Writing a reflection system for C++14

Manu Sanchez @Manu343726

Reflection

From wikipedia

> In computer science, reflection is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime

Reflection

- Examine the program structure
- Manipulate the program structure programatically
- At compile time: "***static reflection***"
- At runtime: "***dynamic reflection***"

Given...

```cpp
namespace mylib
{
    class MyClass
    {
    public:
        MyClass() = default;

        void myMethod(const std::string& str);

        int i = 0;
    };
}
```

Static reflection

```
using Method = cpp::static_reflection::Method<
    decltype(&MyClass::myMethod),
    &MyClass::myMethod
>;

// File where the method is declared
constexpr auto filePath = Method::SourceInfo::file();
```

- Pros:
  - Type safe
  - Zero overhead
- Cons:
  - Ugly for the user. *TMP based* (`constexpr` could help)

Reminder...

```
namespace mylib
{
    class MyClass
    {
    public:
        MyClass() = default;

        void myMethod(const std::string& str);

        int i = 0;
    };
}
```

Dynamic reflection

```cpp
cpp::dynamic_reflection::Runtime runtime;
...
const auto& myObject = runtime
    .namespace_()         // Global namespace
    .namespace_("mylib") // ::myLib namespace
    .class_("MyClass")   // ::myLib::MyClass class
    .create(); // Returns object

// Invoke method on the object:
myObject("myMethod")(std::string{"hello"});

// Set member variable:
myObject["i"] = 42;
```

- Pros:
    - User-friendlier API
    - Entities manipulable at runtime
- Cons:
    - Performance

Our goal

Giving both static and dynamic reflection features to C++

No, really

- Let's restrict the API to OOP:
  - Classes
  - Enums
  - Public non static member functions (*methods*)
  - Public non static member variables (*fields*)

What we need

- Entity metadata:
    - Classes declared in a header
    - Class members
    - Source file path
    - Source line
    - Pointers to members
- An API to read metadata: `api(entity) -> metadata`
    - A C++ representation of the metadata
    - Entity to metadata mapping

> **Entity**: A C++ sourcecode semantic element (A class, member function, enum, namespace, etc)

Collecting metadata

Collecting metadata

- Per-header metadata

> *For each header file generate a file with the metadata collected from the C++ header*

- No global codebase information

> *No cross-header namespace info, but makes things easier*

- Metadata is stored in **generated C++ header files**

> *Easy to import, just* `#include`

  - *So metadata is stored as C++ code?* Exactly. More on this later

Collecting metadata: libclang

libclang

- A C interface to the clang library
- Provides support for C++ AST parsing and transversal

libclang: How it works?

- The Clang compiler is written as a series of C++ libraries

  > `-pedantic` **note**: Clang is not a compiler but a C++ frontend for LLVM

- libclang provides a C interface for the clang API
- The libclang C API **is stable between clang releases**

  > *The C++ API may apply breaking changes between versions*

- C ABI: **There are libclang bindings for multiple languages**

libclang: Hello world with Python

```python
from clang.cindex import Index, Cursor, CursorKind

args = ['-std=c++11']  # Compilation options
index = Index.create() # Create a compilation DB

# Parse a translation unit:
tu = index.parse('project.hpp', args)

def visit(cursor):
    print 'file {}:{} {} ({})'.format(
        cursor.location.file,
        cursor.location.line,
        cursor.spelling,
        cursor.kind)

    for c in cursor.children():
        visit(c)

visit(tu) # Visit the translation unit AST
```

DRLParser

"*Diego Rodriguez Losada Parser*"

("*Dynamic Reflection Library Parser*")

libclang python basics

Python wraps the AST visitation API as a `Cursor` class with properties about the current AST node being visited:

- `kind`: Type of node (A function declaration, a class declaration, a namespace declaration, etc). `CursorKind` enum.
- `spelling`: Name of the entity being declared/defined
- `displayname`: Detailed entity name. `
- `location`: Entity location in the sourcecode: File, line, etc

libclang python basics

Many more properties. See: https://github.com/llvm-mirror/clang/blob/master/bindings/python/clang/cindex.py

(Back to) Reading metadata

Reading metadata

- Take only the interesting cursors (those of an specific kind): `CursorKind.CLASS_DECL`, `CursorKind.STRUCT_DECL`, `CursorKind.ENUM_DECL`, `CursorKind.CXX_METHOD`, `CursorKind.FIELD_DECL`, etc
- Expose new properties you might need:

```
@property
def fullname(self):
    if self.parent is None:
        return '::' + self.cursor.spelling
    else:
        return self.parent.fullname + '::' +
               self.cursor.spelling
```

Generating C++ code

- Write templates with the C++ model of the metadata
- Pass the processed AST to your favorite template engine
- Done!

