Comparing value- and type-based reflection

<table>
<thead>
<tr>
<th>Document Number</th>
<th>P2560R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>2022-02-23</td>
</tr>
<tr>
<td>Reply-to</td>
<td>Matúš Chochlík <a href="mailto:chochlik@gmail.com">chochlik@gmail.com</a></td>
</tr>
<tr>
<td>Audience</td>
<td>SG7</td>
</tr>
</tbody>
</table>
Reflection – Metaobjects

**Value-based**

Reflection uses constant values of a single built-in type:

```cpp
info x = reflect(argument);
```

**Type-based**

Reflection uses constant values of multiple types:

```cpp
auto x = reflect(argument);
```

The actual (implementation-defined) type might be:

```cpp
template <info X>
struct __metaobject {
    constexpr operator info() const {
        return X;
    }
};
```
Reflection – APIs

Either purely constexpr or template functions with non-type template parameters:

```cpp
constexpr auto foo(info mo);
```

```cpp
template <info MO>
constexpr auto bar();
```

consteval template functions taking metaobjects as function arguments:

```cpp
constexpr auto foo(metaobject auto mo);
constexpr auto bar(metaobject auto mo);
```

```cpp
template <typename T>
concept metaobject = unspecified;
```
Reflection – Implementation

**value-based**

Very fast to compile:

```cpp
consteval auto foo(info mo);
```

Somewhat slower to compile:

```cpp
template <info MO>
consteval auto bar();
```

**type-based**

Very fast to compile, but requires `consteval` conversion from `metaobject` to `info`:

```cpp
consteval auto foo(info mo);
```

Slower to compile:

```cpp
template <info MO>
consteval auto bar(___metaobject<MO> mo);
```
Reflection – Usage

Dual syntax:

use (foo (reflect (argument)));

or

use (bar <reflect (argument) >());

When to use which?

Uniform syntax:

use (foo (reflect (argument)));

and

use (bar (reflect (argument)));

value-based

type-based
Reflection – Containers

**Value-based**

Vectors, etc. must be fixed to work in consteval

```cpp
vector<info> x = members_of(...);
vector<info> y = bases_of(...);
```

Can be used with some STL algorithms, unless splicing is involved

**Type-based**

Containers (sequences) are metaobjects themselves

```cpp
auto x = get_data_members(...);
auto y = get_base_classes(...);
```

Have their own implementation of reflection-related algorithms, splicing is no problem
Reflection – Pros

value-based

- Faster to compile
- Uses less resources to compile

type-based

- Consistent and unified API
- More friendly to generic programming
- Plays better with ADL
- Better usability
- Easier to teach
Reflection – Cons

- Inconsistent API
- The `foo(...)` vs. `bar<...>()` syntax makes it less generic
- Rules when to use which, are sort of complicated and may look arbitrary
- More complicated to teach
- Issues with ADL on NTTPs

---

value-based

- Slower to compile
- Uses more resources to compile
Usability issues – the dual value-based API

```cpp
// span<meta::info>, vector<meta::info>
auto mem = members_of(^T);
auto func = // some callable, example later

The following is possible only for a subset of possible reflection operations:

std::count_if1(mem.begin(), mem.end(), func);

Specifically the `func` cannot use splicing, because `count_if` will call it as:

`func(element);`

and not as:

`func<element>();`
```

1 or any other of countless possible algorithms
Usability issues – writing generic algorithms

Users\(^2\) will want to write their own reusable algorithms, that take other functions\(^3\) as their arguments:

```cpp
consteval void my_reusable_algo(
    span<meta::info> s,
    function<bool(meta::info)> predicate,
    function<void(meta::info)> function) {
    for(auto e : s) {
        if(predicate(e) && something_else(e)) {
            function(e);
        }
    }
}
```

predicate, something_else and function cannot do splicing...

\(^2\)and library authors
\(^3\)predicates, transforms, etc.
Usability issues – supporting splicing

...to support splicing we’d have to:

```cpp
template <auto s>
consteval void my_reusable_algo(
    auto predicate,
    auto function) {
    template for(auto e : s) {
        if(predicate<e>() && something_else<e>()()) {
            function<e>();
        }
    }
}
```

making everything a template. But then this becomes slower to compile, partially defeating one of the main points of this API.
BTW, why so much focus on splicing? – some anecdotes...

