

Serialization in C++ has never been easier!
But wait, there's more...

Pavel Novikov

 @cpp_ape

Align Technology R&D

The logo for Align Technology, featuring the word "align" in a dark grey, lowercase, sans-serif font. A small blue dot is positioned above the letter "i".

We'll talk about

- Reflection
- Metaprogramming

We'll talk about

- Reflection
- Metaprogramming
- It's not in C++ 20

We'll talk about

- Reflection
- Metaprogramming
- It's not in C++ 20
 - *maybe* will be in C++ 23

We'll talk about

- Reflection
- Metaprogramming
- It's not in C++ 20
 - *maybe* will be in C++ 23
 - probably will be in C++ 26

We'll talk about

- Reflection
- Metaprogramming
- It's not in C++ 20
 - *maybe* will be in C++ 23
 - probably will be in C++ 26
 - mere mortals will be able to use it in **2027** ... probably...

METAPROGRAMMING



We'll talk about

- Reflection
- Metaprogramming
- It's not in C++ 20
 - *maybe* will be in C++ 23
 - probably will be in C++ 26
 - mere mortals will be able to use it in 2027 ... probably...
 - *everything* can change entirely...

METAPROGRAMMING



It's




```

namespace meta = std::experimental::meta;

template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m })), meta::name_of(%{ m }));
            };
    }
}

```


Serialization (without fatal flaw)

- Review in 5 minutes



Serialization (without fatal flaw)

```
struct Foo {  
    std::optional<int> empty;  
    int number = 42;  
    std::string string = "bleh!";  
    std::vector<int> array = {  
        1,  
        2,  
        3  
    };  
    struct {  
        std::string iAm = "nested!";  
    } nestedStruct;  
    struct {} emptyStruct;  
};
```

```
{  
    "empty": null,  
    "number": 42,  
    "string": "bleh!",  
    "array": [  
        1,  
        2,  
        3  
    ],  
    "nestedStruct": {  
        "iAm": "nested!"  
    },  
    "emptyStruct": {}  
}
```

Serialization

```
serialize(std::cout, Foo{});
```

```
struct Foo {  
    std::optional<int> empty;  
    int number = 42;  
    std::string string = "bleh!";  
    std::vector<int> array = {  
        1,  
        2,  
        3  
    };  
    struct {  
        std::string iAm = "nested!";  
    } nestedStruct;  
    struct {} emptyStruct;  
};
```

```
{  
    "empty": null,  
    "number": 42,  
    "string": "bleh!",  
    "array": [  
        1,  
        2,  
        3  
    ],  
    "nestedStruct": {  
        "iAm": "nested!"  
    },  
    "emptyStruct": {}  
}
```

Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```

Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```

Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```

Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```


Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```

Serialization (without fatal flaw)

```
struct SerializationHelper {  
    //...  
    template<typename T>  
    void printValue(const std::vector<T> &vector, size_t indent);  
    template<typename T>  
    void printValue(const std::optional<T> &value, size_t indent);  
    template<typename T>  
    void printValue(const T &value, size_t indent);  
    template<typename T>  
    static void iterateMembersHelper(std::ostream &stream, const T &value, size_t indent);  
};  
  
template<typename T>  
void serialize(std::ostream &stream, const T &value) {  
    SerializationHelper{ stream }.printValue(value, /*indent*/ 0);  
}
```

Serialization (without fatal flaw)

```
int number = 42;  
serialize(std::cout, number);
```

```
std::string string = "#cpponsea";  
serialize(std::cout, string);
```

```
std::vector<std::string> array = {  
    "one",  
    "two",  
    "three"  
};  
serialize(std::cout, array);
```

42

"#cpponsea"

[
 "one",
 "two",
 "three"
]

Serialization (without fatal flaw)

```
template<typename T>
void printValue(const T &value, size_t indent) {
    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

Serialization (without fatal flaw)

```
template<typename T>
void printValue(const T &value, size_t indent) {
    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

Serialization (without fatal flaw)

```
template<typename T>
void printValue(const T &value, size_t indent) {
    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

Serialization (without fatal flaw)

```
template<typename T>
void printValue(const T &value, size_t indent) {
    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

Serialization (without fatal flaw)

```
template<typename T>
void printValue(const T &value, size_t indent) {
    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```


Serialization (without fatal flaw)

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};
```

```
serialize(std::cout, Bar{});
```

Serialization (without fatal flaw)

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};
```

```
serialize(std::cout, Bar{});
```

error : use of undeclared identifier 'iterateMembers'

Serialization (without fatal flaw)

```
template<typename T>
static void iterateMembersHelper(std::ostream &stream,
                                const T &value,
                                size_t indent) {
    static_assert(std::is_class_v<T>);
    stream << '{';
    SerializationHelper helper{ stream };
    auto visit = [&helper, indent = indent + Indent](const auto& value,
                                                       std::string_view name) {
        helper.print(value, name, indent);
    };
    iterateMembers(visit, value);
    if (!helper.empty)
        helper.printNewLineAndIndent(indent);
    stream << '}';
}
```

Serialization (without fatal flaw)

