

+ 21

Building an Extensible Type Serialization System Using Partial Template Specialization

PEJMAN GHORBANZADE



20
21



Building an Extensible Type Serialization System using Partial Template Specialization



Pejman Ghorbanzade
pejman@touca.io

Or...

A Practical Review of Approaches to Designing Extension Points in C++



Pejman Ghorbanzade
pejman@touca.io

Intro

This talk will not include...

- An exhaustive tutorial about building serialization libraries
- An exhaustive tutorial about partial template specialization
- A deep-dive into generic programming
- An endorsement of any one method for creating extension points

Intro

This talk attempts to...

- Review language features for building extensible libraries
- Showcase a real-world library with an extensible type system
- Make you excited about the new and upcoming language features

Agenda

- **Motivation**
- The Basics
- Practical Example
- Concepts
- Argument Dependent Lookup
- Static Reflection

Motivation

About Me

Professional Software Engineer

Canon Medical Informatics

VMware Carbon Black

Working full-time on touca.io

Continuous Regression Testing

Passionate about maintaining software at scale



3D visualization of Lung CT
Courtesy of Canon Medical Information

Motivation

The Problem

How can we refactor half a million lines of code without causing any side effects?

Motivation

Candidate Solution A

```
auto new_output = new_system(testcase);  
auto old_output = old_system(testcase);  
compare(new_output, old_output);
```

Disadvantages

- Test is difficult to setup
- Test system is inefficient to run
- Test system is not reusable

Motivation

Candidate Solution B

```
auto new_output = new_system(testcase);  
auto new_file = write_to_file(testcase, new_output);  
auto old_file = find_old_file(testcase);  
compare(new_file, new_output);
```

Disadvantages

- Dealing with files is no fun
- Test system is hard to maintain
- Test system is not reusable

Motivation

Candidate Solution C

```
auto new_output = new_system(testcase);  
auto new_description = describe(new_output);  
submit(testcase, new_description);
```

Disadvantages

- Limited customization
- Overkill for small projects
- Requires remote computing resources

Motivation

Simple Example

```
struct Student {  
    std::string username;  
    std::string fullname;  
    Date birth_date;  
    std::vector<Course> courses;  
};  
  
Student find_student(const std::string& username);
```

Motivation

High-level API

```
#include "students.hpp"
#include "touca/touca.hpp"

int main(int argc, char* argv[]) {
    touca::workflow("students", [](const std::string& username) {
        const auto& student = find_student(username);
        touca::check("username", student.username);
        touca::check("fullname", student.fullname);
        touca::check("birth_date", student.birth_date);
        touca::check("courses", student.courses);
    });
    touca::run(argc, argv);
}
```

<https://github.com/trytouca/trytouca>

Design Requirements

- Intuitive developer experience
- Intrinsic support for common types
 - Must support integral types, floating point types, string-like types, containers, and other common standard types
- Extensible design to support user-defined types
 - Must allow users to introduce logic for handling custom types

Agenda

- Motivation
- **The Basics**
- Practical Example
- Concepts
- Argument Dependent Lookup
- Static Reflection

Function Overloading

```
void check(const std::string& key, const boolean_t value);  
void check(const std::string& key, const number_unsigned_t value);  
void check(const std::string& key, const array_t& value);  
void check(const std::string& key, const object_t& value);  
void check(const std::string& key, const string_t& value);  
/** and so it goes... */
```

✗ Extensible design to support user-defined types

Callback Functions

```
check("some-date", [&date]() {  
  return object()  
    .add("year", date.year)  
    .add("month", date.month)  
    .add("day", date.day);  
});
```

✗ Intrinsic support for common types

Polymorphism

```
struct Date : public Serializable {  
    /* ... */  
  
    generic_value serialize() const override;  
};
```

✗ Intuitive Developer Experience

com.google.gson

```
private class MyDateSerializer implements
    JsonSerializer<MyDate> {
    public JsonElement serialize(MyDate src, Type typeOfSrc,
        JsonSerializerContext context) {
        JsonObject obj = new JsonObject();
        obj.addProperty("year", src.getYear());
        obj.addProperty("month", src.getMonth());
        obj.addProperty("day", src.getDay());
        return obj;
    }
}
```

The Basics

com.google.gson

Type adapters are introduced **at runtime** and considered during serialization of any given value.

```
final Gson gson = new GsonBuilder()  
    .registerTypeAdapter(MyDate.class, new MyDateSerializer())  
    .create();
```

Runtime resolution is slow and inefficient. We can do much better in C++.

