Macro free, non intrusive Runtime Reflection System in C++

skypjack

November 24, 2018

Table of contents

Reflection and C++
Overview
Static vs Runtime
Challenges
A toy example
Any as in any type
Track types
A type to rule them all
Data members
Almost there...
Questions?
Beyond the talk

A trending topic

Reflection (or rather, its lack) is a trending topic in the C++ community. It has been around for many years and the committee is still discussing it in the form of several proposals.

Support for **static reflection** may or may not find its way into the standard with C++20. In the meantime, we must continue to face its lack.

To runtime or not to runtime

Static reflection

A compile-time tool to inspect types. Static reflection is what may be introduced in the standard sooner or later.

It's all about actual types.

Runtime reflection

Usually defined as a mixed compile-time and runtime tool.
Most likely it will never find its way into the standard.

Actual types and reflected ones can diverge.

Detection idiom

Scratching the surface of static reflection:

```
template<typename T, typename = void>
struct has_f: std::false_type {};

template<typename T>
struct has_f<T, std::void_t<decltype(std::declval<T>().f())>>
  : std::true_type
{};
```

We need a lot more templates to get to something usable.

Pros and cons

A few hints on pros and cons

Static reflection:

- Types must be known
- It's all about actual types
- Compilation time
- Built-in idiosyncrasies
- Performance
- ...

Runtime reflection:

- Unknown types friendly
- Actual types: who cares?
- Compilation time (maybe)
- Differences can be avoided
- It's a runtime tool
- ...

Why runtime reflection

So, why should I choose runtime reflection instead of static one?

- First and foremost: **why not**?

Why runtime reflection

So, why should I choose runtime reflection instead of static one?

- First and foremost: **why not**?
- Far **easier to implement**, extend and maintain

Why runtime reflection

So, why should I choose runtime reflection instead of static one?

- First and foremost: **why not**?
- Far **easier to implement**, extend and maintain
- **No more differences** between classes, enums and so on

Why runtime reflection

So, why should I choose runtime reflection instead of static one?

- First and foremost: **why not**?
- Far **easier to implement**, extend and maintain
- **No more differences** between classes, enums and so on
- Much more flexible: we are **not bound to actual types**

Why runtime reflection

So, why should I choose runtime reflection instead of static one?

- First and foremost: **why not**?
- Far **easier to implement**, extend and maintain
- **No more differences** between classes, enums and so on
- Much more flexible: we are **not bound to actual types**

Keep note

Actual types and meta types **can diverge** to some extents.

The big picture

Minimum requirements of a reflection system:

- Classes, enums and fundamental types
- Support for public data members
- Meta any, meta data, meta type

Nice to have for a **full-featured reflection system**:

- Constructors, destructors and member functions
- Conversion functions and implicit cast
- Built-in support for setters and getters
- Single, multiple and virtual inheritance
- Custom meta objects (data, functions, and so on)
- Properties (everybody wants them)
- ...

What's around the corner?

The *this* pointer

> *In the body of a non-static member function, the keyword* `this` *is a prvalue whose value is a pointer to the object for which the function is called. The type of this in a member function of a class* `X` *is* `X*`.

Pitfalls

Derived classes, virtual functions, const, implicit cast, conversions, and so on. Developing a full-featured reflection system is definitely **a good exercise** to test skills with C++.

Fifty minutes aren't that much

Remember the **minumum requirements** of a reflection system and cut them a little more:

- Classes and fundamental types
- Support for non-static, non-const public data members
- Meta any, meta data, meta type

More than enough to introduce a minimal *web of meta nodes*.

Any as in any type

`std::any`: is it good?

Short answer: **no**. Long answer: **really, no.**

- To sum up with one word: `std::type_info`
- No control over SBO and therefore over allocations
- No `operator==`, no `can_cast`/`can_convert` or similar

`std::any` is certainly a great tool, but for other purposes.
However, all in all it's also all right for our minimal example.