```
template<typename T>
static void iterateMembersHelper(std::ostream &stream,
                                const T &value,
                                size_t indent) {
    static_assert(std::is_class_v<T>);
    stream << '{';
    SerializationHelper helper{ stream };
    auto visit = [&helper, indent = indent + Indent](const auto& value,
                                                       std::string_view name) {
        helper.print(value, name, indent);
    };
    iterateMembers(visit, value);
    if (!helper.empty)
        helper.printNewLineAndIndent(indent);
    stream << '}';
}
```

Serialization (without fatal flaw)

```
template<typename T>
static void iterateMembersHelper(std::ostream &stream,
                                const T &value,
                                size_t indent) {
    static_assert(std::is_class_v<T>);
    stream << '{';
    SerializationHelper helper{ stream };
    auto visit = [&helper, indent = indent + Indent](const auto& value,
                                                       std::string_view name) {
        helper.print(value, name, indent);
    };
    iterateMembers(visit, value);
    if (!helper.empty)
        helper.printNewLineAndIndent(indent);
    stream << '}';
}

template<typename T>
void print(const T &value,
           std::string_view name,
           size_t indent) {
    //...
    stream << std::quoted(name) << ':';
    printValue(value, indent);
}
```

Serialization (without fatal flaw)

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};  
  
template<typename Visitor>  
void iterateMembers(Visitor &&visit, const Bar &bar) {  
    visit(bar.number, "number");  
    visit(bar.string, "string");  
    visit(bar.array, "array");  
}  
  
serialize(std::cout, Bar{});
```

Serialization (without fatal flaw)

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};  
  
template<typename Visitor>  
void iterateMembers(Visitor &&visit, const Bar &bar) {  
    visit(bar.number, "number");  
    visit(bar.string, "string");  
    visit(bar.array, "array");  
}  
  
serialize(std::cout, Bar{});
```

Serialization (without fatal flaw)

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};  
  
template<typename Visitor>  
void iterateMembers(Visitor &&visit, const Bar &bar) {  
    visit(bar.number, "number");  
    visit(bar.string, "string");  
    visit(bar.array, "array");  
}  
  
serialize(std::cout, Bar{});
```

```
{  
    "number": 42,  
    "string": "boring string",  
    "array": [  
        1,  
        2,  
        3  
    ]  
}
```


Reflection



reflexpr and meta::info

```
int value = 42;  
constexpr meta::info info = reflexpr(value);  
  
if (meta::is_named(info))  
    std::cout << meta::name_of(info);
```

outputs:
value

reflexpr and meta::info

```
constexpr meta::info info = reflexpr(int);
```

```
if (meta::is_named(info))  
    std::cout << meta::name_of(info);
```

outputs:

int

reflexpr and meta::info

```
template<typename T>
void printValue(const T &value, size_t indent) {

    if constexpr (std::is_same_v<T, bool>) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (std::is_same_v<T, int>) {
        stream << value;
    }
    else if constexpr (std::is_same_v<T, std::string>) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

reflexpr and meta::info

```
template<typename T>
void printValue(const T &value, size_t indent) {
    constexpr auto type = reflexpr(T);
    if constexpr (type == reflexpr(bool)) {
        stream << (value ? "true" : "false");
    }
    else if constexpr (type == reflexpr(int)) {
        stream << value;
    }
    else if constexpr (type == reflexpr(std::string)) {
        stream << std::quoted(value);
    }
    else {
        iterateMembersHelper(stream, value, indent);
    }
}
```

reflexpr and meta::info

```
constexpr size_t dataMemberCount(meta::info info) {  
    size_t count = 0;  
    for (auto m : meta::data_member_range(info))  
        ++count;  
    return count;  
}
```

reflexpr and meta::info

```
constexpr size_t dataMemberCount(meta::info info) {  
    size_t count = 0;  
    for (auto m : meta::data_member_range(info))  
        ++count;  
    return count;  
}
```



meta::data_members_of is not yet implemented

reflexpr and meta::info

```
struct Widget {  
    int foo;  
    std::string bar;  
    std::vector<int> baz;  
};
```

```
constexpr auto info = reflexpr(Widget);  
std::cout << meta::name_of(info) << " has "  
           << dataMemberCount(info) << " data members";
```

outputs:

Widget has 3 data members

Reifiers

code

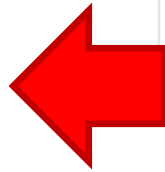
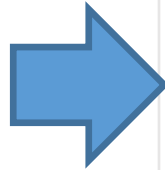
```
int value = 42;
```

value

metainformation

