Implementing Reflection Using the New C++20 Tooling Opportunity: Modules

MAIKO STEEMAN

About me

- Tools & Engine programmer
- AAA Games Industry
- (prev) Creative Assembly
- Guerrilla

Summary

- What is reflection?
- Why?
- Implementing runtime reflection

What is reflection

- Metadata of code
- “What members do I have?”

```
struct Entity
{
    int health;
    std::string tag;

    void eat_burger();
};
```

Why should I care about reflection?

- Serialization
- Binary, JSON, etc.

```
json serialize_struct(any any_value)
{
    json json;
    for (Field f : reflect_fields(any_value)) {
        json[f.name]["value"] = f.value();
        json[f.name]["type"] = f.type;
    }
    return json;
}
```

Why should I care about reflection?

- Serialization
- Binary, JSON, etc.

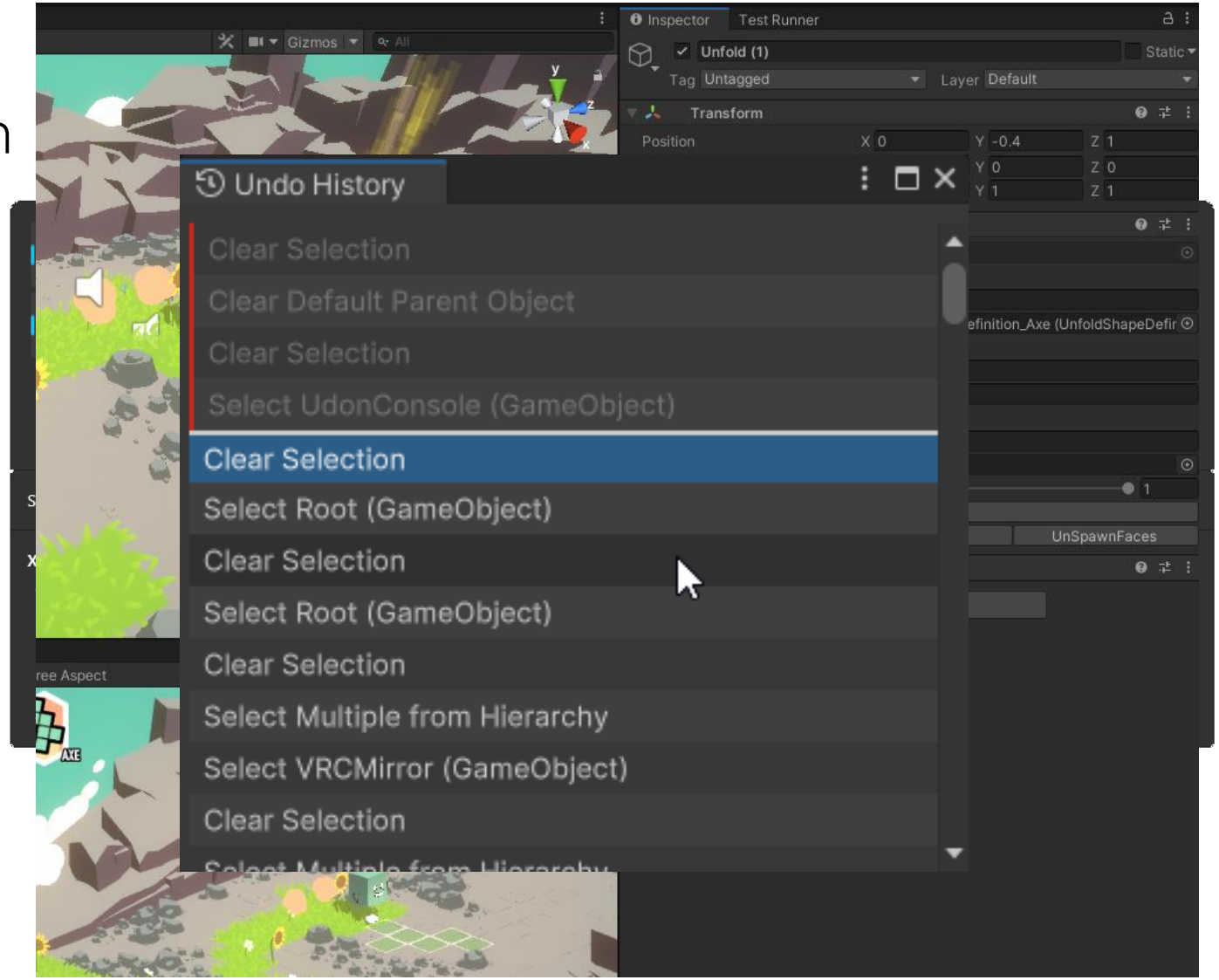
```
json value_to_json(any any_value)
{
    // Handle builtins
    if (any_value.type() == the_type<int>())
        return json{ any_value.value<int>() };
    if (any_value.type() == the_type<double>())
        return json{ any_value.value<double>() };
    if (any_value.type() == the_type<std::string>())
        return json{ any_value.value<std::string>() };

    // Recurse for classes
    if (any_value.is_class()) {
        return serialize_struct(any_value);
    }
}

json serialize_struct(any any_value)
{
    json json;
    for (Field f : reflect_fields(any_value)) {
        json[f.name]["value"] = value_to_json(f.value());
        json[f.name]["type"] = f.type;
    }
    return json;
}
```

Why do I care about reflection even more?

- Extension to the type system
- WPF, Automatic Bindings
- Language bindings: Python
- Content editors
- Automatic change detection



How do we get there?

