# 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

[2] great actually!
[3] some details follow

# What is missing(?[4])

- 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...

[4] 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

```
struct <typename Metaobjects>
struct my_derived;
```

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

# Kudos to the implementers

Some details on the factory builder follow

## 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/`