Simple Example

```
struct Date {  
    unsigned short year;  
    unsigned short month;  
    unsigned short day;  
  
    std::string to_string() const;  
  
    /** and so it goes */  
};
```

std::ostream

```
struct Date {  
    /* ... */  
  
    friend std::ostream& operator<<(std::ostream& os, const Date& dt);  
};
```

```
std::ostream &operator <<(std::ostream &o, const Date &date) {  
    return o << date.to_string();  
}
```

QDataStream

```
QFile file("file.dat");  
file.open(QIODevice::WriteOnly);  
QDataStream out(&file);  
out << QString("the answer is ");  
out << (qint32) 42;
```

```
QDataStream& operator<<(QDataStream&, const Data&);  
QDataStream& operator>>(QDataStream&, Data&);
```

boost::serialization

```
namespace boost {
namespace serialization {

template<class Archive>
void serialize(Archive& archive, Date& date, const unsigned int version)
{
    archive & date.year;
    archive & date.month;
    archive & date.day;
}

} // namespace serialization
} // namespace boost
```


boost::serialization

```
struct Date {  
    /* ... */  
  
private:  
    friend class boost::serialization::access;  
  
    template<class Archive>  
    void serialize(Archive & ar, const unsigned int version) {  
        ar & year;  
        ar & month;  
        ar & day;  
    }  
};
```

std::format

```
template <>
struct std::formatter<Date> : std::formatter<std::string> {
    auto format(const Date& p, auto& ctx) {
        return formatter<std::string>::format(
            std::format("{} / {} / {}", p.month, p.day, p.year), ctx);
    }
};
```

(since C++20)

Function Template Specialization

```
template <typename T>
void print(T arg) {
    std::cout << arg << std::endl;
}
```

```
template <>
void print(const Date& date) {
    std::cout << date.to_string() << std::endl;
}
```

Function Template Specialization

```
void print(auto arg) { std::cout << arg << std::endl; }
```

```
void print(const Date& date) {  
    std::cout << date.to_string() << std::endl;  
}
```

Class Template Specialization

```
template <typename T>
struct printer {
    void print(T arg) { std::cout << arg << std::endl; }
};
```

```
template <>
struct printer<Date> {
    void print(const Date& arg) {
        std::cout << arg.to_string() << std::endl;
    }
};
```

std::hash

```
template <>
struct std::hash<Date> {
    std::size_t operator()(const Date& date) const noexcept {
        return std::hash<std::string>{}(date.to_string());
    }
};
```

Partial Template Specialization

```
template <typename T, typename U>
struct printer {
    void print(T prefix, U value) {
        std::cout << prefix << value << std::endl;
    }
};

template <typename T>
struct printer<T, Date> {
    void print(T prefix, Date value) {
        std::cout << prefix << value.to_string() << std::endl;
    }
};
```

`std::enable_if`

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```


Substitution Failure is not an Error

```
template <typename T, typename Enabled = void>
struct printer {
    void print(T value) { std::cout << value << std::endl; }
};

template <typename T>
struct printer<T,
    typename std::enable_if<std::is_same<T, Date>::value>::type> {
    void print(T value) { std::cout << value.to_string() << std::endl; }
};
```

Helper Types

```
template <bool B, class T = void>  
using enable_if_t = typename enable_if<B,T>::type;
```

```
template <class T, class U>  
constexpr bool is_same_v = is_same<T, U>::value;
```

Leveraging Helper Types

```
template <typename T, typename = void>
struct printer {
    void print(T value) { std::cout << value << std::endl; }
};

template <typename T>
struct printer<T, std::enable_if_t<std::is_same_v<T, Date>>> {
    void print(T value) { std::cout << value.to_string() << std::endl; }
};
```

(since C++17)

Agenda

- Motivation
- The Basics
- **Practical Example**
- Concepts
- Argument Dependent Lookup
- Static Reflection

User-facing API

```
check("some-boolean", true);
check("some-number", 42.0f);
check("some-string", "forty two");
check("some-array", std::vector<int>{42});
check("some-date", Date{.year=2021, .month=10, .day=29});
check("some-map",
      std::map<std::string, Date>{{"today", Date(2021, 10, 29)},
                                {"yesterday", Date(2020, 10, 28)}});
```

Perfect Forwarding

```
template <typename Char, typename Value>
void check(Char&& key, const Value& value) {
    detail::check(std::forward<Char>(key),
                  serializer<Value>().serialize(value));
}
```

Specializing User-defined Types

```
template <>
struct serializer<Date> {
    generic_value serialize(const Date& value) {
        return object()
            .add("year", value.year)
            .add("month", value.month)
            .add("day", value.day);
    }
};
```