```
constexpr auto info =  
  reflexpr(value);
```

```
idexpr(info)
```



idexpr

```
int value = 42;  
constexpr auto info = reflexpr(value);  
  
std::cout << idexpr(info);
```

outputs:

42

idexpr

```
int value = 42;  
constexpr auto info = reflexpr(value);  
  
std::cout << idexpr<value>(info);
```

outputs:

42

unqualid

```
int value = 42;
```

```
std::cout << unqualid("value");
```

outputs:

42

unqualid

```
int value = 42;
```

```
std::cout << unqualid(value);
```

outputs:

42

unqualid

```
Widget w;
```

```
w.bar = "text";
```

```
std::cout << w.unqualid("bar");
```

outputs:

text

unqualid

```
Widget w;
```

```
w.bar = "text";
```

```
std::cout << w.unqualid<bar>("bar");
```

outputs:

text

Reifiers

typename	<code>constexpr auto info = reflexpr(int); typename(info) value = 42;</code>
----------	--

sizeof	<code>const int Const = 23; std::cout << sizeof(reflexpr(Const));</code>
--------	--

templarg	<code>template<typename T> void foo(); foo<templarg(reflexpr(Const))>(); template<int V> void bar(); bar<templarg(reflexpr(Const))>();</code>
----------	--

Reifiers

typename	<code>constexpr auto info = reflexpr(int);</code> <code>typename int(info) value = 42;</code>
----------	--

sizeof	<code>const int Const = 23;</code> <code>std::cout << sizeof(reflexpr(Const));</code>
--------	--

templarg	<code>template<typename T> void foo();</code> <code>foo<templarg(reflexpr(Const))>();</code> <code>template<int V> void bar();</code> <code>bar<templarg(reflexpr(Const))>();</code>
----------	---

Reifiers

typename	<code>constexpr auto info = reflexpr(int);</code> <code>typename(int)(info) value = 42;</code>
----------	---

sizeof	<code>const int Const = 23;</code> <code>std::cout << sizeof(reflexpr(Const));</code>
--------	--

template	<code>template<typename T> void foo();</code> <code>foo<template(reflexpr(Const))>();</code> <code>template<int V> void bar();</code> <code>bar<template(reflexpr(Const))>();</code>
----------	---

Reifiers

typename `constexpr auto info = reflexpr(int);`
 `typename(int)(info) value = 42;`

valueof `const int Const = 23;`
 `std::cout << valueof(reflexpr(Const));`

templarg `template<typename T> void foo();`
 `foo<templarg(reflexpr(Const))>();`

 `template<int V> void bar();`
 `bar<templarg(reflexpr(Const))>();`

Reifiers

typename `constexpr auto info = reflexpr(int);`
 `typename(int)(info) value = 42;`

valueof `const int Const = 23;`
 `std::cout << valueof(reflexpr(Const));`

templarg `template<typename T> void foo();`
 `foo<templarg(reflexpr(Const))>();`

`template<int V> void bar();`
 `bar<templarg(reflexpr(Const))>();`

Reifiers

typename `constexpr auto info = reflexpr(int);`
 `typename(int)(info) value = 42;`

valueof `const int Const = 23;`
 `std::cout << valueof(reflexpr(Const));`

templarg `template<typename T> void foo();`
 `foo<templarg(reflexpr(Const))>();`

`template<int V> void bar();`
 `bar<templarg(reflexpr(Const))>();`

Reflection

```
template<typename Enum>
std::string_view toString(Enum value) {
    constexpr auto info = reflexpr(Enum);
    for constexpr (auto m : meta::members_of(info))
        if (value == valueof(m))
            return meta::name_of(m);
    return {};
}
```

Reflection

```
template<typename Enum>
std::string_view toString(Enum value) {
    constexpr auto info = reflexpr(Enum);
    for constexpr (auto m : meta::members_of(info))
        if (value == valueof(m))
            return meta::name_of(m);
    return {};
}
```

Reflection

```
template<typename Enum>
std::string_view toString(Enum value) {
    constexpr auto info = reflexpr(Enum);
    template for constexpr auto m : meta::members_of(info)
        if (value == valueof(m))
            return meta::name_of(m);
    return {};
}
```


Reflection

```
template<typename Enum>
std::string_view toString(Enum value) {
    constexpr auto info = reflexpr(Enum);
    template for constexpr auto m : meta::members_of(info)
        if (value == valueof(m))
            return meta::name_of(m);
    return {};
}
```

does not currently work

Reflection

```
template<typename Enum>
constexpr size_t getEnumMemberCount() {
    auto info = reflexpr(Enum);
    size_t count = 0;
    for (auto m : meta::members_of(info))
        ++count;
    return count;
}
```

Reflection


```
template<typename Enum>
constexpr auto getEnumMemberName(size_t index) {
    auto info = reflexpr(Enum);
    auto it = meta::members_of(info).begin();
    for (; index; --index)
        ++it;
    return meta::name_of(*it);
}
```

Reflection

```
template<typename Enum, size_t... I>
std::string_view toStringImpl(Enum value,
                              std::index_sequence<I...>){
    std::string_view name;
    (void)((
        value == Enum::unqualid(getEnumMemberName<Enum>(I)) ?
            (name = getEnumMemberName<Enum>(I), true) :
            false
        ) || ...
    );
    return name;
}
```

Reflection

```
template<typename Enum, size_t... I>
std::string_view toStringImpl(Enum value,
                               std::index_sequence<I...>){
    std::string_view name;
    (void)((
        value == Enum::unqualid(getEnumMemberName<Enum>(I)) ?
            (name = getEnumMemberName<Enum>(I), true) :
            false
        ) || ...
    );
    return name;
}
```



fold expression

Reflection

