most of them

We need to look up info on a type statically, on a type polymorphically, and for external types that we can't/won't modify to insert extra data

Static Lookup Try 1

The easiest way to do static lookup is to insert a static method into a type that returns the corresponding `TypeInfo*`

```
class Foo {
  static const TypeInfo s_TypeInfo;
public:
  static const TypeInfo* MyType() { return &s_TypeInfo; }
};

const TypeInfo* foo_type = Foo::MyType();
```

Wrap in macros to make it easy to use

Problems

The previous approach assumed that every class will be able to have a `MyType()` static method added to it, which is not always possible

We need a way to store and retrieve a `TypeInfo` for a type externally but be able to look it up at compile-time

The global name trick from C plays poorly with namespaces, so it's a no-go

Mo' Templates

The C++ One Definition Rule states that there is a single instance of every function/global defined, including for templates

`std::vector<int>::push_back()` might be instantiated in multiple translation units, but after linking, there will be only one copy

The same is true for static variables in templates

Template TypeInfo Holder

A template that hold a `TypeInfo` for a type can be made

```
template <typename Type>
struct TypeHolder {
  static TypeInfo s_TypeInfo;
  static const TypeInfo* Get() { return *s_TypeInfo; }
};

// you must still define storage for template statics
// usually done as part of a META_DEFINE_EXTERNAL macro
TypeInfo::TypeHolder<int> s_TypeInfo = /* builder */;

// and this will always return the address of the same one
// no matter which translation unit this is called from
const TypeInfo* int_type = TypeHolder<int>::Get();
```

Reconciling

You aren't going to want to remember how to use both meta lookup systems and you want to wrap them up in a single macro/API anyhow

Now is time for gross meta-programming hacks

Doable in C++03 but I'm lazy and did it with C++11 which is way easier

Meta-Programming

First piece required is the ability to test if a type has a static `MyType()` or not (not implied that `TypeHolder` must be queried)

Second piece is to select a piece of code to invoke based on that test inside of a simple API, which is doable with template specializations (which can only be types in C++03 and need wrappers to be used from functions)

Piece 1 C++11 style

```
// replace use of decltype and std::is_same with sizeof
// and struct <size_t S> empty {}; to make this C++03
template <typename T>
struct has_mytype {
  // these must be templated _again_ because of how
  // SFINAE works
  template <typename U> char test(decltype(U::MyType())*);
  template <typename U> int test(...);

  // in C++03 this must be: enum { value = ... };
  static const bool value =
    std::is_same<test(static_cast<T*>(nullptr)),
    char>::value;
};
```

Piece 2 Page 1

```
// called when HasMyType is true
template <typename Type, bool HasMyType>
struct _get_meta {
  static const TypeInfo* Get() { return Type::MyType(); }
};

// called when HasMyType is false
template <typename Type>
struct _get_meta<Type, false> {
  static const TypeInfo* Get() { return
    TypeHolder<Type>::Get();
};
```

Piece 2 Page 1

```
// this helper function selects which _get_meta to call
template <typename Type>
const TypeInfo* get_meta() {
  return _get_meta<Type, has_mytype<Type>::value>::Get();
}
```

`has_mytype<>::value` is a compile-time constant value and is used to select the specialization of `_get_meta`, noting that if true it will select the unspecialized version, and for false we provided a specialization for using `MetaHolder`

Polymorphic Lookup

If given a pointer to a polymorphic base class, it's really handy to find the type of the actual instance being referenced

Requires a virtual method to do; this only works if you can modify the type to add this method

Again easily added to any type with a macro

Magic Lookup

As reader exercise (if not looking at sample code), make a `get_meta` that can take a pointer to a type and, if that type supports the virtual polymorphic type lookup, use that; otherwise, use the previous static type lookup code

Should be trivial to implement if you understand the static type lookup code

TypeBuilder

Easily constructing a TypeInfo

TypeBuilder's Purpose

Assuming there's a static `TypeInfo` member on a type, how do we initialize it?

It's the same as any other static/global variable

Using method chaining and a helper class, we can get the nice syntax shown earlier

Initialization

```
// header
struct Foo {
  META_DECLARE(Foo); // declares statics
  ...
};

// cpp file
TypeInfo Foo::s_TypeInfo;
TypeBuilder<Foo> Foo::s_TypeBuilder(
  "Foo", &Foo::s_TypeInfo) /* more here */;
```

TypeBuilder... Builds Types

The idea is the the `TypeBuilder` type has methods on it that modify a `TypeInfo`

Make the `TypeBuilder` take a pointer to a `TypeInfo`, or be convertible to one, whichever, so long as the instance of `TypeBuilder` is constructed in the context of a static inner variable of the base type so the private trick works.