Generating C++ code

- The format of your generated metadata depends on the reflection API you are going to write
  - See Qt's MOC compiler, `QMetaType`, and `xxxx_moc.cpp` generated code.
- We are writing an static reflection API, so metadata must be manipulable at compile time
- Which means...

Template metaprogramming!!!

- No, really?
- Well, that's what I did
- Try `constexpr` + `boost::hana`

Generating C++ code: Metadata format

- `cpp::static_reflection::meta`: Templates storing metadata

```
namespace cpp::static_reflection::meta {
    template<
        typename Spelling,
        typename DisplayName,
        typename FullName,
        typename FilePath,
        std::size_t Line
        ...
    > class SourceInfo { ... };
}
```

Generating C++ code: Metadata format

- Public static constexpr member functions to return the properties:

```
template<...>
class SourceInfo
{
public:
    // Returns a const ref to the constexpr string
    // with the full name:
    static constexpr decltype(auto) fullName()
    {
        return cpp::constexp::SequenceToString<
            FullName
        >::str();
    }
    ...
};
```

Generating C++ code: Metadata format

- Public typedefs to typelists to represent sets of metadata:

```
template<
    // A SourceInfo<...> instance:
    typename SourceInfo_,
    // A typelist of Function<fptr type, fptr>
    // instances:
    typename Methods_,
    ...
> class Class
{
public:
    using SourceInfo = SourceInfo_;
    using Methods = Methods_;
    ...
};
```

- `cpp::static_reflection::meta` templates have another metadata instances (or lists of) as parameters

Generating C++ code: Writing metadata

- Write a template that takes a processed AST and writes `cpp::static_reflection::meta` instances into a header file.

  > *Remember when we said "Per-header info only?*

Generating C++ code: Writing metadata

```
{% macro sourceinfo(node, kind, class = None) -%}
::cpp::static_reflection::meta::SourceInfo<
    {{sourceinfo_entity(node, class)}},
    ::cpp::static_reflection::Kind::{{kind}},
    ::cpp::meta::string<{{node.fullname_as_charpack}}>,
    ::cpp::meta::string<{{node.spelling_as_charpack}}>,
    ::cpp::meta::string<{{node.displayname_as_charpack}}>
    ::cpp::meta::string<{{node.file_as_charpack}}>,
    {{node.cursor.location.line}}
>
{%- endmacro %}
```

See `siplasplas@github/src/reflection/parser/templates/`

The static reflection API

The static reflection API: Goals (again)

- **Mapping from entity to metadata**
- Metadata can be queried at compile-time

The static reflection API: API entry points

- How we access `cpp::static_reflection::meta` instances?
- **`cpp::static_reflection` namespace**: `Class`, `Enum`, `Field`... All counterparts of `meta` templates that take **an entity as parameter**:

```
#include <path/to/gen/code/myclass.hpp>

using MyClassData = cpp::static_reflection::Class<
    MyClass
>;
```

The static reflection API: API entry points

- When generating code, we specialize `cpp::static_reflection::codegen` templates with the entities as parameter.
- Those specializations inherit from the metadata (`cpp::static_reflection::meta` instances)
- `cpp::static_reflection::codegen` instead of `cpp::static_reflection` to provide an extra layer of indirection

  > So we can get rid of all the libclang stuff when standard reflection becomes real