```
template<typename Enum, size_t... I>
std::string_view toStringImpl(Enum value,
                               std::index_sequence<I...>){
    std::string_view name;
    (void)((
        value == Enum::unqualid(getEnumMemberName<Enum>(I)) ?
            (name = getEnumMemberName<Enum>(I), true) :
            false
        ) || ...
    );
    return name;
}
```

Reflection

```
template<typename Enum, size_t... I>
std::string_view toStringImpl(Enum value,
                               std::index_sequence<I...>){
    std::string_view name;
    (void)((
        value == Enum::unqualid(getEnumMemberName<Enum>(I)) ?
            (name = getEnumMemberName<Enum>(I), true) :
            false
        ) || ...
    );
    return name;
}
```

Reflection

```
template<typename Enum, size_t... I>
std::string_view toStringImpl(Enum value,
                              std::index_sequence<I...>){
    std::string_view name;
    (void)((
        value == Enum::unqualid(getEnumMemberName<Enum>(I)) ?
            (name = getEnumMemberName<Enum>(I), true) :
            false
        ) || ...
    );
    return name;
}
```


Reflection

```
template<typename Enum>
concept Enumeration = std::is_enum_v<Enum>;

template<Enumeration Enum>
std::string_view toString(Enum value) {
    using IndexSequence =
        std::make_index_sequence<getEnumMemberCount<Enum>()>;
    return toStringImpl(value, IndexSequence{});
}
```

Reflection

```
enum class Enum { One, Two, Three };
```

```
std::cout << toString(Enum::One) << '\n';
```

```
std::cout << toString(Enum::Two) << '\n';
```

```
std::cout << toString(Enum::Three) << '\n';
```

outputs:

One

Two

Three

Reflection

```
enum class Enum { One, Two, Three };
```

```
std::cout << toString(Enum::One) << '\n';
```

```
std::cout << toString(Enum::Two) << '\n';
```

```
std::cout << toString(Enum::Three) << '\n';
```

outputs:

One

Two

Three



**I CAN HAS
SERIALIZATION?**



Serialization only using reflection

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    constexpr auto type = reflexpr(std::decay_t<T>);
    template for (constexpr auto m :
                  meta::data_member_range(type))
        visit(value.idexpr(m), meta::name_of(m));
}
```

Serialization only using reflection

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    constexpr auto type = reflexpr(std::decay_t<T>);
    template for (constexpr auto m :
                  meta::data_member_range(type))
        visit(value.idexpr(m), meta::name_of(m));
}
```

does not currently work

Serialization only using reflection

```
constexpr size_t dataMemberCount(meta::info info) {  
    size_t count = 0;  
    for (auto m : meta::data_member_range(info))  
        ++count;  
    return count;  
}
```

Serialization only using reflection

```
template<typename T>
constexpr auto getMemberName(size_t index) {
    auto it = meta::data_member_range(reflexpr(T)).begin();
    for (; index; --index)
        ++it;
    return meta::name_of(*it);
}
```


Serialization only using reflection

```
template<typename Visitor, typename T, size_t... I>
void iterateMembersImpl(Visitor &&visit,
                        T && object,
                        std::index_sequence<I...>) {
    using Type = std::decay_t<T>;
    (
        visit(object.unqualid(getMemberName<Type>(I)),
              getMemberName<Type>(I))
        , ...
    );
}
```

Serialization only using reflection

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&object) {
    using Type = std::decay_t<T>;
    using IndexSequence =
        std::make_index_sequence<
            dataMemberCount(reflexpr(Type))>;
    iterateMembersImpl(std::forward<Visitor>(visit),
                      std::forward<T>(object),
                      IndexSequence{});
}
```

Serialization only using reflection

```
struct Bar {  
    int number = 42;  
    std::string string = "boring string";  
    std::vector<int> array = { 1, 2, 3 };  
};
```

```
serialize(std::cout, Bar{});
```

```
{  
  "number": 42,  
  "string": "boring string",  
  "array": [  
    1,  
    2,  
    3  
  ]  
}
```

A meme image featuring a ginger and white cat as the central figure. The cat is depicted with a wide-eyed, open-mouthed expression of shock or awe. It is wearing a dark blue space suit with white gloves and boots. The background is a deep black space filled with numerous white stars and a faint, glowing nebula. The word "Metaprogramming" is written in a large, white, sans-serif font across the middle of the image, partially obscuring the cat's chest. In the bottom right corner, the text "[adult swim]" is visible in a smaller, white, sans-serif font.

Metaprogramming

[adult swim]

Metaprogramming

- injection
- fragments
- metaprograms

Injection

consteval -> ...