Primary Template

```
template <typename T, typename = void>
struct serializer {
    generic_value serialize(const T& value) {
        static_assert(std::is_same<generic_value, T>::value,
            "did not find any specialization of serializer "
            "for the given type");
        return static_cast<T>(value);
    }
};
```


Basic Types

- Dependent on product requirements
- Other Types
 - Binary data
 - Short string
 - Large text
 - Large number sequences
- Properties
 - Ordered/Unordered
 - File Paths

```
enum class internal_type : std::uint8_t {  
    null,  
    object,  
    array,  
    string,  
    boolean,  
    number_signed,  
    number_unsigned,  
    number_float,  
    number_double,  
    unknown  
};
```

Data Storage

```
union internal_value {  
    object_t* object;  
    array_t* array;  
    string_t* string;  
    boolean_t boolean;  
    number_signed_t number_signed;  
    number_unsigned_t number_unsigned;  
    number_float_t number_float;  
    number_double_t number_double;  
}
```

```
using object_t =  
    std::map<std::string, generic_value>;  
using array_t =  
    std::vector<generic_value>;  
using string_t = std::string;  
using boolean_t = bool;  
using number_signed_t = int64_t;  
using number_unsigned_t = uint64_t;  
using number_float_t = float;  
using number_double_t = double;
```

Type Wrapper

```
class generic_value {
public:
    generic_value(const internal_type type) : _type(type) {}

    static generic_value boolean(const boolean_t value)
        : _type(internal_type::boolean), _value(value) {}

    /** and so it goes */

private:
    internal_type _type = internal_type::null;
    internal_value _value;
};
```

Specializing for Boolean Types

```
template <typename T>
struct serializer<T, std::enable_if_t<std::is_same_v<T, bool>>> {
    generic_value serialize(const T& value) { return value; }
};
```

Specializing for Boolean Types

```
template <typename T>
constexpr bool is_boolean_v = std::is_same_v<T, bool>;
```

```
template <typename T>
struct serializer<T, std::enable_if_t<is_boolean_v<T>>> {
    generic_value serialize(const T& value) { return value; }
};
```

Specializing for Numeric Types

```
template <typename T>
struct serializer<T, std::enable_if_t<is_number_signed_v<T>>> {
    generic_value serialize(const T& value) {
        return static_cast<std::int64_t>(value);
    }
};
```

Specializing for Numeric Types

```
template <typename T, typename = void>
struct is_number_signed : std::false_type {};

template <typename T>
struct is_number_signed<
    T, std::enable_if_t<!std::is_same_v<T, bool> &&
                      std::is_integral_v<T> &&
                      std::is_signed_v<T>>>
    : std::true_type {};
```

✗ We can do better

Specializing for Numeric Types

```
template <typename T>
constexpr bool is_number_signed_v =
    std::conjunction_v<std::negation<std::is_same<T, bool>>,
                      std::is_integral<T>,
                      std::is_signed<T>>;
```


Specializing for String-like Types

```
template <typename T>
using is_string =
    std::disjunction<std::is_constructible<std::string, T>,
                    std::is_constructible<std::wstring, T>>;
```

Specializing for Containers - Attempt 1

```
template <typename T, typename = void>
struct is_array : std::false_type {};

template <typename T, std::size_t N>
struct is_array<std::array<T, N>> : std::true_type {};

template <typename... args>
struct is_array<std::set<args...>> : std::true_type {};

template <typename... args>
struct is_array<std::vector<args...>> : std::true_type {};

/** and so it goes */
```

Helper Trait: `is_specialization`

```
template <typename Test, template <typename...> class Ref>
struct is_specialization : std::false_type {};

template <template <typename...> class Ref, typename... Args>
struct is_specialization<Ref<Args...>, Ref> : std::true_type {};
```

Specializing for Containers - Attempt 2

```
template <typename T>
struct is_array<T, enable_if_t<disjunction<
    is_specialization<T, std::deque>,
    is_specialization<T, std::list>,
    is_specialization<T, std::map>,
    is_specialization<T, std::set>,
    is_specialization<T, std::unordered_map>,
    is_specialization<T, std::vector>>>::value>> : std::true_type {};
```

Helper Trait: `is_iterable`

```
template <typename T, typename = void>
struct is_iterable : std::false_type {};

template <typename T>
struct is_iterable<T, void_t<decltype(std::begin(std::declval<T>())),
                          decltype(std::end(std::declval<T>()))>>
    : std::true_type {};
```

Practical Example

Specializing for Containers - Attempt 3

```
template <typename T>  
using is_array =  
    std::conjunction<std::negation<is_string<T>>, is_iterable<T>>;
```