- Implement reflection
- Go over current techniques
- Modules
- Patterns and Tricks for runtime reflection

Implementing an RTTI runtime

- Go from client API

```
json serialize_struct(any any_value)
{
    json json;
    for (Field f : reflect_fields(any_value)) {
        json[f.name]["value"] = f.value();
        json[f.name]["type"] = f.type;
    }
    return json;
}
```

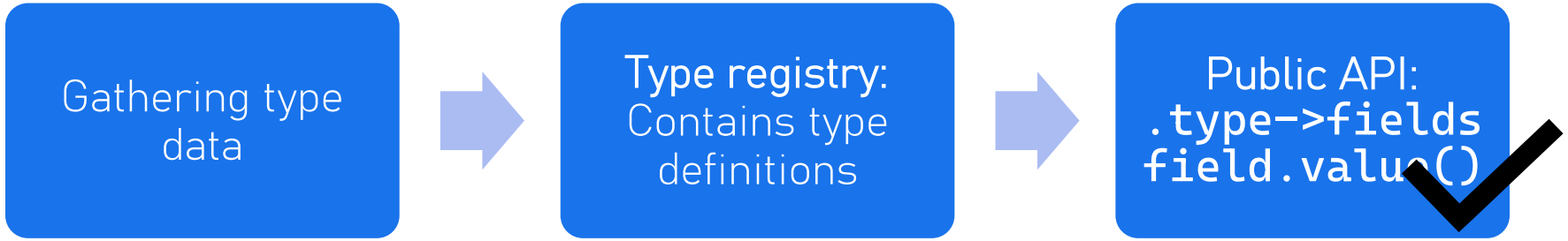
Implementing an RTTI runtime

- Go from **realistic** client API

```
struct AnyRef
{
    void* value;
    const Type* type;
};

json serialize_struct(AnyRef any_value)
{
    json json;
    for (const Field& f : any_value.type->fields) {
        json[f.name]["value"] = f.value(any_value);
        json[f.name]["type"] = f.type;
    }
    return json;
}
```

Implementing an RTTI runtime




```
json serialize_struct(AnyRef any_value)
{
    json json;
    for (const Field& f : any_value.type->fields) {
        json[f.name]["value"] = f.value(any_value);
        json[f.name]["type"] = f.type;
    }
    return json;
}
```

Implementing type registry

- Store all types

```
extern std::unordered_map<std::string, Type> type_registry;

template<typename T> void register_type(Field[], Method[]);
```

- Defining Type

```
struct Field;
struct Method;

struct Type
{
    std::string name;
    std::vector<Field*> fields;
    std::vector<Method*> methods;
};
```

Defining Field interface

- Type erasing a member variable
- 1st instinct, base class with virtuals

```
class Field
{
public:
    virtual ~Field() = default;

    virtual std::string_view name() = 0;
    virtual const Type* type() = 0;
    virtual AnyRef value(void* object) = 0;
};
```

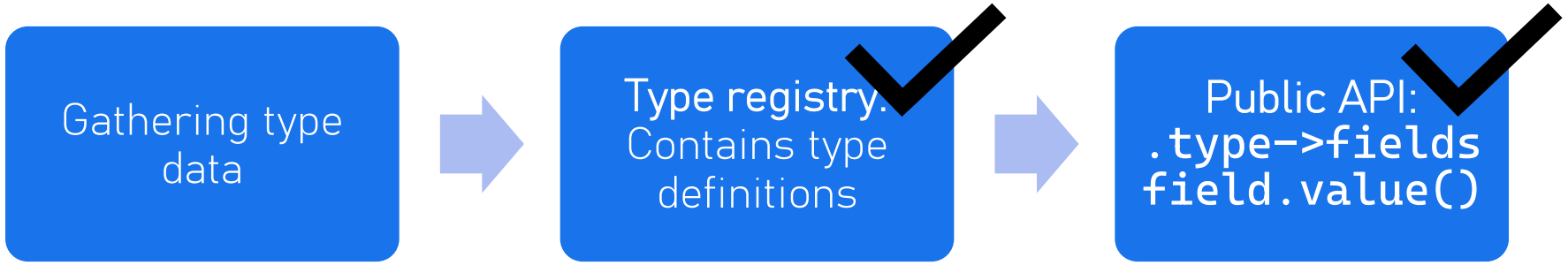
Defining Method interface

```
class Method
{
public:
    virtual ~Method() = default;

    virtual std::string_view name() = 0;
    virtual const Type* return_type() = 0;
    virtual std::span<const Type> parameter_types() = 0;
    virtual AnyRef invoke(void* object, std::span<void*> args) = 0;
};
```

Gathering type data

- Meat and potatoes of our reflection library




```
template<typename T>
void register_type(Field[], Method[]);
```

Current techniques: Manual

- Manual (E.g. RTTR)
  - Constantly add definitions

```
struct MyStruct {
    MyStruct() {};
    void func(double) {};
    int data;
};

RTTR_REGISTRATION
{
    registration::class_<MyStruct>("MyStruct")
        .constructor<>()
        .property("data", &MyStruct::data)
        .method("func", &MyStruct::func);
}
```

Current techniques: Automatic

- std::tuple_element (boost.pfr, magic_get)
  - Not flexible enough, no member functions, only tuple-like types.
  - *Reflection without Reflection TS - Fabian Renn Giles*
- Code parser
- Use existing compiler frontend