Injection

```
namespace a {  
    int value = 42;  
}
```

```
namespace b {  
    consteval -> reflexpr(a::value);  
}
```

Injection

```
namespace a {  
    int value = 42;  
}
```

```
namespace b {  
    constexpr int value = 42;  
}
```


Fragments

```
constexpr auto frag = fragment {};
```

Fragments

```
constexpr auto frag = fragment {};
```

```
constexpr -> frag;
```

Fragments

```
constexpr auto frag = fragment {};
```

```
constexpr -> frag;
```

```
constexpr -> fragment {};
```

Fragments

```
consteval -> fragment {  
    std::string value = "injected variable";  
};
```

Fragments

```
consteval -> fragment {  
    std::string value = "injected variable";  
};
```

Metaprograms

```
consteval {  
    -> fragment {  
        std::string value = "injected variable";  
    };  
}
```

Metaprograms

```
void foo() {  
    consteval {  
        -> fragment {  
            std::string value = "injected variable";  
        };  
    }  
    use(value);  
}
```

Metaprograms

```
consteval void immediateFunction() {  
    -> fragment {  
        std::string value = "injected variable";  
    };  
}
```

```
void foo() {  
    consteval { immediateFunction(); }  
    use(value);  
}
```


Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(info);
    };
}
```

Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(info);
    };
}
```



reference to local variable

Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(%{ info });
    };
}
```

Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(%{ info });
    };
}
```

Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(%{ info });
    };
}
```

Unquote operator

```
template<typename T>
constexpr void injectName() {
    auto info = reflexpr(T);
    -> fragment {
        const std::string name = meta::name_of(%{ info });
    };
}
```

Fragment requirements

```
constexpr auto initialValueFrag = fragment {  
    requires typename T;  
  
    T initialValue{};  
};
```

Fragment requirements

```
constexpr auto initialValueFrag = fragment {  
    requires typename T;
```

```
    T initialValue{};  
};
```

```
struct T { int i = 0; };
```


Fragment requirements

```
constexpr auto initialValueFrag = fragment {  
    requires typename T;
```

```
    T initialValue{};  
};
```

```
struct T { int i = 0; };
```

```
constexpr -> initialValueFrag;
```

```
std::cout << "initial value = " << initialValue.i;
```

outputs:
initial value = 0

Fragment requirements

```
constexpr auto newValueFrag = fragment {  
    requires std::string identifier;  
  
    std::string newValue = identifier + " is cool";  
};
```

Fragment requirements

```
constexpr auto newValueFrag = fragment {  
    requires std::string identifier;  
  
    std::string newValue = identifier + " is cool";  
};  
  
std::string identifier = "metaprogramming";
```

Fragment requirements

```
constexpr auto newValueFrag = fragment {  
    requires std::string identifier;  
  
    std::string newValue = identifier + " is cool";  
};
```

```
std::string identifier = "metaprogramming";
```

```
constexpr -> newValueFrag;  
std::cout << newValue;
```

```
outputs:  
metaprogramming is cool
```

Fragments

- block fragment
- namespace fragment
- class fragment (used for metaclasses)
- enum fragment

Class fragment

```
constexpr void makeNonCopyable() {  
    -> fragment struct T {  
        T(const T&) = delete;  
        T &operator=(const T&) = delete;  
    };  
}
```

Class fragment

```
constexpr void makeNonCopyable() {
```

```
    -> fragment struct T {
```

```
        T(const T&) = delete;
```

```
        T &operator=(const T&) = delete;
```

```
    };
```

```
}
```



the type we are injecting into

Class fragment

```
struct NonCopyable {  
    consteval { makeNonCopyable(); }  
};
```



```
struct NonCopyable {  
    NonCopyable(const NonCopyable &) = delete;  
    NonCopyable &operator=(const NonCopyable &) = delete;  
};  
  
static_assert(!std::is_copy_constructible_v<NonCopyable> && !std::is_copy_assignable_v<NonCopyable>); // ✓  
static_assert(!std::is_move_constructible_v<NonCopyable> && !std::is_move_assignable_v<NonCopyable>); // ✓
```


Class fragment

```
constexpr void makeMovable() {  
    -> fragment struct T {  
        T(T&&) = default;  
        T &operator=(T&&) = default;  
    };  
}
```

Class fragment

```
struct MovableNonCopyable {  
    constexpr {  
        makeNonCopyable();  
        makeMovable();  
    }  
};
```

Class fragment


```
struct MovableNonCopyable {  
    consteval {  
        makeNonCopyable();  
        makeMovable();  
    }  
};
```



```
struct MovableNonCopyable {  
    MovableNonCopyable(const MovableNonCopyable&) = delete;  
    MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;  
    MovableNonCopyable(MovableNonCopyable&&) = default;  
    MovableNonCopyable &operator=(MovableNonCopyable&&) = default;  
};
```

Class fragment

```
struct MovableNonCopyable {  
    consteval {  
        makeNonCopyable();  
        makeMovable();  
    }  
};
```



```
struct MovableNonCopyable {  
    MovableNonCopyable(const MovableNonCopyable&) = delete;  
    MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;  
    MovableNonCopyable(MovableNonCopyable&&) = default;  
    MovableNonCopyable &operator=(MovableNonCopyable&&) = default;  
};
```

Enum fragment

```
consteval void injectBitMask(const char *name,  
                             size_t value) {  
    -> fragment enum {  
        unqualid(%{ name }) = %{ value },  
    };  
}
```

Enum fragment

