

# Reflection – Metaobjects

value-based

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

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

Reflection uses constant values of multiple types:

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

type-based

The actual (implementation-defined) type might be:

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

# Reflection – APIs

value-based

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

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

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

type-based

constexpr template functions taking metaobjects as function arguments:

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

```
template <typename T>  
concept metaobject = unspecified;
```

# Reflection – Implementation

value-based

Very fast to compile:

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

Somewhat slower to compile:

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

type-based

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

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

Slower to compile:

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

# Reflection – Usage

value-based

Dual syntax:

```
use(foo(reflect(argument)));
```

or

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

When to use which?

type-based

Uniform syntax:

```
use(foo(reflect(argument)));
```

and

```
use(bar(reflect(argument)));
```

# Reflection – Containers

value-based

Vectors, etc. must be fixed to work in consteval.

```
span<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.

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

Have their own implementation of reflection-related algorithms, splicing is no problem.



# Reflection – Cons

value-based

- 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

---

type-based

- Slower to compile
- Uses more resources to compile

# Usability issues – the dual value-based API

```
// 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>();
```

---

<sup>1</sup>or any other of countless possible algorithms



# Usability issues – writing generic algorithms

Users<sup>2</sup> will want to write their own reusable algorithms, that take other functions<sup>3</sup> as their arguments:

```
constexpr 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...

---

<sup>2</sup>and library authors

<sup>3</sup>predicates, transforms, etc.

# Usability issues – supporting splicing

...to support splicing we'd have to:

```
template <auto S>
constexpr 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, defeating one of the main points of this API.

# BTW, why so much focus on splicing? – some anecdotes...

- Out of these use-cases<sup>45</sup>
  - 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<sup>6</sup> required splicing
- Various forms of splicing are *very* common in use-cases

---

<sup>4</sup>all implemented here: <https://github.com/matus-chochlik/mirror>

<sup>5</sup>and there is a whole other presentation about the details

<sup>6</sup>UML generation

# What are we trying to do?

Determine what is the actual overhead of this:

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

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

compared to this:

```
consteval auto foo(info mo);

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

# 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...

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

..., compile and measure:

```
time $(CXX) $(CXXFLAGS) -o /dev/null $<
done
```

# The boilerplate – level 1

```
template <size_t ... K>
constexpr 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>
constexpr 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

```
template <size_t K, size_t J, size_t ... I>
constexpr 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

```
template <size_t K, size_t J, size_t ... I>
constexpr 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

```
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>{ }));
```

```
return ( ... + foo<MOID>() );
```

## Value-based metaobject & consteval function