Current techniques: Automatic

- std::tuple_element (boost.pfr, magic_get)
  - Not flexible enough, no member functions, only tuple-like types.
- Code parser (E.g. Qt: Meta Object Compiler, Unreal Engine: Unreal Header Tool)
  - C++ grammar is very complex. Maintenance is a mess.
  - Requires deep build system integration
- Use existing compiler frontend

Current techniques: Automatic

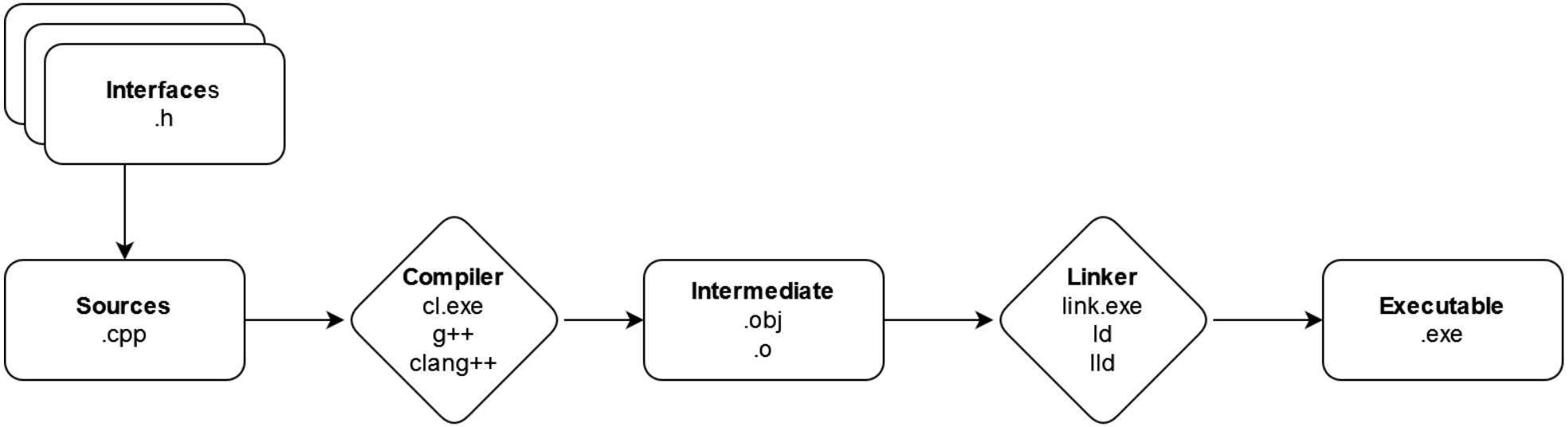
- std::tuple_element (boost.pfr, magic_get)
  - Not flexible enough, no member functions, only tuple-like types.
- Code parser (E.g. Qt: Meta Object Compiler, Unreal Engine: Unreal Header Tool)
  - C++ grammar is very complex. Maintenance is a mess.
  - Requires deep build system integration (passing includes, macro's).
- Use existing compiler frontend (LibClang, ClangTooling)
  - Slow.
  - Requires deep build system integration (passing includes, macro's).

Current techniques: Automatic

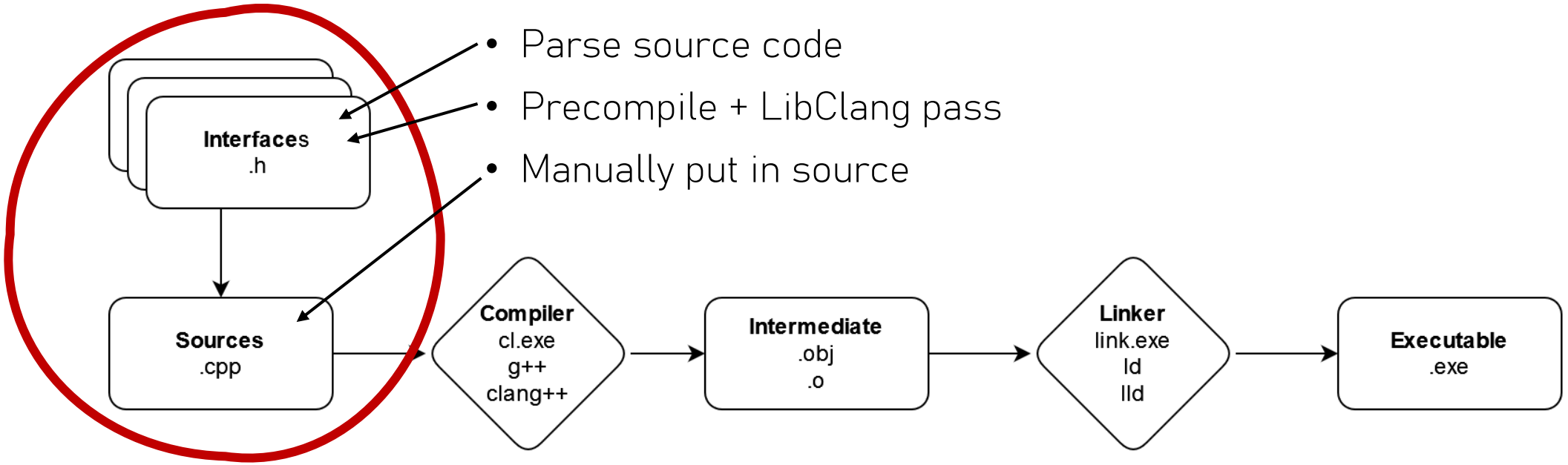
- std::tuple_element (boost.pfr, magic_get)
  - Not flexible enough, no member functions, only tuple-like types.
- Code parser (E.g. Qt: Meta Object Compiler, Unreal Engine: Unreal Header Tool)
  - C++ grammar is very complex. Maintenance is a mess.
  - Requires deep build system integration (passing includes, macro's).
- Use existing compiler frontend (LibClang, ClangTooling)
  - Slow.
  - Requires deep build system integration (passing includes, macro's)