Meta information

Real world tools require **shortcuts** for **direct access**.
In our case, this is more than enough:

```
template<typename...>
inline meta_type_node *types = nullptr;
```

Vademecum:

- `types<>` is the head of the implicit list of meta types
- `types<T>` is used for checks and direct access

The meta type node

The actual type is **erased**:

```
struct meta_type_node final {
  const char *name;
  meta_type_node * const next;
  meta_data_node *data;
};
```

The web of meta objects is all about **implicit intrusive lists**:

- They allow us to get rid of allocations
- They guarantee a linear complexity
- There is much room for optimizations

Factory

Attaching a meta type to its own list is trivial:

```
template<typename Type>
struct meta_factory {
  meta_factory<Type> reflect(const char *name) {
    static meta_type_node node{ name, types<>, nullptr };
    types<Type> = &node;
    types<> = &node;
    return *this;
  }

  // ...
}
```

Tips and tricks

Runtime reflection is (almost always) **all about** `static` **stuff**.

Iterate them all

What if I want to iterate all the available types?

```
template<typename Op>
void iterate(Op op) {
  auto *curr = types<>;

  while(curr) {
    op(meta_type{curr});
    curr = curr->next;
  }
}
```

Remember

Implicit intrusive lists started from a `nullptr` value.

Types, types everywhere

Users should not work directly with the underlying nodes:

```
struct meta_type {
  // ...

  template<typename Op>
  void data(Op op) const {
    auto *curr = node->data;

    while(curr) {
      op(meta_data{curr});
      curr = curr->next;
    }
  }

private:
  const meta_type_node *node;
};
```

The meta data node

The actual type is **erased**:

```
struct meta_data_node final {
  const char *name;
  meta_data_node * const next;
  meta_type_node *(* const type)();
  void(* const set)(void *, const std::any &);
  std::any(* const get)(void *);
};
```

Pretty much the same as the meta type node:

- It's in turn an **implicit intrusive list**
- The type of the data member is now its meta type
- `set` and `get` work with `void *` and `std::any`

Factory

Attaching meta data is also known as *welcome* `auto`:

```
template<auto Data>
meta_factory & data(const char *str) {
  using dtype = std::decay_t<decltype(std::declval<Type>().*Data)>;

  static meta_data_node node{
    str,
    types<Type>->data,
    []() { return types<dtype>; },
    [](void *inst, const std::any &value) {
      static_cast<Type *>(inst)->*Data = std::any_cast<dtype>(value);
    },
    [](void *inst) {
      return std::make_any<dtype>(static_cast<Type *>(inst)->*Data);
    }
  };

  types<Type>->data = &node;
  return *this;
}
```

Things start getting complicated despite being simple.

An erased data member

Users should not work directly with the underlying nodes:

```
struct meta_data {
  // ...

  meta_type type() const {
    return node->type();
  }

  void set(void *inst, const std::any &value) const {
    node->set(inst, value);
  }

  std::any get(void *inst) const {
    return node->get(inst);
  }

private:
    const meta_data_node *node;
};
```

Let's *reflect*

Construction phase

```
struct S { int i; double d; };

meta_factory<S>{}.reflect("MyType")
  .data<&S::i>("Int").data<&S::d>("Double");
```

Enjoy

```
S instance;
meta_data data = resolve("MyType").data("Int");
data.set(&instance, data.get(&instance));
```

Use cases

A short, yet incomplete list:

- **Serialization**: extend types from third-party libraries
- **Scripting**: glue code detached from actual types
- **Editors**: known and unknown types are welcome
- **Plugins**: types injection, but look at the boundaries
- ...
- Put **your idea** here

Questions?

References

- Meta - Header-only runtime reflection system in C++
- EnTT - Gaming meets modern C++
- RTTR - C++ Reflection Library
- Qt Meta-Object System
- A Flexible Reflection System in C++