```
constexpr size_t foo(size_t i) {
    return i;
}
```

```
return ( ... + foo(MOID));
```

# Test hardware

- Old desktop<sup>7</sup>:
  - i5-2400U @ 3.10GHz (4 cores)
  - 24GB RAM
- Corporate dev laptop<sup>8</sup>:
  - i7-1185G7 @ 3.0GHz (8 cores)
  - 32GB RAM
- Mid-range gaming laptop<sup>9</sup>:
  - AMD Ryzen7 4800HS (16 cores)
  - 16GB RAM
- RPi 4B
  - ARM v7l
  - 4GB RAM

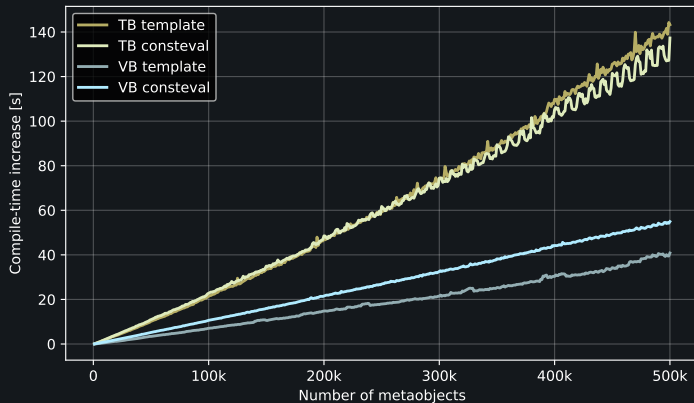
---

<sup>7</sup>2010

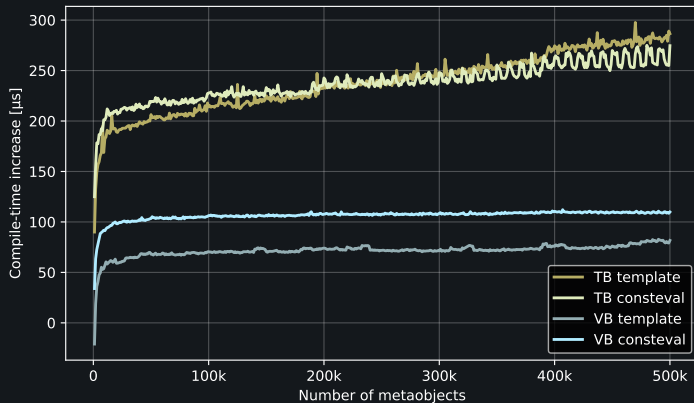
<sup>8</sup>2021

<sup>9</sup>2019

# i5-2400 – compile time increase per N metaobjects

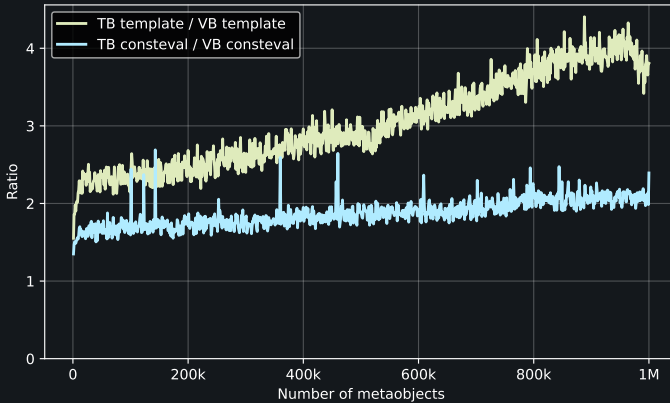


# i5-2400 – compile time increase per 1 metaobject

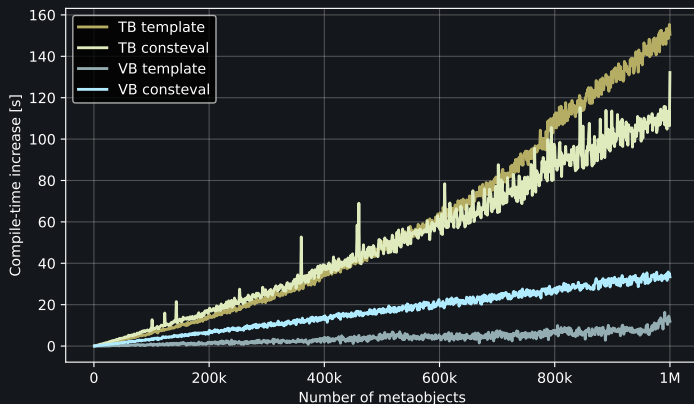




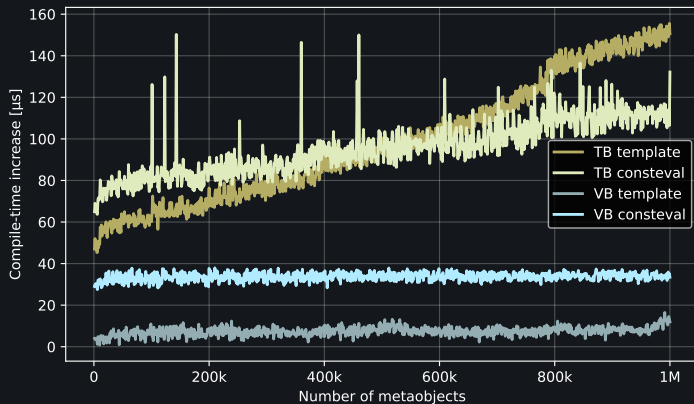
# i5-2400 – How much faster is VB vs. TB



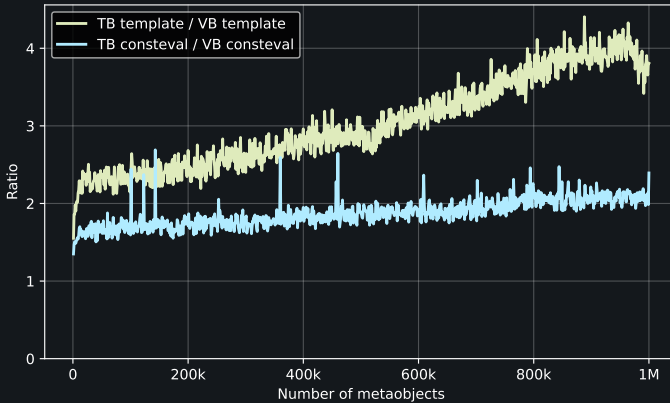
