Introduction to Introspection in C

Sean Middleditch
DigiPen Game Engine Architecture Club

About the Presenter

Sean Middleditch

DigiPen BSCSRTIS Senior

20 years programming experience

Architecture nerd

About This Presentation

Introspection, metadata and its uses

Pure C implementation

Basics only

Introspection Uses

What is it good for?

Introspection

C/C++ lacks any native form of compile-time or run-time introspection

Introspection allows tools or code to make decisions based on structure of code

Removes the need to write the same structures over and over for different uses (serialization, networking, editor GUIs, debug tools, etc.)

Debugging

Debug output wants to print values from a data structure

Normally requires writing tons of custom code to display each structure

Introspection removes all the redundancy, debug code can find all readable members of a structure and their types

Editing

Very similar to debugging

Requires identifying writable members

Networking

Often better served at a different layer without introspection

For some architectures, using introspection to figure out which components or properties (structure members) to synchronize is easiest

Serialization

Saving and loading game data again requires a lot of custom code

Often similar if not identical to editor support

Need to enumerate structure members and types, with names

Need to map incoming member names to members

Structure Member Introspection

Necessary data

C Structures

```
struct PhysicsBody {
  float mass;
  float inverse_mass;
  vec3 center_of_mass;
  vec3 velocity;
  vec3 acceleration;
  vec3 force;
};
```

Each structure member has a *type* and and *offset*, in addition to a logical *name*

The "center of mass" member has type `vec3` and an offset of **8 bytes**

offsetof **Macro**

Use the `stddef.h` (or `cstddef` in C++) to get the `offsetof` macro

Pass it the *name of the structure* as the first member, then the *name of the member*

Returns the *offset in bytes* of the member

Member must be accessible from calling context (important for C++ users)

Member Names

There's no way to extract the names of structure members from code in the compiler itself

Often desirable to have a different or multiple names/identifiers anyway (XML/JSON element names, user-visible editor names, unique integer IDs for networking or binary formats)

Must be specified a second time in a new macro or function when defining metadata

Offsetof Example

```
struct property {
  const char* name;
  ptrdiff_t offset;
  metatype* type;
};

property p_mass;
p_mass.name = "mass";
p_mass.offset = offsetof(PhysicsBody, mass);
p_mass.type = &metatype_float;
```

Macros

Saving a ton of typing

Use of Macros

Building a list of properties for each class would be tedious if done all by hand

Simple macros can simplify the code

Remove the redundancy wherever possible

Macro Example

```
#define META_PROPERTY(base, name, type) \
  p__##base##__##type.name = #name; \
  p__##base##__##type.offset = offsetof(base, type); \
  p__##base##__##type.type = &t__##type;

META_PROPERTY(PhysicsBody, mass, float);
META_PROPERTY(PhysicsBody, inverse_mass, float);
META_PROPERTY(PhysicsBody, center_of_mass, vec3);
META_PROPERTY(PhysicsBody, velocity, vec3);
```

Type Registry

Simply creating instances of property classes is not enough - we need a way to find them and look them up

Need to add each property to a registry mapping it to its type so we can find it later

Registry needs to allow adding new types and adding new attributes

Type Registry Mechanics

C has no standard way to invoke code before `main()`, automatically or otherwise

Need to create a registration function for each type and invoke it in `main()`

Also need a global metatype pointer so we can look up types at compile time and not just runtime

Registry Macro Example

```
extern void meta_register_type(metatype* type);
#define META(TYPE) (&g_meta__##TYPE)
#define META_DECLARE(TYPE) \
  extern metatype g_meta__##TYPE; \
  extern void meta_init__##TYPE(void);
#define META_BEGIN_DEFINE(TYPE, BASE) \
  metatype g_meta__##TYPE; \
  void meta_init__##TYPE(void) { \
    typedef TYPE my_type; \
    metatype* my_meta = &g_meta__##TYPE; \
    g_meta__##TYPE.name = #TYPE; \
    g_meta__##TYPE.base = &META(BASE); \
#define META_END_DEFINE() \
  }
```

Registry Macro Example

```
void meta_add_property(metatype* type, metaproperty*
prop);
#define META_PROPERTY(type, name) \
  { \
    metaproperty prop; \
    prop.name = #name; \
    prop.offset = offsetof(my_type, name); \
    prop.type = META(type); \
    meta_add_property(my_type, &prop); \
  }
#define META_INIT(TYPE) \
  extern void init_meta__##TYPE(void); \
  init_meta__##TYPE();
```

Registry Macro Usage Example

```
// foo.h
#include "meta.h"
struct foo { float bar; };
META_DECLARE(foo);
```

```
// foo.c
#include "foo.h"
META_BEGIN_DEFINE(foo, void)
  META_PROPERTY(bar, float)
META_END_DEFINE()
```

```
// main.c
int main() { INIT_META(foo); }
```

Explanation of Macros

Use the `__` to avoid name clashes

`foo_bar_baz` could be `foo_bar` + `baz` or it could be `foo` + `bar_baz`

`foo_bar__baz` is a completely separate identifier from `foo__bar_baz` to the compiler