- Out of these use-cases\(^4\)\(^5\):
  - enum / string conversion,
  - serialization and deserialization,
  - parsing of command line arguments into a config structure,
  - RPC stubs and skeletons,
  - generic wrapper for a REST API,
  - automated registering with a scripting engine,
  - generating UML diagrams from code,
  - fetching and converting data from an SQL database,
  - generating SQL queries from the names in an “interface” class,
  - implementation of the factory pattern.

- All but one\(^6\) required splicing

- Various forms of splicing are very common in use-cases

---

\(^4\) all implemented here: [https://github.com/matus-chochlik/mirror](https://github.com/matus-chochlik/mirror)

\(^5\) and there is a whole other presentation about the details

\(^6\) UML generation
What are we trying to do?

Determine what is the actual overhead of this:

```cpp
template <info X>
struct __metaobject {
    constexpr operator info() const { return X; }
};
concept metaobject = unspecified;

constexpr auto foo(info mo);
constexpr auto bar(metaobject auto mo);
```

compared to this:

```cpp
constexpr auto foo(info mo);
```

```cpp
template <info MO>
constexpr auto bar();
```

in a real-life scenario.
The cost of reflection in a “large-ish” project?

- **Let’s try clang**
  - Estimate the number of “things” to reflect
  - Measure the overall compilation time
  - Measure the contribution of reflection
  - Compare purely value-based and typed metaobjects
How to materialize 100’000s of metaobjects?

Use a shell script...

L=100  # number of repeats
S=1000 # sampling step size
for l in $(seq 1 ${L})
do
  N=$((l * S))
  # factorize N into three integers
  D=...; E=...; F=...

...to generate a C++ source file...

```cpp
int main() {
  return bool(qux(make_index_sequence<${D}>({[]}))) ? 0 : 1;
}
```

..., compile and measure:

```bash
time $(CXX) $(CXXFLAGS) -o /dev/null <
done
```
The boilerplate – level 1

template <size_t ... K>
consteval auto qux(index_sequence<K...>) {
  return ( ... + baz(
    integral_constant<size_t, K>{},
    make_index_sequence<$\{E\}>{})));
}
The boilerplate – level 2

template <size_t K, size_t ... J>
consteval auto baz(
    integral_constant<size_t, K>,
    index_sequence<J...>) {
    return (... + bar(
        integral_constant<size_t, K>{},
        integral_constant<size_t, J>{},
        make_index_sequence<$\{F\}$>{{}}));
}
The boilerplate – level 3

```cpp
template <size_t K, size_t J, size_t ... I>
consteval auto bar(
    integral_constant<size_t, K>,
    integral_constant<size_t, J>,
    index_sequence<I...>) {
    // Simulate the metaobject "id" as:
    // MOID =
    //   K * $(N / D) +
    //   J * $(N / (D * E)) +
    //   I;
    return /* Do something with MOID... */
}
```
The baseline

Just sum the *MOID* values at compile-time

```cpp
template <size_t K, size_t J, size_t ... I>
consteval auto bar(
    integral_constant<size_t, K>,
    integral_constant<size_t, J>,
    index_sequence<I...>) {
    return (... +
        K * $((N / D)) +
        J * $((N / (D * E))) +
        I);
}
```

Measure how long does this take to compile and subtract from “real” measurements.
Type-based metaobject & template function

template <size_t M>
struct wrapper {
  consteval operator size_t() const {
    return M;
  }
};

template <size_t M>
consteval size_t foo(wrapper<M> w) {
  return w;
}

return ( ... + foo(wrapper<MOID>{}) );
Type-based metaobject & consteval function

```cpp
template <size_t M>
struct wrapper {
    consteval operator size_t() const {
        return M;
    }
};

consteval size_t foo(size_t m) {
    return m;
}

return ( ... + foo(wrapper<MOID>{}) );
```
Value-based metaobject & template function

```cpp
template <size_t M>
consteval size_t foo() {
    return M;
}

return (... + foo<MOID>());
```
Value-based metaobject & constexpr function

```cpp
constexpr size_t foo(size_t m) {
    return m;
}
```

```cpp
return ( ... + foo(MOID));
```
Test hardware