```
constexpr void injectBitMask(const char *name,
                             size_t value) {
    -> fragment enum {
        unqualid(%{ name }) = %{ value },
    };
}
template<typename... T>
    requires (std::is_same_v<T, const char *> && ...)
constexpr void generateBitMasks(T... names) {
    size_t counter = 0;
    (injectBitMask(names, 1 << counter++), ...);
}
```

Enum fragment

```
constexpr void injectBitMask(const char *name,
                             size_t value) {
    -> fragment enum {
        unqualid(%{ name }) = %{ value },
    };
}

template<typename... T>
    requires (std::is_same_v<T, const char *> && ...)
constexpr void generateBitMasks(T... names) {
    size_t counter = 0;
    (injectBitMask(names, 1 << counter++), ...);
}
```

Enum fragment

```
enum class BitMask {  
    consteval { generateBitMasks("Foo", "Bar", "Baz"); }  
    All = ~0  
};
```



```
enum class BitMask : int {  
    Foo = 1 << 0,  
    Bar = 1 << 1,  
    Baz = 1 << 2,  
    All = ~0  
};
```


Enum fragment

```
enum class BitMask {  
    consteval { generateBitMasks("Foo", "Bar", "Baz"); }  
  
};
```



```
enum class BitMask : int {  
    Foo = 1 << 0,  
    Bar = 1 << 1,  
    Baz = 1 << 2,  
  
};
```

Namespace fragment

```
template<Enumeration Enum>
constexpr void defineBitwiseOperations() {
    -> fragment namespace {
        Enum operator|(Enum a, Enum b) {
            using T = std::underlying_type_t<Enum>;
            return static_cast<Enum>(static_cast<T>(a) | static_cast<T>(b));
        }
        Enum operator&(Enum a, Enum b) {
            using T = std::underlying_type_t<Enum>;
            return static_cast<Enum>(static_cast<T>(a) & static_cast<T>(b));
        }
    };
}
```

Namespace fragment

```
template<Enumeration Enum>
constexpr void defineBitwiseOperations() {
    -> fragment namespace {
        Enum operator|(Enum a, Enum b) {
            using T = std::underlying_type_t<Enum>;
            return static_cast<Enum>(static_cast<T>(a) | static_cast<T>(b));
        }
        Enum operator&(Enum a, Enum b) {
            using T = std::underlying_type_t<Enum>;
            return static_cast<Enum>(static_cast<T>(a) & static_cast<T>(b));
        }
    };
}
```

Namespace fragment

```
enum class BitMask {  
    consteval { generateBitMasks("Foo", "Bar", "Baz"); }  
    All = ~0  
};
```

Namespace fragment

```
enum class BitMask {  
    consteval { generateBitMasks("Foo", "Bar", "Baz"); }  
    All = ~0  
};
```

```
consteval { defineBitwiseOperations<BitMask>(); }
```

Namespace fragment

```
enum class BitMask {  
    constexpr { generateBitMasks("Foo", "Bar", "Baz"); }  
    All = ~0  
};
```

```
constexpr { defineBitwiseOperations<BitMask>(); }
```

```
const auto val = BitMask::Foo | BitMask::Bar;
```

More metaprogramming

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                requires Enum value;
                if (value == valueof(%{ member }))
                    return meta::name_of(%{ member });
            };
    }
    return {};
}
```

More metaprogramming

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                requires Enum value;
                if (value == valueof(%{ member }))
                    return meta::name_of(%{ member });
            };
    }
    return {};
}
```


More metaprogramming

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                requires Enum value;
                if (value == valueof(%{ member }))
                    return meta::name_of(%{ member });
            };
    }
    return {};
}
```

More metaprogramming

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                requires Enum value;
                if (value == valueof(%{ member }))
                    return meta::name_of(%{ member });
            };
    }
    return {};
}
```

More metaprogramming

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                requires Enum value;
                if (value == valueof(%{ member }))
                    return meta::name_of(%{ member });
            };
    }
    return {};
}
```

does not currently work

More metaprogramming — janky workaround

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        auto valueInfo = reflexpr(value);
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                const auto name = meta::name_of(%{ member });
                if (idexpr(%{ valueInfo }) == valueof(%{ member }))
                    return name;
            };
    }
    return {};
}
```

More metaprogramming — janky workaround

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        auto valueInfo = reflexpr(value);
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                const auto name = meta::name_of(%{ member });
                if (idexpr(%{ valueInfo }) == valueof(%{ member }))
                    return name;
            };
    }
    return {};
}
```

More metaprogramming — janky workaround

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        auto valueInfo = reflexpr(value);
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                const auto name = meta::name_of(%{ member });
                if (idexpr(%{ valueInfo }) == valueof(%{ member }))
                    return name;
            };
    }
    return {};
}
```

More metaprogramming — janky workaround

```
template<typename Enum>
std::string_view toString(Enum value) {
    consteval {
        auto valueInfo = reflexpr(value);
        for (auto member : meta::members_of(reflexpr(Enum)))
            -> fragment {
                const auto name = meta::name_of(%{ member });
                if (idexpr(%{ valueInfo }) == valueof(%{ member }))
                    return name;
            };
    }
    return {};
}
```

Assembling it all together — serialization

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m }), meta::name_of(%{ m }));
            };
    }
}
```