TypeBuilder

```
template <typename Type>
struct TypeBuilder {
  TypeInfo* m_Type;
  TypeBuilder(const char* name, TypeInfo* ti) : m_Type(ti)
  { ti->SetName(name); }
  // bind a direct member variable; must return *this
  template <typename MemberType>
  TypeBuilder& member(const char* name,
      MemberType Type::*mptr) {
    ti->AddMember(name,
       new TypedMember<Type, MemberType>(mptr));
    return *this;
  }
};
```

Method Chaining

Each method on `TypeBuilder` must return a reference to itself

That allows method chaining to work

Otherwise, you could only call `member()` once.

```
TypeBuiler<Foo>("Foo")
.member("a", &Foo::a)
.member("b", &Foo::b)
.member("c", &Foo::c);
```

TypeBuilder Complexities

`TypeBuilder` needs methods to handle adding bases, adding direct members, members with getters and setters, and adding method

Using the simplified type trick with helpers you can keep `TypeBuilder` mildly simple, but it'll still need a fair number of overloads

Sample code does not handle many variations (e.g. no `const` methods other than getters)

Bases

Harder than you think

C Recap

Last time we said that in C and simple C++, the following declarations of B were (usually, in most implementations) equivalent in memory layout and interchangeable for purposes of introspection

```
struct A { int a; };

struct B : A { int b; };
struct B { A base; int b; };
struct B { int a; int b; };

B b;
A* pa = (A*)&b;
```

Multiple Inheritance

Expand the previous example to C++ multiple inheritance

```
struct A { int a; };
struct B { int b; };

struct C : public A, public B { int c; };
struct C { A base1; B base2; int c; };
struct C { int a; int b; int c; };

C c;
A* pa = (A*)&c;
// oops, the following is wrong!
B* pb = (B*)&c;
```

Hackily Fixing Multiple Inheritance

Fortunately, if we make the same assumptions about memory layout, we *can* do this

```
struct A { int a; };
struct B { int b; };

struct C : public A, public B { int c; };
struct C { A base1; B base2; int c; };
struct C { int a; int b; int c; };

C c;
A* pa = (A*)&c;
// if we adjust for the offset of B in C, this will work
// (on most compilers/platforms we care about)
B* pb = (B*)((char*)&c + sizeof(A));
```

Properly Fixing Multiple Inheritance

The more correct "works everywhere" solution is to use a template that wraps `static_cast` from a derived type to any of its bases

Sample code does *not* do this because I'm lazy and don't currently care about the "weirder" compilers/platforms, but I'm sure it'll be relevant to some of you at some point

Supporting Multiple Inheritance

In some places it will be important to cast a derived type to its base, or maybe even the other way around

A type cannot just know which bases it is compatible with but also how to adjust a pointer to the derived type to its base

`TypeInfo` stores bases as a pair of adjustments and the base's `TypeInfo` pointer

Base Table

```
std::vector<std::pair<std::ptrdiff_t, const TypeInfo*>>
m_Bases;
// this handles recursive types, returns nullptr if
// this type does not derive from the given base
void* AdjustToBase(void* obj, const TypeInfo* base) {
  if (base == this)
    return obj;
  for (auto& bi : m_Bases) {
    void* adjusted = bi.second->AdjustTo(
      (char*)obj + bi.first, base)
    if (adjust != nullptr)
      return adjust;
  return nullptr;
}
```

Any Type

Runtime calling convention

Type Problem

`Enemy::Damage()` is a bound method taking an `int`

`MethodInfo::Call()` must be a generic method that works for any bound method, whether it takes no parameters, one `int`, or four `float`s, or whatever

Calling Convention

The calling convention is a feature of the platform's ABI that defines how arguments are passed to a function, such as how they are put on the stack, how registers are used, who cleans up, etc.

Projects like libffi support "native" calling conventions but require a lot of work to setup and use correctly

Scripting and Runtime Requirements

At runtime, you may just have a JSON file that says `"InitializePosition": [3.5, 7, 4.9]`

Most scripting languages call native code functions only by passing in an array of some special wrapper type (or have an implicit stack of script types like Lua)

Converting Calling Convetion

You need a `MethodInfo::Call()` that can take some array of generic types and make an actual call to the bound method in native C++ code to honor the calling convention properly

The type of this array can be an `Any`[] where `Any` is a special type that is made to hold "any" type

`boost::any` but with `TypeInfo` supported

Pointers

Making an `Any` that only supports pointers is easy

Store a `void*` to the object and the `TypeInfo*` for the object

Add a templated convert that "does the right thing"

Pointer-Only Any

```
class Any {
  void* m_Ptr;
  const TypeInfo* m_Type;

public:
  template <typename Type>
  Any(Type* obj) : m_Ptr(obj), m_Type(get_meta<Type>::
get())
  { }

  template <typename Type>
  Type* GetPointer() {
    return m_Type->AdjustToBase(m_Ptr, m_Type);
  }
};
```

Any Notes

The GetPointer<>() method must remember to adjust, because we might be converted to a base type that needs adjustment to work