The static reflection API: API entry points

```
namespace cpp::static_reflection::codegen {
    template<>
    class Field<
        decltype(&::project::Dog::dogAlias),
        &::project::Dog::dogAlias
    > :
        public ::cpp::static_reflection::meta::Field<
            ::cpp::static_reflection::meta::SourceInfo<
                ::cpp::static_reflection::codegen::Field<
                    decltype(&::project::Dog::dogAlias),
                    &::project::Dog::dogAlias
                >,
                ::cpp::static_reflection::Kind::FIELD,
                ::cpp::meta::string<':', ':', 'p', 'r', '
                ::cpp::meta::string<'d', 'o', 'g', 'A', '
                ::cpp::meta::string<'d', 'o', 'g', 'A', '
                ::cpp::meta::string<'/', 'h', 'o', 'm', '
                47
            >,...
        >
    {} } // C++17 only...
```

Playing with the static reflection API: JSON serialization

JSON Serialization

- Let's write two template functions:
  - `serialize(const T& value)`
  - `deserialize<T>(const nlohmann::json& json)`

  > This one must be implemented through a template class,
  > (we cannot apply SFINAE directly)

- Completely generic
- Type safe
- JSON for Modern C++

JSON Serialization

- Map C++ values to JSON objects `{"type": "typename", "value": ...}`
- The type of the original value is stored as an string. We can check it later

  > *A type name string can be "easily" computed at compile time. See https://github.com/Manu343726/ctti*

- Each kind of C++ value it's serialized in an optimal way

JSON Serialization: Fundamental types

```
template<typename T>
std::enable_if_t<std::is_fundamental<T>::value, nlohmann:
serialize(const T& value)
{
   auto json = nlohmann::json::object();

   json["type"]  = cpp::lexical_cast(ctti::type_id(value)
   json["value"] = value;

   return json;
}
```

JSON Serialization: Fundamental types

```
template<typename T>
class Deserialize<T, std::enable_if_t<std::is_fundamental
{
public:
    static T apply(const nlohmann::json& json)
    {
        SIPLASPLAS_ASSERT_EQ(
            json["type"].get<std::string>(),
            cpp::lexical_cast(ctti::type_id<T>().name())
        );

        return json["value"];
    }
};
```

JSON Serialization: unordered_map

```
template<typename Key, typename Value>
nlohmann::json serialize(const std::unordered_map<Key, Va
{
    auto json   = nlohmann::json::object();
    auto array  = nlohmann::json::array();

    for(const auto& keyValue : value)
    {
        const auto& key   = keyValue.first;
        const auto& value = keyValue.second;

        array.push_back(nlohmann::json::object({
            {"key", serialize(key)},
            {"value", serialize(value)}
        }));
    }

    json["type"]  = cpp::lexical_cast(ctti::type_id(value
    json["value"] = array;

    return json;
}
```

JSON Serialization: unordered_map

```
template<typename Key, typename Value>
class Deserialize<std::unordered_map<Key, Value>, void>
{
public:
    static std::unordered_map<Key, Value> apply(const nlo
    {
        SIPLASPLAS_ASSERT(json["value"].is_array());
        SIPLASPLAS_ASSERT_EQ(
            json["type"].get<std::string>(),
            cpp::lexical_cast(ctti::type_id<
                std::unordered_map<Key, Value>
            >().name())
        );
```

```
        std::unordered_map<Key, Value> map;

        for(const auto& keyValue : json["value"])
        {
            // Weird "I was debugging" checks here...
            SIPLASPLAS_ASSERT(keyValue.is_object())(
                "keyValue ({}) must be an object node. Is
                keyValue.dump(),
                keyValue.type()
            );

            map[Deserialize<Key>::apply(keyValue["key"])]
                Deserialize<Value>::apply(keyValue["value
        }

        return map;
    }
};
```

And finally...