Explanation of Macros

Create `g_meta__##TYPE` so that you can look up any type's meta data using a well-known name, constructed via the `META` macro

Note that this will *not* work with C++ namespaces: `g_meta__foo::bar` cannot be declared

Explanation of Macros

The `META_PROPERTY()` macro must be invoked between the begin/end macros

`BEGIN` uses typedef and a local variable to ensure that later macros can find the type without having to repeat it in each macro

The `meta_add_*()` functions can be implemented with dumb linked lists to get started

Meta Functions (Methods)

C doesn't support OOP-style methods naturally, but C libraries and applications code often has an OOP-style in its API design

```
typedef struct _foo foo;
foo* foo_create(void);
void foo_destroy(foo*);
void foo_do_bar(foo*, int, float);
```

Complexity of the Problem

Just knowing that a function exists and address is not enough

We also needs to know its return type, arity, and parameter types

Then we can only invoke it in code that is expecting a particular signature or we need a convoluted thunking mechanism like a scripting language

Simplifying the Problem

Rarely need to dynamically find any method

Usually we need a specific genre of function, like an event handler or factory

We can restrict binding to only handling specific signatures that we need to lookup dynamically

C++ makes this easier, ignore hard stuff in C

Event Handlers Example

Assume objects can just have event handles with no data invoked

Would be easy to extend this to take a `message*` instead of just `void`

Event Handlers Macroas

```
void meta_add_event(metatype* type, metaevent* prop);
#define META_EVENT(handler, name) \
  { \
    metaevent ev; \
    ev.name = #name; \
    ev.handler = &handler; \
    meta_add_event(my_type, &ev); \
  }
```

Use is the same as `META_PROPERTY()`

Handler has a known signature `void (*handler)(void)` so using it is trivial

Using Introspection

Making use of all this support

Serialization example

```
void serialize(const metatype* meta, const void* obj) {
  for (metaproperty* prop = meta_get_first_property(meta);
       prop != NULL; prop = meta_property_next(prop)) {
    if (prop->type == META(int))
      write_xml_int(prop->name,
        *(const int*)((const char*)obj + prop->offset));
    else if (prop->type == META(float))
      write_xml_float(prop->name,
        *(const float*)((const char*)obj + prop->offset));
    else if (prop->type == META(string))
      write_xml_string(prop->name,
        *(const char* const*)((const char*)obj +
        prop->offset));
  }
}
```

Debug Consoles

```
void set_field(const metatype* type, void* obj,
  const char* name, const char* value)
{
  const metaproperty* prop = meta_get_property(type,
name);
  if (prop != NULL)
  {
    if (prop->type == META(int))
     *(int*)((const char*)obj + prop->offset) =
       atoi(value);
  }
}
```

Inheritance and Runtime Metatype Lookup

Handling OOP cases in C

Inheritance in C vs C++

In the single non-virtual inheritance case, a C++ class essentially just does this:

```
// C++
class Foo { int a; }
class Bar : public Foo { int b; }

// C
struct Foo { int a; }
struct Bar { struct Foo base; int b; }
```

C++ handles scoping different, but the layout is the same on all "real" implementations

Using Inheritance in C

Base member field can be accessed by specifying the base field, and the type's pointer can be "cast" into the base type using the base field's address

```
Bar bar;
bar.base.a = 3;
foo_do_something(&bar->base);
```

Using Inheritance in C

If the base type is always the first field of an object, and you're on a "real" C implementation, you can just cast to the base type

```
struct Bar { struct Foo base; int b; };
struct Bar bar;
foo_do_something((struct Foo*)&bar);
```

Runtime Metatype Lookup

There are two options, neither ideal, need C++ to make it really strong

1) Always "inherit" and generic object base type that supports introspection

2) Expect there to be a meta field of any type being looked up dynamically

Base Object Type

```
typedef struct _Object { const metatype* my_type; }
Object;
typedef Foo { Object base; };

void init_object(void* object, const metatype* type) {
  ((Object*)object)->my_type = type;
}

const metatype* get_meta(const void* object) {
  return ((const Object*)object)->my_type;
}

Foo foo;
init_object(&foo, META(foo));
```

Field Lookup

Notice with the previous example we just assumed that we could cast any object to `Object`, which may not be true, and may crash at runtime

We could just access the field and fail at runtime, so long as every struct as its own `my_type` field we set (which is redundant and wasteful for long inheritance chains)

Field Lookup

```
#define GET_META(object) ((object)->my_type)

struct Foo { const metatype* my_type; int a; }
struct Bar { struct Foo base;
  const metatype* my_type; int b; }

Bar bar;
bar.base.my_type = bar.my_type = META(Bar);
```

Notice the need to set `my_type` twice for the same object

Further Notes

Additional Work

Sample Code

Very simple but working sample code can be found online

http://github.com/seanmiddleditch/dpengineclub

C++

Moving to C++ makes a lot of things easier and safer

That's a future talk

General Function Binding

Could add a `meta_add_method` that allows you to register a function with any arity or types

Invoking said methods will require comparing a signature string to ensure that a method takes what you think it takes, or the creation of a general `Value` type that can hold anything and be passed as an array (like most scripting languages)

The End

Questions?