- Old desktop\(^7\):
  - i5-2400U @ 3.10GHz (4 cores)
  - 24GB RAM

- Corporate dev laptop\(^8\):
  - i7-1185G7 @ 3.0GHz (8 cores)
  - 32GB RAM

- Mid-range gaming laptop\(^9\):
  - AMD Ryzen7 4800HS (16 cores)
  - 16GB RAM

- RPi 4B\(^{10}\)
  - ARM v7l
  - 4GB RAM

\(^7\) 2010
\(^8\) 2021
\(^9\) 2019
\(^{10}\) timeless
i5-2400 – compile time increase per N metaobjects
i5-2400 – compile time increase per 1 metaobject

![Compile time increase graph](image-url)
i5-2400 – How much faster is value-based vs. type-based
i7-1185 – compile time increase per N metaobjects
i7-1185 – compile time increase per 1 metaobject
i7-1185 – How much faster is value-based vs. type-based

![Graph showing compile-time ratio for type-based template vs. value-based template and type-based consteval vs. value-based consteval. The x-axis represents the number of metaobjects, ranging from 0 to 1M, and the y-axis represents the compile-time ratio, ranging from 0 to 4. The graph shows a comparison between the two methods as the number of metaobjects increases.]
Ryzen7-4800HS – compile time increase per N metaobjects
Ryzen7-4800HS – compile time increase per 1 metaobject

![Graph showing compile time increase per metaobject for Ryzen7-4800HS](image-url)

- Type-based template
- Type-based constexpr
- Value-based template
- Value-based constexpr
Ryzen7-4800HS – How much faster is value-based vs. type-based

![Graph](image.png)
**ARMv7** – compile time increase per N metaobjects

![Graph showing compile-time increase for different types of template and consteval methods](image-url)
ARMv7 – compile time increase per 1 metaobject

![Graph showing compile time increase per 1 metaobject for different types of templates and constevals on ARMv7. The x-axis represents the number of metaobjects ranging from 0 to 250k, and the y-axis represents the compile-time increase in microseconds. The graph compares type-based template, type-based consteval, value-based template, and value-based consteval.]
**ARMv7** – How much faster is value-based vs. type-based
What about executable sizes?

- This is boring. . .
- When the reflection-related functions are `constexpr`, the executable size stays the same regardless of the representation of metaobjects or their count.
- The test source code shown above always compiles into an executable roughly 16kB in size.
But, what if – we tried something different...

- Instead of that template gizmo instantiating metaobjects in one place in the source
- Just generate a lot of source code with many separate metaobject operations
- As in...
Type-based metaobject & template function

```cpp
template <size_t M>
struct wrapper {
    consteval operator size_t() { return M; }
};

template <size_t M>
consteval size_t foo(wrapper<M> w) {
    return w;
}

consteval int bar() {
    return static_cast<int>(
        foo(wrapper<1Z>{})+
        foo(wrapper<2Z>{})+
        // ...
        foo(wrapper<NZ>{})
    );
}
```
Type-based metaobject & `consteval` function

```cpp
template <size_t M>
struct wrapper {
    consteval operator size_t() { return M; }
};

consteval size_t foo(size_t m) {
    return m;
}

consteval int bar() {
    return static_cast<int>(
        foo(wrapper<1Z>{}) +
        foo(wrapper<2Z>{}) +
        // ...
        foo(wrapper<NZ>{})
    );
}
```
Value-based metaobject & template function

```cpp
template <size_t M>
constexpr size_t foo() {
    return M;
}

constexpr int bar() {
    return static_cast<int>(
        foo<1Z>() +
        foo<2Z>() +
        // ...
        foo<NZ>();
    }
```
Value-based metaobject & consteval function

```cpp
consteval size_t foo(size_t m) {
    return m;
}

consteval int bar() {
    return static_cast<int>(
        foo(1Z) +
        foo(2Z) +
        // ...
        foo(NZ));
}
```
i5-2400 – compile time increase per N metaobjects
**i5-2400 – compile time increase per 1 metaobject**