Assembling it all together — serialization

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m }), meta::name_of(%{ m }));
            };
    }
}
```

Assembling it all together — serialization

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m }), meta::name_of(%{ m }));
            };
    }
}
```

Assembling it all together — serialization

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m }), meta::name_of(%{ m }));
            };
    }
}
```

Assembling it all together — serialization

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                requires Visitor &&visit;
                requires T &&value;
                visit(value.idexpr(%{ m }), meta::name_of(%{ m }));
            };
    }
}
```

does not currently work

Assembling it all together — janky workaround

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    using Type = std::decay_t<T>;
    consteval {
        auto visitInfo = reflexpr(visit); // workaround
        auto valueInfo = reflexpr(value);
        for (auto m : meta::data_member_range(reflexpr(Type)))
            -> fragment {
                { // workaround
                    auto &visit = idexpr(%{ visitInfo });
                    const auto &value = idexpr(%{ valueInfo });
                    const auto &field = value.idexpr(%{ m });
                    const auto name = meta::name_of(%{ m });
                    visit(field, name);
                }
            };
    }
}
```


<https://cppx.godbolt.org/>

Proposal in the C++ standard P1717: <https://wg21.link/p1717>

https://youtu.be/ARxj3dfF_h0

<https://youtu.be/kjQXhuPX-Ac>

Cppcon | 2019
The C++ Conference
cppcon.org



Andrew Sutton

Why care about static reflection?


- Reducing boilerplate
- Type-based optimization
- New forms of composition
- Runtime introspection

© 2019 Lock3 Software, LLC

Reflections:
Compile-time
Introspection of
Source Code

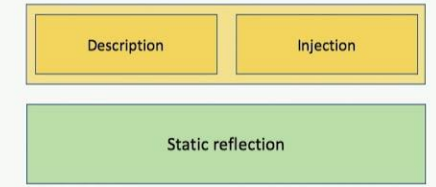
Video Sponsorship Provided By:
ansatz

Cppcon | 2019
The C++ Conference
cppcon.org



Andrew Sutton

Features of generative metaprogramming



Meta++ Language
Support for Advanced
Generative
Metaprogramming

Video Sponsorship Provided By:
ansatz



Metaprogramming is coming... kinda.

Pavel Novikov

 @cpp_ape

Align Technology R&D

align

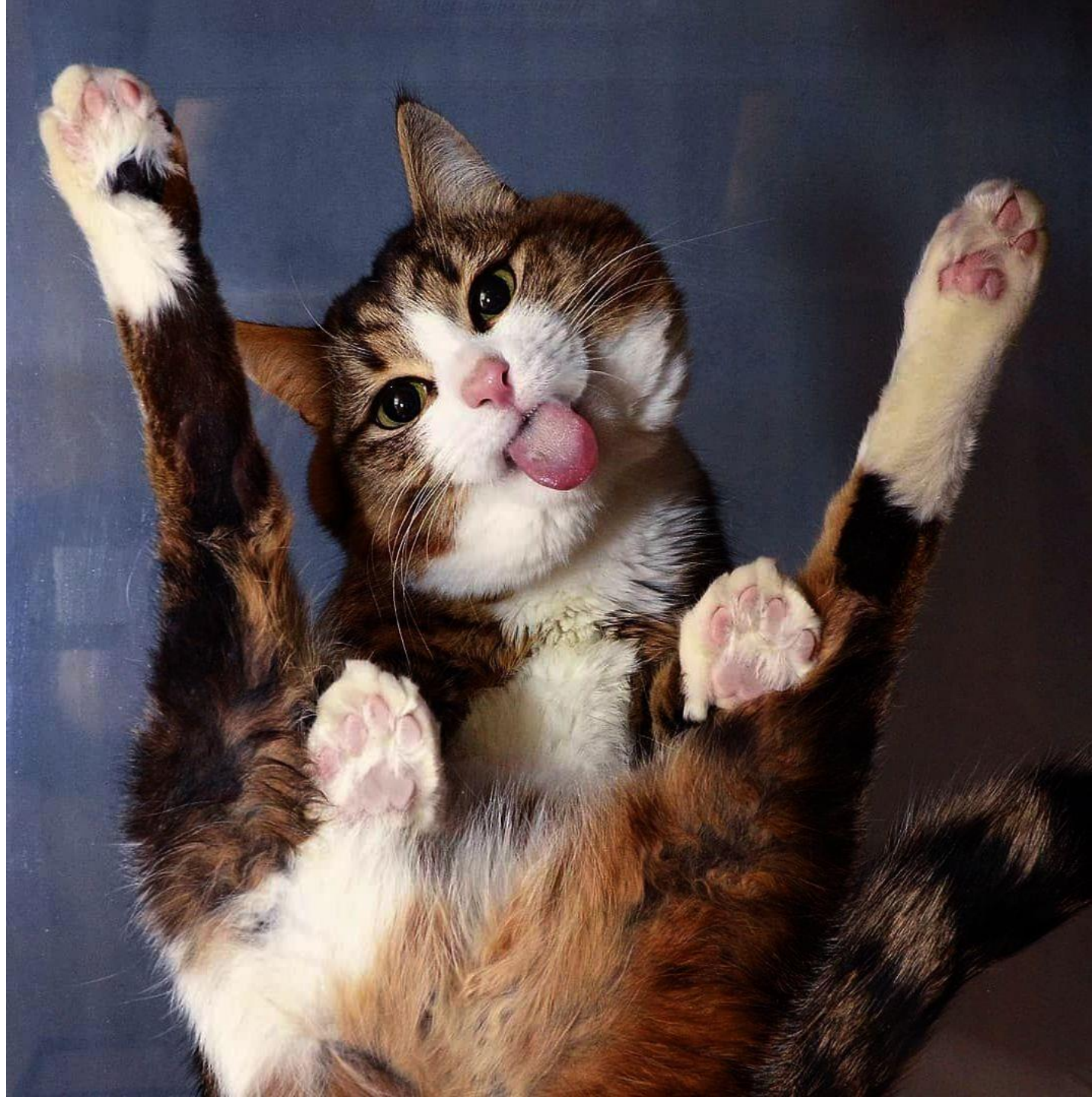
Thanks to Wyatt Childers for feedback!

Slides: <https://git.io/JJ3dJ>



@GameOfThrones

Bonus slides



Metaclasses