JSON Serialization: Objects

```
template<typename T>
std::enable_if_t<std::is_class<T>::value, nlohmann::json>
serialize(const T& object)
{
    auto json = nlohmann::json::object();
    auto fields = nlohmann::json::object();
```

```
    // User defined classes are serialized by serializing
    // generating a "<member name>: <value>" JSON object
    // serialized values
    cpp::foreach_type<
        typename cpp::static_reflection::Class<T>::Fields
    >([&](auto type)
    {
        using FieldInfo = cpp::meta::type_t<decltype(type

        // We use the spelling of the field as key, and t
        fields[cpp::lexical_cast(FieldInfo::SourceInfo::s
            cpp::invoke(FieldInfo::get(), object) // C++1
        );
    });

    json["type"]  = cpp::lexical_cast(ctti::type_id(objec
    json["value"] = fields;

    return json;
}
```

JSON Serialization: Objects

```
template<typename T>
class Deserialize<
    T,
    std::enable_if_t<
        std::is_class<T>::value &&
        IsNotSpecializedType<T>::value // std::string,
                                       // std::tuple,
                                       // etc
    >
>
{
public:
```

```cpp
    static T apply(const nlohmann::json& json)
    {
        SIPLASPLAS_ASSERT_EQ(
            json["type"].get<std::string>(),
            cpp::lexical_cast(ctti::type_id<T>().name())
        );

        const auto& fields = json["value"];
        T object;
```

```
    cpp::foreach_type<
        typename cpp::static_reflection::Class<T>::Fields
    >([&](auto type)
    {
        using FieldInfo = cpp::meta::type_t<decltype(type
        using Type = typename FieldInfo::value_type;

        cpp::invoke(FieldInfo::get(), object) =
            Deserialize<Type>::apply(fields[cpp::lexical_
                FieldInfo::SourceInfo::spelling()
            )]);
    });

    return object;
}
};
```

JSON Serialization

See the full example at:

`siplasplas@github/examples/reflection/static/serialization.cpp`

Google Protobuf comm model to C++ data model

- Given a communication model written using Protobuf, and a C++ business data model
- **Can we map between these two models automatically?**

Google Protobuf comm model to C++ data model

- Why?
- WM&IP aka *"telefonillo"*. We have a 2k LOC file **just for this**.
- I'm sick of doing this stuff again and again...

Google Protobuf comm model to C++ data model

- We invoke two `serialize<Protobuf, T>(const T& value)` and `deserialize<T, Protobuf>(const Protobuf& proto)`
- Very similar to the JSON example
- Users can define conversions through a `cast<From, To>(const From& from)` customization point

Google Protobuf comm model to C++ data model: Algorithm

- To serialize (C++ to protobuf):
  1. Foreach field in the C++ object...
  2. Search for matching setters in the protobuf c++ generated code: `set_xxxx()`, `set_allocated_xxxx()`, etc
  3. Get the nearest setter. Invoke it and pass the value of the field. Apply conversion if needed
- To deserialize (Protobuf to C++):
  1. Default construct a C++ object
  2. Foreach field in the C++ object...
  3. Search for matching getters in the protobug c++ generated code: `xxxx()`, `get_xxxx()`, etc

Google Protobug comm model to C++ data model

Full example at `siplasplas@github/examples/reflection/static/protoserialization`

Dynamic reflection

Dynamic reflection

- Parsing and accessing metadata at compile time is the hard thing
- Dynamic reflection just exposes this information at runtime

Dynamic reflection

- `cpp::dynamic_reflection` exposes reflection information in the form of a classic inheritance-based entity system
    - `Class`, `Field`, `Enum`; inheriting from `Entity`
    - Entities are managed by a `Runtime`

Dynamic reflection: Entities

```
namespace cpp::dynamic_reflection {

class Entity : public std::enable_shared_from_this<Entity
{
public:
    virtual ~Entity() = default;

    Entity& parent();
    Runtime& runtime();

    const SourceInfo& sourceInfo() const;
    const std::string& name() const;
    const std::string& fullName() const;
    const SourceInfo::Kind& kind() const;

    bool detached() const;
    void attach(Runtime& runtime);
    bool orphan() const;
```

```
    void attach(const std::weak_ptr<Entity>& parent);
    void addChild(const std::shared_ptr<Entity>& entity);

    bool isChild(const std::shared_ptr<Entity>& entity) c
    bool isChild(const Entity& entity) const;
    bool isChildByName(const std::string& name) const;
    bool isChildByFullName(const std::string& fullName) c
    Entity& getChildByFullName(const std::string& fullNam
    Entity& getChildByName(const std::string& name);
    std::vector<std::string> getChildrenNamesByKind(const

    std::shared_ptr<Entity> pointer();

    friend bool operator==(const Entity& lhs, const Entit
    friend bool operator!=(const Entity& lhs, const Entit

protected:
    Entity(const SourceInfo& sourceInfo);

private:
    ...
};
```