Using this for method arguments or member binding is easy

Method with Any Arguments

```
template <typename ClassType, typename Arg0Type>
struct MemberInfo0 : public MemberInfo {
  virtual void Call(void* obj, Any* argv) {
    (static_cast<ClassType*>(obj)->*m_Method)
       (*argv[0]->GetPointer<
         typename std::remove_reference<Arg0Type>::type
       >());
  } };
```

This is where remembering `remove_reference<>` and friends is helpful, so the right type is passed to `GetPointer()` even if `Arg0Type` is a reference

Pointer-Only Any Not Enough

```
int GetHealth() const;
```

Can't just store the return value of that in a pointer-any `Any`, since the return value is a temporary

Need to be able to copy small values into `Any` for returning

Versatile Any

You can stuff small types into the space of a `void*`

If you only support very small POD types, this is all you need

For larger POD types, use a union between the pointer and a `char` buffer of the desired size

Versatile Any Pointer

Store a flag indicating if the pointer is stored or a value is stored

`GetPointer()` should return the address of the stored value (e.g. the address of the pointer or the buffer)

This way you have a consistent pointer-based API to access values while construction can be done using a pointer or value

Any References

Treat references like pointers

Remember that templates, a value will always be treated as a reference if deduced, so use explicit types and a helper for return values from methods to detect actual reference return values and purely by-value return values

References-as-pointers is not legal C++, but it works everywhere you probably care about

Complex Any Values

Storing non-POD types is trickier

You must remember to call the destructor on the stored type and must know how to copy/move real classes since `memcpy` won't be sufficient

Many blog posts and articles on how to implement more complete `Any` types like this

Miscellaneous Notes

Stuff to consider

Any Simplifications

The `Any` type in the sample code can be simplified a lot

Type traits stored like the `Destructor` and `Mover` in the `TypeInfo` of the type, since those can be useful elsewhere

These traits could be specialized away for POD types and the like

Attributes

You will likely need more information on types, member variables, and methods, like editor types, user-friendly names, network sync flags, etc.

You can use an additional chain level or pretty syntax for this

Attributes Examples

```
// kind of pretty, must grok comma operator overloading
META_DEFINE(Foo)[
  EditNameAttribute("A Foo"),
  IsComponentAttribute()
]
.member("bar", &Foo::bar)[
  EditIntRange(10, 1000)
];
// vs the following needing advanced sub-type chaining
META_DEFINE(Object)
  .pretty_name("An Object")
.member("id", &Object::id)
  .read_only().pretty_name("Unique Identifier")
.method("Clone", &Object::Clone);
```

Type Special Operators

The introspection system can use meta-programming during type constructing to automatically bind the type's destructor, copy constructor, move constructor, and other operations to make them available to anyone who wants them, which can be handy

Script Integration

Instead of or in addition to a custom `Any` object it may be much simpler to convert to/from script types like `Py_Object` or the Lua stack

Of course, this requires the introspection system to marshal non-primitive types back and forth

Helpful to this, automatically bind `ToScript` and `FromScript` operations when building a type

True Portability

In a few places I mentioned reliance on a particular behavior from the compiler that is not guaranteed by the standard

All of these can be "fixed," though sometimes at a cost of greater implementation complexity or even worse runtime performance

Emscripten? Should work with the "bad" code, hasn't been tested though

Macros

I left out examples of where and how to use macros where appropriate

Similar to last talk, should be easy to figure out for anyone grokking the template issues

Mostly just need macros to declare the `TypeInfo` holders and start the definition, see sample code

Optimization

Talk slides and sample code don't make any attempt to optimize in a few key places

For example, a type could contain a list of all children types, make it easy to find all `IComponent` derived classes for example

Be wary of initialization order! No guarantee that `Base` will have its `TypeInfo` created before `Derived` has its created!

Type Registry

The sample code leaves out the rather important piece of the type registry

When a `TypeInfo` is created, add it to a map

Again, beware initialization order! Your `TypeRegister` may be uninitialized when the first `TypeInfo` is created! Recall that statics are guaranteed zero-initialized, use pointers and null checks to manually initialize your registry

Visual Studio Linking

There are some quirks with linking

Visual Studio will throw away any translation unit which has no referenced symbols, including translation units that invoke constructors of otherwise unreferenced-objects

Avoid making a .cpp file that has nothing but introspection metadata, it may get thrown out silently and then you get weird linker errors

Sample Code URL (Again)

http://tinyurl.com/cootluj

Or search GitHub for 'dpengineclub' on user 'seanmiddleditch', the code is under the 2013/Introspection-2 folder.

Sample Code Legality

The sample code is ***not*** licensed for your use

It is ***academic plagiarism*** to use it in your project

It is an illegal ***copyright violation*** to use it in your project

Don't use it in your project, period

Questions?

Ask away