```
constexpr void AddTextMember(meta::info type) {  
    for (auto m : meta::member_range(type))  
        ->m;  
  
    -> fragment struct {  
        std::string text;  
    };  
}
```

```
struct(AddTextMember) AddedMember {};
```



```
struct AddedMember {  
    std::string text;  
};
```

Metaclasses

```
constexpr void Interface(meta::info type) {
    bool needDtor = true;
    for (auto m : meta::member_range(type)) {
        if (meta::is_data_member(m))
            meta::compiler.error("an interface can not contain data members");
        if (meta::is_destructor(m)) {
            needDtor = false;
            meta::make_virtual(m);
        }
        else if (meta::is_nonstatic_member_function(m) && !meta::is_defined(m)) {
            meta::make_pure_virtual(m);
        }
        -> m;
    }
    if (needDtor) {
        -> fragment struct T {
            virtual ~T() = default;
        };
    }
}
```

Metaclasses

```
struct(Interface) Cat {  
    void meow();  
    void purr();  
};
```



```
struct Cat {  
    virtual ~Cat() = default;  
    virtual void meow() = 0;  
    virtual void purr() = 0;  
};
```

Metaprogramming as of December 2019

Metaprogramming

- injection
- fragments
- metaprograms

Injection

consteval -> ...

Injection

```
namespace a {  
    int value = 42;  
}
```

```
namespace b {  
    consteval -> reflexpr(a::value);  
}
```

Injection

```
namespace a {  
    int value = 42;  
}
```

```
namespace b {  
    constexpr int value = 42;  
}
```


Fragments

```
const auto fragment = __fragment {};
```

Fragments

```
const auto fragment = __fragment {};
```

```
consteval -> fragment;
```

Fragments

```
const auto fragment = __fragment {};
```

```
consteval -> fragment;
```

```
consteval -> __fragment {};
```

Fragments

```
consteval -> __fragment {  
    std::string value = "injected variable";  
};
```

Fragments

```
consteval -> __fragment {  
    std::string value = "injected variable";  
};
```

Fragments

- namespace fragment
- class fragment (used for metaclasses)
- enum fragment

Metaprograms

```
consteval {  
    -> __fragment {  
        std::string value = "injected variable";  
    };  
}
```

Metaprograms

```
void foo() {  
    consteval {  
        -> __fragment {  
            std::string value = "injected variable";  
        };  
    }  
    use(value);  
}
```


Metaprograms

```
consteval void immediateFunction() {  
    -> __fragment {  
        std::string value = "injected variable";  
    };  
}
```

```
void foo() {  
    consteval { immediateFunction(); }  
    use(value);  
}
```

More metaprogramming

```
template<typename Enum>
std::string toString(Enum value) {
    consteval {
        for (auto i : meta::members_of(reflexpr(Enum))
            -> __fragment {
            if (value == valueof(i))
                return meta::name_of(i);
        };
    }
    return "<unknown>";
}
```

More metaprogramming

```
template<typename Enum>
std::string toString(Enum value) {
    consteval {
        for (auto i : meta::members_of(reflexpr(Enum))
            -> __fragment {
            if (value == valueof(i))
                return meta::name_of(i);
        };
    }
    return "<unknown>";
}
```

Assembling it all together

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    consteval {
        auto type = reflexpr(std::decay_t<T>);
        for (auto m : meta::data_member_range(type)) {
            -> __fragment {
                visit(value.unqualid(meta::name_of(m)), meta::name_of(m));
            };
        }
    }
}
```

Assembling it all together

```
template<typename Visitor, typename T>
void iterateMembers(Visitor &&visit, T &&value) {
    consteval {
        auto type = reflexpr(std::decay_t<T>);
        for (auto m : meta::data_member_range(type)) {
            -> __fragment { { // workaround
                auto ptr = valueof(m);
                visit(value.*ptr, meta::name_of(m));
            } };
        }
    }
}
```