![Graph showing compile time increase per 1 metaobject for different types of templates and consteval methods.](image)

- **Yellow line**: type-based template
- **Green line**: type-based consteval
- **Blue line**: value-based template
- **Gray line**: value-based consteval

The graph illustrates the increase in compile time as the number of metaobjects increases, with a focus on the performance differences between the various template and consteval methods.
i5-2400 – How much faster is value-based vs. type-based
Ryzen7-4800HS – compile time increase per N metaobjects
**Ryzen7-4800HS** – compile time increase per 1 metaobject
Ryzen7-4800HS – How much faster is value-based vs. type-based

![Graph showing compile-time ratio for type-based template vs. value-based template]

- Compile-time ratio for type-based template / value-based template
- Compile-time ratio for type-based consteval / value-based consteval
ARMv7 – compile time increase per N metaobjects
**ARMv7** – compile time increase per 1 metaobject

- **Type-based template**
- **Type-based consteval**
- **Value-based template**
- **Value-based consteval**
ARMv7 – How much faster is value-based vs. type-based

![Graph showing compile-time ratio for ARMv7 comparing value-based vs. type-based templates and consteval]

- Yellow line: type-based template / value-based template
- Blue line: type-based constexpr / value-based constexpr

Number of metaobjects: 0, 1k, 2k, 3k, 4k, 5k, 6k, 7k, 8k

Compile-time ratio: 0, 1, 2, 3, 4, 5, 6, 7, 8
Why the *huge* difference?

- Just some guesses. . .
  - In the second setup, each metaobject is an individually-parsed expression
  - Different than just multiple instantiations
  - Different source locations, etc.
- Even in the value-based cases the compile-times grow non-linearly
- This is *not* how you typically use reflection in real-life scenarios
Representing the reflection “operator”

- Above we have just used integer literals to represent metaobject “ids”
- What if we used a template function\textsuperscript{11} to simulate the reflection expression
- What would be the effects on the compile-times?

\textsuperscript{11} with NTTP
Representing the reflection “operator” – (cont.)

```cpp
template <size_t I>
consteval auto reflect() {
    return I;
}

foo(reflect<NZ>());
// and
foo<reflect<NZ>()>();
```

```cpp
template <size_t I>
consteval wrapper<I> reflect() {
    return {};
}

foo(reflect<NZ>());
```
Ryzen7-4800HS – compile time increase per N metaobjects

![Graph showing compile time increase per N metaobjects for different template types.](image)
Ryzen7-4800HS – compile time increase per 1 metaobject
Ryzen7-4800HS – How much faster is value-based vs. type-based

Graph showing the compile-time ratio of type-based template to value-based template and type-based constexpr to value-based constexpr against the number of metaobjects.
Estimating number of declarations in clang

Let's try documented declarations

Edit doxygen-cfg.in:

- GENERATE_XML = \textit{NO}
+ GENERATE_XML = \textit{YES}

Configure:

\begin{verbatim}
cmake \ \ -D LLVM\_ENABLE\_DOXYGEN=On \ \ ...
\end{verbatim}

Generate Doxygen docs:

\texttt{ninja doxygen-clang}

Merge into a single XML file clang.xml:

\texttt{xsltproc combine.xslt index.xml > clang.xml}
Counting documented declarations in clang

Create `count.xslt`:

```xml
<?xml version="1.0" encoding="utf8"?>
<xsl:stylesheet version = '1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <xsl:value-of select="count(
      descendant::compounddef
      | descendant::member
      | descendant::value
      | descendant::para
      | descendant::param
    )"/>
  </xsl:template>
</xsl:stylesheet>
```

12 structs, classes, enums, ...
13 data members, member functions, enumerators, ...
14 enumerator values, default arguments, ...
15 function/constructor/operator parameters, ...
16 template parameters, ...
Counting documented declarations in clang

Calculate!

```bash
xsltproc \ count.xslt \ clang.xml
```

The result:

379091

- That’s for version 15.0.0
- Around FEB-05-2022
- Round that up to 400’000, 500’000 or even 1’000’000
- Let’s assume we want to reflect every single declaration
Clean build of clang

Edit toolchain.cmake:

```bash
set(LLVM_USE_LINKER lld)
set(CMAKE_EXE_LINKER_FLAGS -fuse-ld=${LLVM_USE_LINKER})
set(CMAKE_SHARED_LINKER_FLAGS -fuse-ld=${LLVM_USE_LINKER})
```

Configure:

```bash
cmake \
  -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" \
  -DLLVM_ENABLE_RUNTIMES="libcxx;libcxxabi" \
  -DLLVM_TOOLCHAIN_FILE="toolchain.cmake" \
  ...
```

Build and measure elapsed time:

```bash
time ninja install install-cxx install-cxxabi
```
Clean build of clang

Results:

<table>
<thead>
<tr>
<th>CPU:</th>
<th>i5-2400</th>
<th>i7-1185</th>
<th>Ryzen 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>122m25,943s</td>
<td>66m59,909s</td>
<td>34m45,899s</td>
</tr>
<tr>
<td>user</td>
<td>433m50,123s</td>
<td>510m55,382s</td>
<td>525m16,660s</td>
</tr>
<tr>
<td>sys</td>
<td>11m22,881s</td>
<td>12m52,287s</td>
<td>17m5,738s</td>
</tr>
</tbody>
</table>

Added, rounded and converted to seconds:

<table>
<thead>
<tr>
<th>CPU:</th>
<th>i5-2400</th>
<th>i7-1185</th>
<th>Ryzen 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>real-time</td>
<td>7346s</td>
<td>4020s</td>
<td>2086s</td>
</tr>
<tr>
<td>cpu-time (user+sys)</td>
<td>27313s</td>
<td>31427s</td>
<td>32543s</td>
</tr>
</tbody>
</table>
Compared to build-time with 400’000 metaobjects

Compile-time of a typical `clang` build vs. compile-time spent on materializing 400’000 metaobjects:

<table>
<thead>
<tr>
<th></th>
<th>CPU:</th>
<th>i5-2400</th>
<th>i7-1185</th>
<th>Ryzen 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang:</td>
<td></td>
<td>27313s</td>
<td>31427s</td>
<td>32543s</td>
</tr>
<tr>
<td></td>
<td>type-based template</td>
<td>115.9s</td>
<td>48.8s</td>
<td>62.4s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.42%</td>
<td>0.16%</td>
<td>0.19%</td>
</tr>
<tr>
<td></td>
<td>type-based constexpr</td>
<td>111.3s</td>
<td>53.4s</td>
<td>62.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.41%</td>
<td>0.16%</td>
<td>0.19%</td>
</tr>
<tr>
<td></td>
<td>value-based template</td>
<td>36.3s</td>
<td>16.5s</td>
<td>19.0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.13%</td>
<td>0.05%</td>
<td>0.06%</td>
</tr>
<tr>
<td></td>
<td>value-based constexpr</td>
<td>50.3s</td>
<td>27.4s</td>
<td>29.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.18%</td>
<td>0.09%</td>
<td>0.09%</td>
</tr>
</tbody>
</table>
Conclusions

- The typical compile-time overhead of materializing a metaobject is on the order of tens or hundreds of microseconds.
- The type-based metaobject representation is between 2x and 6x\(^1\) slower to compile compared to the purely value-based representation.

\(^1\)in the worst “copy-paste” use-case
Conclusions – (cont.)

- Most typical reflection use-cases don’t require reflecting every declaration in a project.
- Even if reflecting almost everything, the overhead compared to total build time is a fraction of a percent even in the worst case.
- For projects similar in complexity to clang, this results in 1-2 minutes added to several hours of compilation-time.
Conclusions – (cont.)

• Some of the compile-time advantage of value-based API disappears, when splicing is involved
• In the value-based API splicing requires passing metaobject as non-type template arguments
• Splicing is quite common in various use-cases
• Combining reflection with template metaprogramming is common as well
The big question

Is the improvement in compile-time worth the decrease in usability of the value-based reflection API?