Dynamic reflection: Entities

- All entities are registered **by full qualified name** in their runtime
- Entities maintain a hierarchichal relationship, so users can walk along
  the namespaces, classes, etc
- Entities can be queried by sipmple name (spelling) **through the parent entity**
- The global namespace is the root of the hierarchy

Dynamic reflection: Entities

- There's an `Entity` subclass for each kind of entity supported (`Class`, `Enum`, `Namespace`, etc)
- Entity classes provide factories to return entity objects from type-erased entity pointers:

```
// Throws if there's no "::mylib::MyClass" entity or
// a class entity
Class& class_ = Class::fromEntity(runtime.get("::myli
```

- Entity classes provide operations specific to that kind of entity:

```
auto object = class_.create(); // Default constructs
Enum& enum_ = class_.enum_("Enum"); // Return ::mylib

std::int64_t firstEnumVal = enum_.value(0);
const std::string& enumValStr = enum_.toString(firstE
std::int64_t enumValue = enum_.fromString("FIRST");
```

Dynamic reflection: Objects

- The `Class` entity returns instances of the class
- The instances are stored in a type-erased container, `cpp::Any32`

Dynamic reflection: Objects

- `cpp::Any32`:
  - Can store values of different types at runtime
  - Similar to C++17 `std::any` or `boost::any`
  - 32 byte fixed-size storage with fallback to dynamic allocation
  - Stores a map of **attributes and functions** for easy OO-like
    manipulation of the object

Dynamic reflection: Objects

```
cpp::Any32 object = class_.create();

// Invoke object.method with params 42 and "hello"
object("method")(42, std::string{"hello"});

// Assign a value to the public member variable "i"
object["i"] = 42 * 42;
```

Dynamic reflection: Runtime API loading

- Dynamic libraries can export their API by invoking `SIPLASPLAS_REFLECTION_DYNAMIC_EXPORT_TYPES(Ts...)` macro
- The macro uses static reflection to collect reflection information of the specified library types
- That information is accesible through a C API function that loads it into a dynamic reflection `Runtime` object

```
// mylib_export.cpp

#include <reflection/mylib.hpp>
#include <siplasplas/reflection/dynamic/exporttypes.hpp>

SIPLASPLAS_REFLECTION_DYNAMIC_EXPORT_TYPES(
    ::mylib::MyClass,
```

Dynamic reflection: Runtime API loading

- Users can load APIs at runtime with a `cpp::dynamic_reflection::RuntimeLoader` object, which takes a
  dynamic library handle and returns a `Runtime` object

Dynamic reflection: Runtime API loading

```cpp
#include <siplasplas/reflection/dynamic/runtimeloader.hpp

int main()
{
    auto lib = cpp::DynamicLibrary::load("libmylib.so");
    cpp::dynamic_reflection::RuntimeLoader loader{lib};
    auto& runtime = loader.runtime();

    auto object = runtime
        .namespace_()
        .namespace_("mylib")
        .class_("MyClass")
        .create();

    object("run")(); // Invoke object.run()
}
```

Future work

Future work: A dynamic type system

- Dynamic type conversions
- Declaring "metatypes" at runtime

Future work: Runtime C++ compilation

- Proof of concept at: `siplasplas@github/examples/cmake/project.cpp`
- See `siplasplas-cmake` module in the docs
  - Really looking forward to put my hands on cmake-server!

Future work: Maintainance

- I've had no time to handle some things properly
- From June to middle July I was full(siplasplas)-time writing docs :(
- This used to be fully cross-platform (VS2015 included) but I wroke a lot
  of things when writing siplasplas-typeerasure...
- The CI builds are broken (*It works on my machine*)
- Third party management
  - Github says I have 5% of cmake scripts in the project...
  - conan.io save me. Diego you owe me a master class!
- Devlog and more crazy stuff blogposts!

- https://github.com/Manu343726/siplasplas
- Website
- Latest doxygen docs

thanks_for_coming(); }

www.by.com.es