C++ compilation process

- Headers mostly contain type info



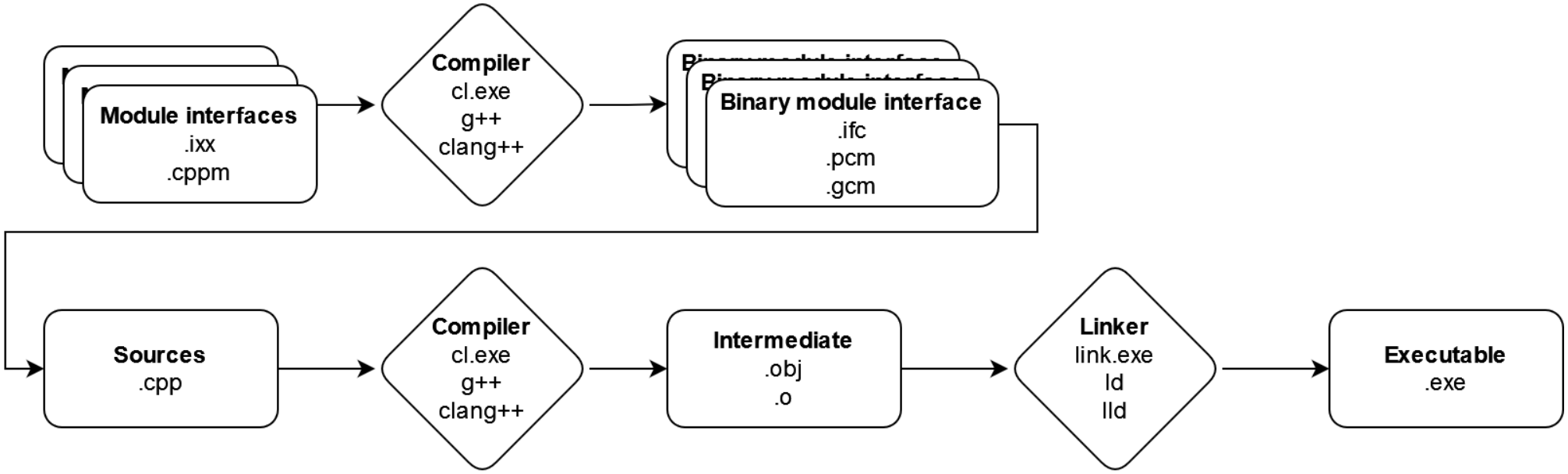
C++ compilation process



- Parse source code
- Precompile + LibClang pass
- Manually put in source

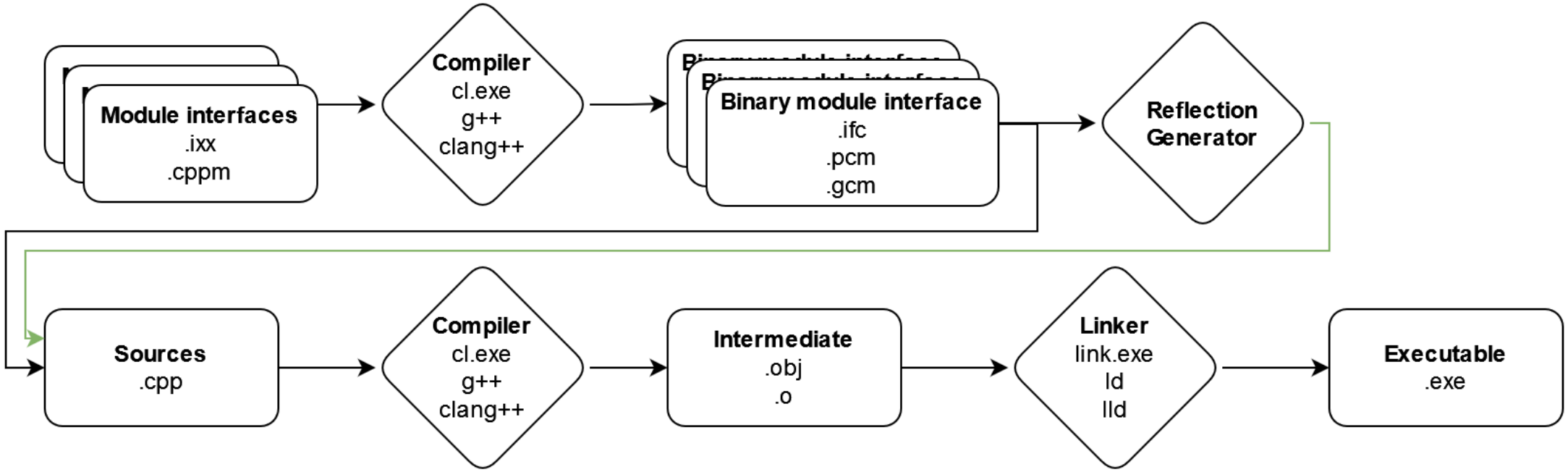
What happens with modules?