Specializing for Containers - Attempt 4

```
template <typename T>
struct serializer<T, std::enable_if_t<is_array<T>::value>> {
    generic_value serialize(const T& value) {
        generic_value out(internal_type::array);
        for (const auto& v : value) {
            out.add(serializer<typename T::value_type>().serialize(v));
        }
        return out;
    }
};
```

Specializing for other Standard Types

```
template <typename T>
struct serializer<T,
    std::enable_if_t<is_specialization<T, std::pair>::value>> {
    generic_value serialize(const T& value) {
        return detail::array()
            .add(serializer<typename T::first_type>().serialize(value.first))
            .add(serializer<typename T::second_type>().serialize(value.second));
    }
};
```


Specializing for other Standard Types

- Pointer Types
- Enum Types
- `std::variant`
- `std::tuple`
- `std::optional`
- And so it goes...

What are we doing with our lives?

This is clearly not elegant.

Agenda

- Motivation
- The Basics
- Practical Example
- **Concepts**
- Argument Dependent Lookup
- Static Reflection

The Basics

- Concepts are named predicates evaluated at compile-time.
 - Constrain template parameters
- Reaching for the aims of C++
 - Improved readability
 - Reduced complexity
 - Better diagnostics
 - Faster compilation time

The Basics

```
template <typename T>  
requires CONDITION  
void foo(T t) {}
```

```
template <typename T>  
void foo(T t) requires CONDITION {}
```

```
template <CONDITION T>  
void foo(T t) {}
```

```
void foo(CONDITION auto t) {}
```

The Basics

```
template <typename T>
concept HasToString = requires(const T& value) {
    value.to_string();
};

template <typename T>
void printer(const T& value) {
    std::cout << value << std::endl;
}

template <HasToString T>
void printer(const T& value) {
    std::cout << value.to_string() << std::endl;
}
```

The Basics

```
template <typename T>
concept HasToString = requires(const T& value) {
    value.to_string()
};

void printer(const auto& value) {
    std::cout << value << std::endl;
}

void printer(const HasToString auto& value) {
    std::cout << value.to_string() << std::endl;
}
```

Reconsidering our Approach

```
template <typename Char, typename Value>
void check(Char&& key, const Value& value) {
    detail::check(std::forward<Char>(key), serialize(value));
}
```

Specializing Basic Types

```
generic_value serialize(std::nullptr_t value) {  
    /** and so it goes */  
}
```

```
generic_value serialize(const bool value) {  
    /** and so it goes */  
}
```


Specializing Numeric Types

```
template <typename T>  
concept Arithmetic = std::integral<T> || std::floating_point<T>;
```

```
generic_value serialize(const Arithmetic auto& value) {  
    /** and so it goes */  
}
```

Specializing String Types

```
template <typename T>
concept StringLike =
    std::convertible_to<T, std::basic_string<typename T::value_type>>;
```

```
generic_value serialize(const StringLike auto& value) {
    /** and so it goes */
}
```

Specializing Fixed-Sized Arrays

```
template <typename Char, std::size_t N>  
generic_value serialize(const Char (&value)[N]) {  
    /** and so it goes */  
}
```

Helper Trait: `is_iterable`

```
template <typename T, typename = void>
struct is_iterable : std::false_type {};

template <typename T>
struct is_iterable<T, void_t<decltype(std::declval<T>().begin()),
                        decltype(std::declval<T>().end())>>
    : std::true_type {};
```

✗ We can do better

Helper Concept: Iterable

```
template <typename T>
concept Iterable = requires(const T x) {
    { x.begin() } -> std::same_as<typename T::const_iterator>;
    { x.end() } -> std::same_as<typename T::const_iterator>;
};
```

✗ We can do better

Helper Concept: Container

```
template <typename T>
concept Container = requires(T a, const T b) {
    { a.begin() } -> std::same_as<typename T::iterator>;
    { a.end() } -> std::same_as<typename T::iterator>;
    { b.begin() } -> std::same_as<typename T::const_iterator>;
    { b.end() } -> std::same_as<typename T::const_iterator>;
    { a.cbegin() } -> std::same_as<typename T::const_iterator>;
    { a.cend() } -> std::same_as<typename T::const_iterator>;
    { a.size() } -> std::same_as<typename T::size_type>;
    { a.max_size() } -> std::same_as<typename T::size_type>;
    { a.empty() } -> std::same_as<bool>;

    /** part 1/3 */
};
```