# i7-1185 – compile time increase per N metaobjects



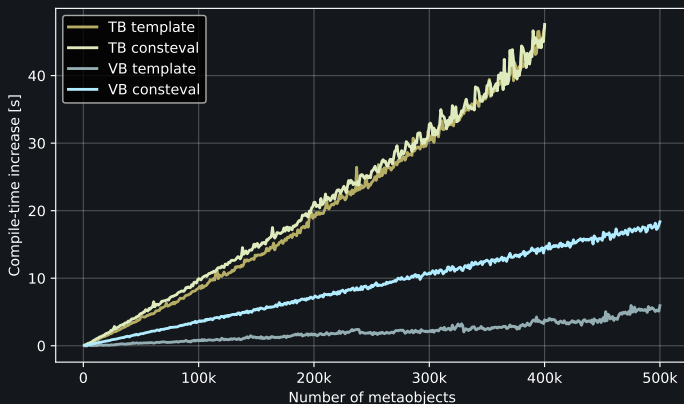
# i7-1185 – compile time increase per 1 metaobject



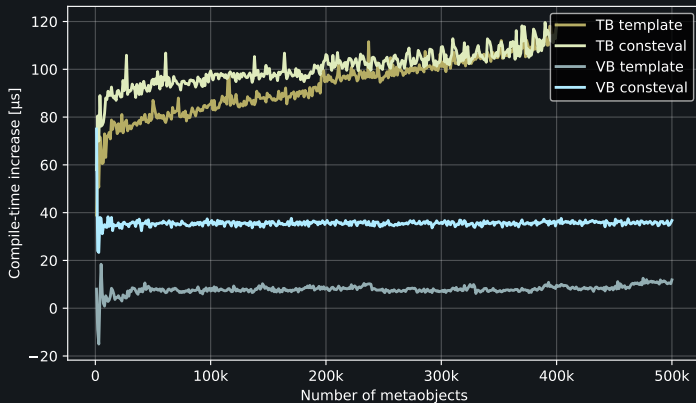
# i7-1185 – How much faster is VB vs. TB



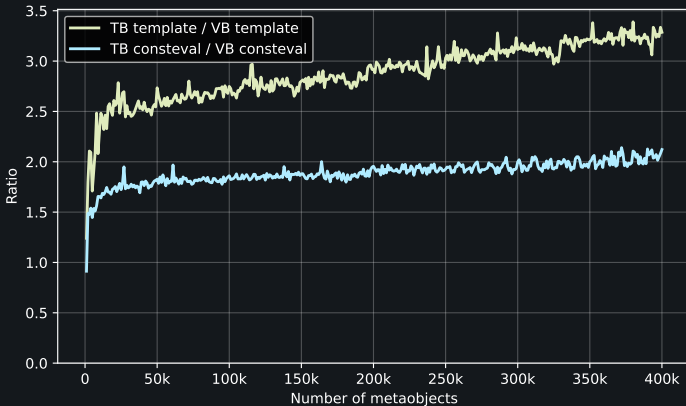
# Ryzen7-4800HS– compile time increase per N metaobjects



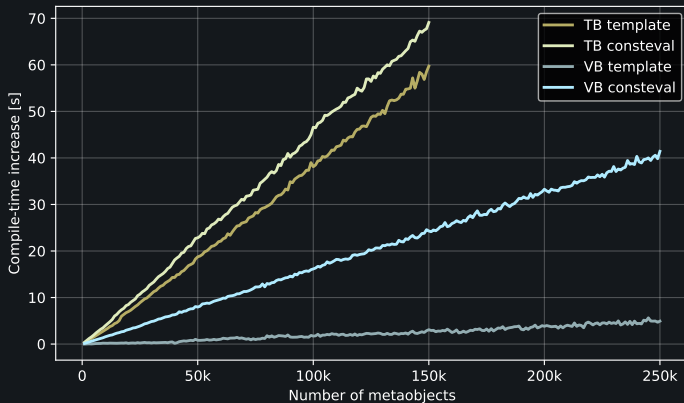
# Ryzen7-4800HS – compile time increase per 1 metaobject



# Ryzen7-4800HS – How much faster is VB vs. TB

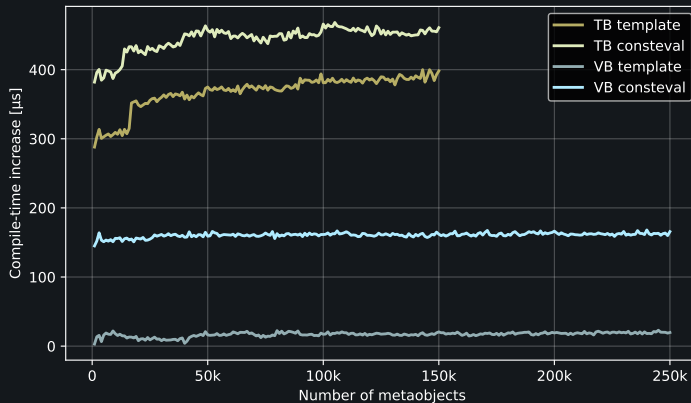


# ARMv7– compile time increase per N metaobjects

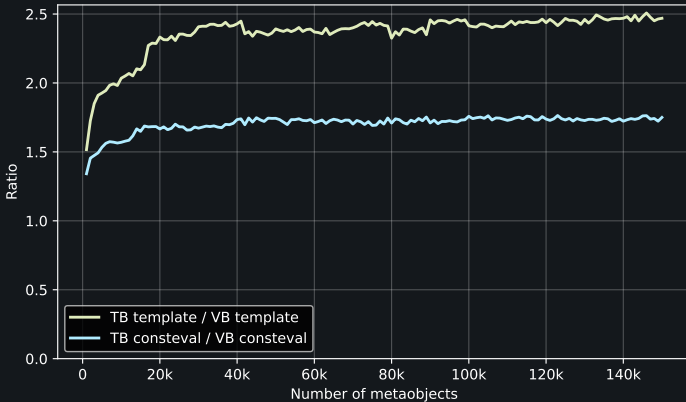




# ARMv7– compile time increase per 1 metaobject



# ARMv7 – How much faster is VB vs. TB



## 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

## 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

# Estimating number of declarations in clang

Let's try *documented* declarations

Edit doxygen-cfg.in:

```
- GENERATE_XML    = NO
+ GENERATE_XML    = YES
```

Configure:

```
cmake \
  -DLLVM_ENABLE_DOXYGEN=On \
  ...
```

Generate Doxygen docs:

```
ninja doxygen-clang
```

Merge into a single XML file clang.xml:

```
xsltproc combine.xslt index.xml > clang.xml
```

# Counting documented declarations in clang

Create count.xslt:

```
<?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::compounddef10 |
      descendant::member11 |
      descendant::value12 |
      descendant::para13 |
      descendant::param14)
    " />
  </xsl:template>
</xsl:stylesheet>
```

<sup>10</sup>struct, class, enum, ...

<sup>11</sup>data members, member functions, enumerators, ...

<sup>12</sup>enumerator values, default arguments, ...

<sup>13</sup>function/constructor/operator parameters, ...

<sup>14</sup>template parameters, ...

# Counting documented declarations in clang

Calculate!

```
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:

```
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:

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

Build and measure elapsed time:

```
time ninja -j 16 install install-cxx install-cxxabi
```



# Clean build of clang

## Results:

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

Added, rounded and converted to seconds:

CPU:	i5-2400	i7-1185	Ryzen 7
real-time	7346s	4020s	2086s
cpu-time (user+sys)	27313s	31427s	32543s

## Compared to build-time with 400'000 metaobjects

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

CPU:	i5-2400	i7-1185	Ryzen 7
clang:	27313s	31427s	32543s
400k MO (TB-template):	115.9s	48.8s	62.4s
	0.42%	0.16%	0.19%
400k MO (TB-consteval):	111.3s	53.4s	62.7
	0.41%	0.16%	0.19%
400k MO (VB-template):	36.3s	16.5s	19.0s
	0.13%	0.05%	0.06%
400k MO (VB-consteval):	50.3s	27.4s	29.6s
	0.18%	0.09%	0.09%

## 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 4x slower to compile compared to the purely value-based representation





# The big question

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