- “Modules are a tooling opportunity” – Gabriel dos Reis
- Fast! Work already happened



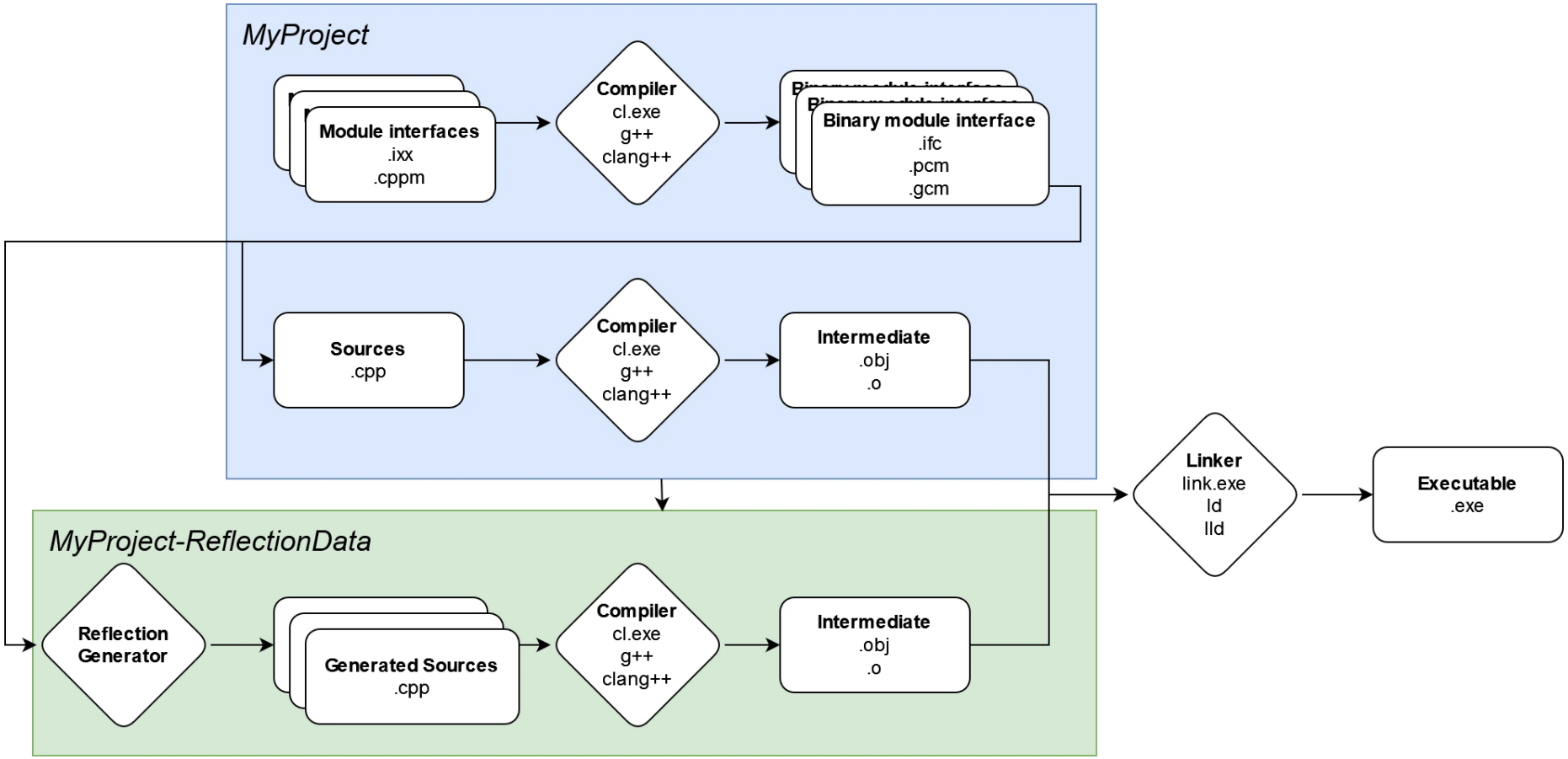
Using it for reflection

- No further context needed
- In theory, BMI always processed before .cpp



Using it for reflection

- In practice, no way to invoke mid compilation pass.
- Separate target = separation boundary



What is .ifc?

- Header
- “Map<str, Partition*>”
- E.g. partition

  “type.qualified” = QualifiedType[N];

