Implementing Factory builder on top of P2320

April 8, 2021

The goal

(Semi-)automate the implementation
of the
Factory design pattern

What is meant by “Factory” here?

- Class that constructs instances of a “Product” type.
- From some external representation:
  - XML,
  - JSON,
  - YAML,
  - a GUI,
  - a scripting language,
  - a relational database,
  - . . .

Factory builder

- Framework combining the following parts
  - Meta-data
    - obtained from reflection.
  - Traits
    - units of code that handle various stages of construction,
    - specific to a particular external representation,
    - provided by user.

Factory builder

- Used meta-data
  - type name strings,
  - list of meta-objects reflecting constructors,
  - list of meta-objects reflecting constructor parameters,
  - parameter type,
  - parameter name,
  - . . .

Factory builder

- Units handling the stages of construction
  - selecting the “best” constructor,
  - invoking the selected constructor,
  - conversion of parameter values from the external representation,
  - may recursively use factories for composite parameter types.

Used reflection features

- `^T`
- `[: :]`[1]
- `meta::name_of`
- `meta::type_of`
- `meta::members_of`
- `meta::is_constructor`
- `meta::is_default_constructor`
- `meta::is_move_constructor`
- `meta::is_copy_constructor`
- `meta::parameters_of`
- `size(Range)`
- range iterators

[1] to get back the reflected type

Reflection review

- The good[2]
  - How extensive and powerful the API is
- The bad
  - Didn't find much[3]
- The ugly
  - Some of the syntax[4]

[2] great actually!

[3] some details follow

[4] but then this is a matter of personal preference

What is missing(?[5])

- The ability easily to unpack meta-objects from a range into a template, without un-reflecting them.
- I used the following workaround + `make_index_sequence`:

```
template <typename Iterator>
consteval auto advance(Iterator it, size_t n) {
  while(n-- > 0) {
    ++it;
  }
  return it;
}
```

- Details follow...

[5] maybe I overlooked something

Metaobject range unpacking

The goal is to unpack a metaobject range into a template like this:

```
template <meta::info... MO>
struct unpacked_range {
  constexpr static auto count = sizeof...(MO);
  // etc.
};
```

Metaobject range unpacking (cont.)

Unlike `meta::info` the `detail::range` type is not part of the public API, passing ranges as template parameters is not straightforward.

So we are using this helper function, which makes the whole thing less generic.

```
template <meta::info MO>
consteval auto constructors_of() {
  return meta::members_of(
    ^my_class, meta::is_constructor);
}
```

Metaobject range unpacking (cont.)

A helper:

```
template <meta::info MO, std::size_t... I>
consteval auto do_unpack_range(std::index_sequence<I...>)\
    {
  return unpacked_range<*advance(
    constructors_of<MO>().begin(),
    I)...>{};
}
```

The unpack function:

```
template <meta::info MO>
consteval auto unpack_range() {
  return do_unpack_range<MO>(
    std::make_index_sequence<
      size(constructors_of<MO>())
    >{});
}
```

Metaobject range unpacking – use case

```
template <meta::info>
class my_base;

template <typename Metaobjects>
class my_derived;
```

```
template <meta::info ... MO>
class my_derived<unpacked_range<MO...>>
 : public my_base<MO>... {
  // ...
};
```

Make ranges a “thing”

- It would be great if the ranges were:
  - either `meta::info` themselves or
  - had some public type like `meta::range`

Make ranges a “thing” (cont.)

Instead of:

```
template <typename Range>
class my_class;
```

either

```
template <meta::info Range>
class my_class;
```

or

```
template <meta::range Range>
class my_class;
```

But generally,

kudos to the implementers!

Some details on the factory builder follow[6]

[6] it there's interest

The mirror reflection utilities

- `https://github.com/matus-chochlik/mirror`
- implements the factory builder framework and some traits:
  - simple input from `iostreams`,
  - input from JSON (using RapidJSON),
  - input from a GUI (using Qt5/QML),
  - others are planned.
- plans for some additional use-cases:
  - serialization/de-serialization,
  - Python bindings,
  - ...
- There is an older implementation using manually-provided meta-data: `https://sourceforge.net/projects/mirror-lib/`

Factory builder – test classes

```
class point {
public:
    point() noexcept = default;

    point(float v) noexcept
      : _x{v} , _y{v} , _z{v} {}

    point(float x, float y, float z) noexcept
      : _x{x} , _y{y} , _z{z} {}

    // ...
private:
    float _x{0.F};
    float _y{0.F};
    float _z{0.F};
};
```

Factory builder – test classes

```
class triangle {
public:
    triangle() noexcept = default;

    triangle(const point& a, const point& b, const point&\
        c)
      : _a{a}
      , _b{b}
      , _c{c} {}

    // ...
private:
    point _a;
    point _b;
    point _c;
};
```

Factory builder – test classes

```
class tetrahedron {
public:
    tetrahedron() noexcept = default;
    tetrahedron(const triangle& base, const point& apex)
      : _base{base}
      , _apex{apex} {}

    // ...
private:
    triangle _base;
    point _apex;
};
```

Factory builder – JSON input

```json
{
    "base": {
        "a": {
            "x": 2.0,
            "y": 0.0,
            "z": 0.0
        },
        "b": {
            "x": 0.0,
            "y": 1.0,
            "z": 0.0
        },
        "c": {
            "x": 0.0,
            "y": 0.0,
            "z": 1.0
        }
    },
    "apex": {
        "v": 0.0
    }
}
```

Factory builder – JSON factory, usage

Working example[7]:

```
void print_info(const test::tetrahedron&);
const auto json_str = ...;

rapidjson::Document json_doc;
const rapidjson::ParseResult parse_result{
    json_doc.Parse(json_str)};

if(parse_result) {
    using namespace mirror;
    factory_builder<rapidjson_factory_traits> builder;
    auto factory = builder.build<test::tetrahedron>();
    print_info(factory.construct({json_doc}));

}
```

[7] https://github.com/matus-chochlik/mirror/blob/develop/example/factory/rapidjson.cpp