unqualified: TypeIndex
qualifiers: Qualifiers

Figure 9.16: Structure of a qualified type

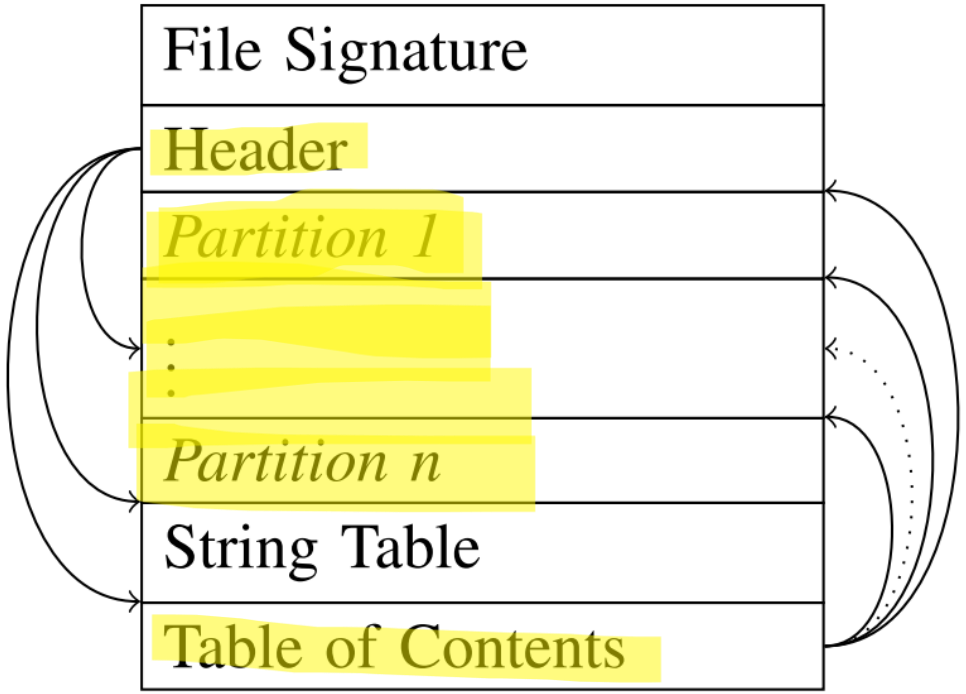


Figure 2.1: IFC file general structure

- https://github.com/microsoft/ifc-spec

What is .ifc?

- AbstractIndex = 2x numbers
- 1 indexes **which** partition
- The other **into that** partition

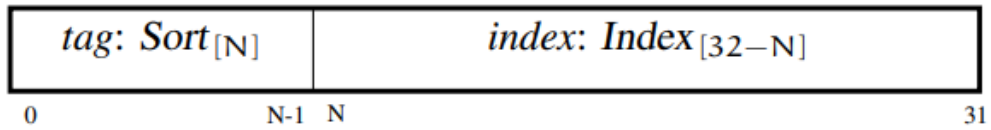


Figure 2.2: Abstract reference parameterized by the sort of designated entity.

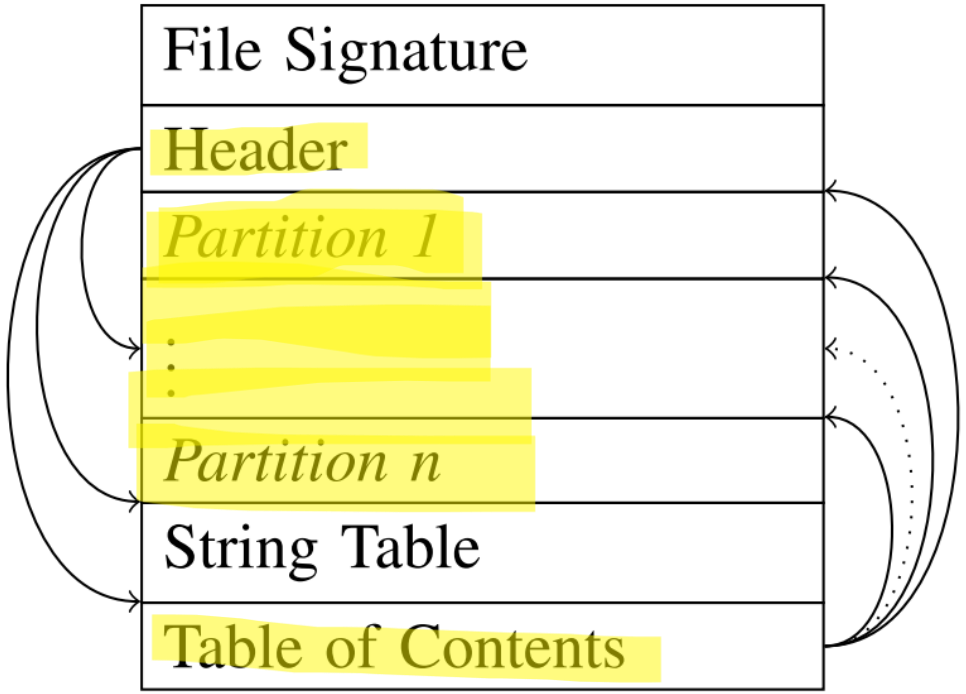
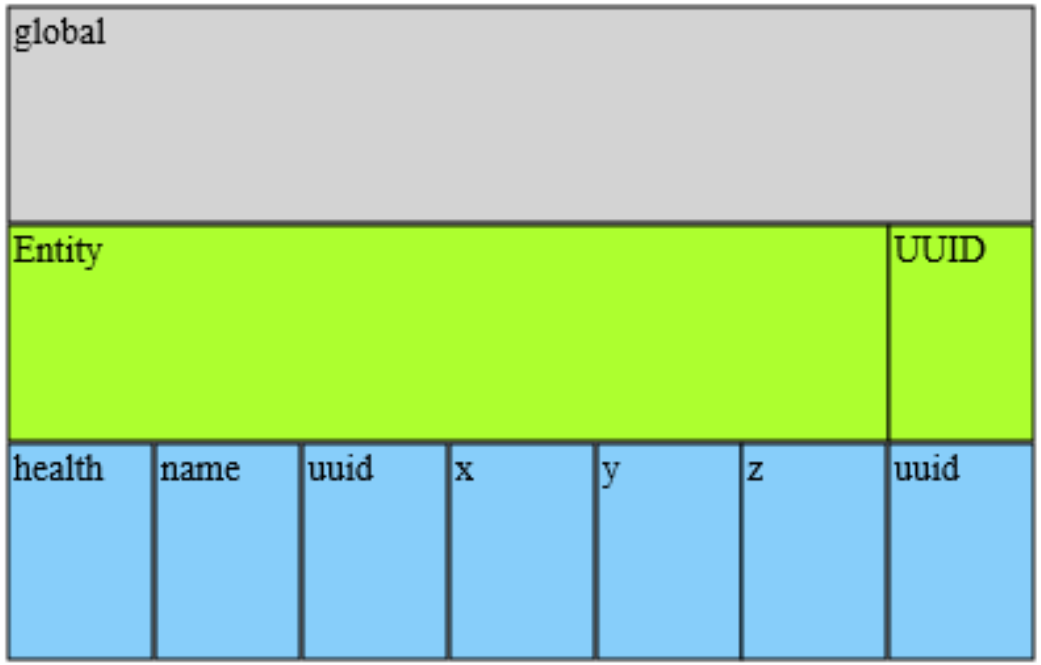


Figure 2.1: IFC file general structure

Inspecting .ifc

- Ifc sdk visualizer
  - https://github.com/microsoft/ifc



EntityModule.ixx

```
export module EntityModule;

export struct UUID
{
    uint64_t uuid;
};

export struct Entity
{
    int health;
    std::string name;
    UUID uuid;
    float x, y, z;
};
```

Cheekily plugging my own work

- Neatly:
  - implements simple, powerful reflection runtime library
  - generates data neatly listening to MSVC
  - integrates into your CMake project

```
export struct MyStruct {
    int damage;
};
```

```
add_library(MyProject "MyCode.ixx" "MyCode.cpp")
add_reflection_target(MyProject_ReflectionData MyProject)
add_executable(TheExe)
target_link_libraries(TheExe PUBLIC
    MyProject
    MyProject_ReflectionData
)
```

```
MyStruct value{ .damage = -5 };
Neat::get_type<MyStruct>();
id };
ields[0];
field.set_value(value_ptr, 75);
assert(value.damage == 75);
```

Source code:

https://github.com/FireFlyForLife/NeatReflection

Trivial Ifc limitations

- Only modules
- No user attributes
- BMI filename query

Non-trivial limitations

- Templates not instantiated
- Compiler specific
  - https://github.com/GabrielDosReis/ipr

Just Compile Time Reflection Things

- But then in runtime reflection land.
- During registration you have the type.
- Design of library meant to be rebuild using library pieces.

Implementing type registry

- Store all types

```
extern std::unordered_map<std::string, Type> type_registry;

template<typename T> void register_type(Field[], Method[]);
```

- Implementing Type

```
struct Field;
struct Method;

struct Type
{
    std::string name;
    std::vector<Field*> fields;
    std::vector<Method*> methods;
};
```

Field abstraction

- Reverse of member access

```
Entity* some_entity = GetSomeEntity();
std::cout << (*some_entity).name;
```

- Make the **member** a variable instead

```
auto some_member = GetSomeMember();
Entity entity;
std::cout << entity.*some_member;
```

- Relative pointer

```
struct Entity
{
    int health;
    std::string name;
    float x, y, z;
};
```

C++ pointer to member

- Syntax

```
using PtrToMemberFloat = float Entity::*;
PtrToMemberFloat member = &Entity::y;
```

- Dereference syntax

```
Entity entity;
entity.*member = 42.0f;

Entity* heap_entity = new Entity();
heap_entity->*member = 2024.0f;
```

- We now have our "relative pointer"

```
struct Entity
{
    int health;
    std::string name;
    float x, y, z;
};
```

Defining Field interface

- Type erasing a member variable
- 1st instinct, base class with virtuals

```
class Field
{
public:
    virtual ~Field() = default;

    virtual std::string_view name() = 0;
    virtual const Type* type() = 0;
    virtual AnyRef value(void* object) = 0;
};
```

Implement Field

```
class FieldImpl : public Field
{
public:
    using PtrToMember = int Entity::*;

    // Data
    PtrToMember ptr_to_member;

    // Functions
    AnyRef value(void* object) override
    {
        Entity* entity_object = static_cast<Entity*>(object);
        int* field_ptr = &(entity_object->*ptr_to_member);
        return AnyRef{ field_ptr, int_type };
    }
};
```

Implement Field

```
template<typename TObject, typename TField>
class FieldImpl : public Field
{
public:
    using PtrToMember = TField TObject::*;

    // Data
    PtrToMember ptr_to_member;

    // Functions
    AnyRef value(void* object) override
    {
        TObject* typed_object = static_cast<TObject*>(object);
        TField* field_ptr = &(typed_object->*ptr_to_member);
        return AnyRef{ field_ptr, field_type };
    }
};
```

Finished Field Implementation

```
template<typename TObject, typename TField>
class FieldImpl : public Field
{
public:
    using PtrToMember = TField TObject::*;

    // Data
    PtrToMember ptr_to_member;
    std::string field_name;
    Type* field_type;

    // Functions
    std::string_view name() override { return field_name; }
    const Type* type() override { return field_type; }
    AnyRef value(void* object) override
    {
        TObject* typed_object = static_cast<TObject*>(object);
        TField* field_ptr = &(typed_object->*ptr_to_member);
        return AnyRef{ field_ptr, field_type };
    }
};
```

Simplifying Field

```
template<typename TObject, typename TField, TField TObject::* PtrToMember>
class FieldImpl : public Field
{
public:
    // Data
    std::string field_name;
    Type* field_type;

    // Functions
    std::string_view name() override { return field_name; }
    const Type* type() override { return field_type; };
    AnyRef value(void* object) override
    {
        TObject* typed_object = static_cast<TObject*>(object);
        TField* field_ptr = &(typed_object->*PtrToMember);
        return AnyRef{ field_ptr, field_type };
    }
};
```

Simplified Field

```
struct Field
{
    // Data
    std::string name;
    Type* type;

    // Functions
    AnyRef value(void* object)
    {
        // ???
        // Where type erasure
    }
};
```

De-virtualized Field Implementation

```
struct Field
{
    // Data
    std::string name;
    Type* type;

    // Functions
    using ValueFunc = AnyRef (*)(void* object);
    ValueFunc value;
};

template<typename TObject, typename TField, TField TObject::* PtrToMember>
AnyRef value_func(void* object)
{
    TObject* typed_object = static_cast<TObject*>(object);
    TField* field_ptr = &(typed_object->*PtrToMember);
    return AnyRef{ field_ptr, field_type };
}
```

De-virtualized Method Implementation

```
struct Method
{
    // Data
    std::string method_name;
    Type* method_return_type;
    std::vector<Type*> method_parameter_types;

    // Functions
    using InvokeFunc = AnyRef(*)(void*, std::span<void*>);
    InvokeFunc invoke;
};

template<auto PtrToMemberFunction, typename TObject, typename TReturn, typename... TParams>
AnyRef invoke_func(void* object, std::span<void*> arguments)
{
    static_assert(std::is_same_v<decltype(PtrToMemberFunction), TReturn(TObject::*)(TParams...)>);

    TObject* typed_object = static_cast<TObject*>(object);

    auto invoke_internal = [&]<size_t... Indices>(std::index_sequence<Indices...>)
    {
        return (typed_object->*PtrToMemberFunction)(*static_cast<TParams*>(arguments[Indices])...);
    };

    return invoke_internal(std::index_sequence_for<TParams...>{});
}
```

Rapid fire additions

- Constructor
- Destructor
- Container access

```
struct Type
{
    // Data
    std::string name;
    std::vector<Field*> fields;
    std::vector<Method*> methods;

    // Functions
    using ConstructorFunc = void (*)(void*);
    ConstructorFunc constructor;
    using DestructorFunc = void (*)(void*);
    DestructorFunc destructor;
};

template<typename T>
void erased_constructor(void* object_memory)
{
    new (object_memory) T();
}

template<typename T>
void erased_destructor(void* object)
{
    static_cast<T*>(object)->~T();
}
```

Typeid

- Be able to get type by template parameter

```
template<typename T>
const Type* get_type();
```

- C++ language rtti **`typeid()`**
  - Generally, in gamedev & embedded turned off.
- Trick time!

```
int id_int = get_id<int>();
int id_double = get_id<double>();
int id_int2 = get_id<int>();
assert(id_int == id_int2);
```

Template type id trick

- Associate **template type** with **id number**
- Falls apart with dll's
- Call order dependent
- Not constexpr

```
// Simple number counter
extern std::atomic_int g_id_counter;

inline int generate_id()
{
    return g_id_counter++;
}

// Templated type id implementation
template<typename T>
int get_id()
{
    static const int id = generate_id();
    return id;
}
```

Template type id trick

- constexpr variation
- Still falls apart with dll's
- Call order independent.

```
// Reserve static memory
template<typename>
bool dummy_variable = false;

// Templated type id implementation
using TemplateTypeId = void*;

template<typename T>
constexpr TemplateTypeId get_id()
{
    return &dummy_variable<T>;
}
```

Base class slicing

- Represented with array of type refs

```
struct Type
{
    // Data
    std::string name;
    std::vector<int> base_class_type_ids;
```

Slicing

- `this` pointer needs to be at start of object
- Can let the compiler deal with it (duplication)
- `static_cast` to base class will do offsetting.

```
struct BaseA { int a; };
struct BaseB { int b; void func(); };

struct C : BaseA, BaseB {};

auto c_func = &C::func;
```

```
template<typename T, typename TBase>
void* rebase_ptr(void* object)
{
    T* typed_object = static_cast<T*>(object);
    return static_cast<TBase*>(typed_object);
}
```

Thank you for listening!

- Any questions?

NeatReflection code:

https://github.com/FireFlyForLife/NeatReflection

Resources

- Ifc sdk: https://github.com/microsoft/ifc
- Ifc spec: https://github.com/microsoft/ifc-spec
- Ifc-reader: https://github.com/AndreyG/ifc-reader
- Ifc-sdk showcase: https://www.youtube.com/watch?v=t6QCzVXrwIw
- NeatReflection: https://github.com/FireFlyForLife/NeatReflection
- Gaby Dos Reis - Programming in the Large With C++ 20 - Meeting C++ 2020 Keynote: https://www.youtube.com/watch?v=j4du4LNsLiI

Future stuff to be excited about!

- Given BMI is a processed MI file, a code generator could directly go there, and forego the compiler's source code parser entirely!

Alternative Field de-virtualization

- every impl ptr to member being equally sized
- But MSVC...
- PtrToMember = 4 bytes for POD.
- 12 for forward decl

Alternative Method invoke impl

```
template<auto PtrToMemberFunc>
struct InvokeHelper;

template<typename TObject, typename TReturn, typename... TArgs, TReturn (TObject::*PtrToMemberFunc)(TArgs...)>
struct InvokeHelper<PtrToMemberFunc>
{
    static AnyRef invoke_func(void* object, std::span<void*> arguments)
    {
        TObject* typed_object = static_cast<TObject*>(object);
        auto invoke_internal = [&]<size_t... Indices>(std::index_sequence<Indices...>)
        {
            return (typed_object->*PtrToMemberFunction)(*static_cast<TParams*>(arguments[Indices])...);
        };
        return invoke_internal(std::index_sequence_for<TParams...>{});
    }
};
```