Helper Concept: Container

```
template <typename T>
concept Container = requires(T a, const T b) {
    /** ... */

    requires std::regular<T>;
    requires std::swappable<T>;
    requires std::destructible<typename T::value_type>;
    requires std::same_as<typename T::reference, typename T::value_type&>;
    requires std::same_as<typename T::const_reference, const typename T::value_type&>;
    requires std::forward_iterator<typename T::iterator>;
    requires std::forward_iterator<typename T::const_iterator>;

    /** part 2/3 */
};
```

Helper Concept: Container

```
template <typename T>
concept Container = requires(T a, const T b) {
    /** ... */

    requires std::signed_integral<typename T::difference_type>;
    requires std::same_as<
        typename T::difference_type,
        typename std::iterator_traits<typename T::iterator>::difference_type>;
    requires std::same_as<
        typename T::difference_type,
        typename std::iterator_traits<typename T::const_iterator>::difference_type>;

    /** part 3/3 */
};
```


Specializing for Containers

```
template <typename T>
concept ArrayLike = !StringLike<T> && Container<T>;
```

```
generic_value serialize(const ArrayLike auto& values) {
    auto& out = generic_value::array();
    for (const auto& v : values) {
        out.add(serialize(v));
    }
    return out;
}
```

Taking a Step Back

```
struct Date {  
    unsigned short year;  
    unsigned short month;  
    unsigned short day;  
  
    /** and so it goes */  
  
    generic_value serialize() const;  
};
```

Specializing for User-defined Types

```
template <typename T>
concept Serializable = requires(T x) {
    { x.serialize() } -> std::same_as<generic_value>;
};
```

```
generic_value serialize(const Serializable auto& value) {
    return value.serialize();
}
```

Handling unsupported types

```
template <typename T>
generic_value serialize(const T& value) {
    static_assert(std::is_same_v<generic_value, T>,
                  "did not find any serializer for the given type");
    return static_cast<T>(value);
}
```

Agenda

- Motivation
- The Basics
- Practical Example
- Concepts
- **Argument Dependent Lookup**
- Static Reflection

The Basics

Argument-Dependent Lookup (ADL) enables the lookup of an unqualified function name, in a function call expression, in the namespaces of its arguments.

*Some restrictions apply

```
endl(std::cout);
```

absl::Hash

```
struct Date {
    unsigned short year;
    unsigned short month;
    unsigned short day;

    /** and so it goes */

    friend bool operator==(const Date& lhs, const Date& rhs);

    template <typename H>
    friend H AbslHashValue(H h, const Date& m);
};
```

absl::Hash

```
bool operator==(const Date& lhs, const Date& rhs) {  
    return lhs.year == rhs.year &&  
        lhs.month == rhs.month &&  
        lhs.day == rhs.day;  
}  
  
template <typename Hash>  
H AbslHashValue(Hash h, const Date& date) {  
    return H::combine(std::move(h),  
        date.year, date.month, date.day);  
}
```


nlohmann::json

- Modern JSON library with intuitive API
 - Extensible
 - Customizable

```
json j = {  
    {"year", date.year},  
    {"month", date.month},  
    {"day", date.day}  
};
```

```
{  
    "year": 2021,  
    "month": 10,  
    "day": 29  
}
```

nlohmann::json

```
template <typename T>
struct adl_serializer {
    static void to_json(json& j, const T& value) {
        // calls the "to_json" method in T's namespace
    }

    static void from_json(const json& j, T& value) {
        // calls the "from_json" method in T's namespace
    }
};
```

nlohmann::json

```
using namespace nlohmann;

void to_json(json& j, const Date& date) {
    j = json{{"year", date.year}, {"month", date.month}, {"day", date.day}};
}
```

```
NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(Date, year, month, day)
```

Specializing for User-defined Types

```
struct Date {  
    unsigned short year;  
    unsigned short month;  
    unsigned short day;  
  
    friend void serialize(generic_value& context, const Date& date);  
};
```

Specializing for User-defined Types

```
void serialize(generic_value& context, const Date& date) {  
    return context  
        .add("year", date.year)  
        .add("month", date.month)  
        .add("day", date.day);  
});
```

Agenda

- Motivation
- The Basics
- Practical Example
- Concepts
- Argument Dependent Lookup
- **Static Reflection**

User-facing API

```
check("some-boolean", true);
check("some-number", 42.0f);
check("some-string", "forty two");
check("some-array", std::vector<int>{42});
check("some-date", Date{.year=2021, .month=10, .day=29});
check("some-map",
      std::map<std::string, MyDate>{{"today", MyDate(2021, 10, 29)},
                                   {"yesterday", MyDate(2020, 10, 28)}});
```

Static Reflection

Current Status

Proposals

- Reflection TS Draft (N4856)
- Alternative Draft (P1240, P2237, P2320)

Circle Compiler (with Different Syntax)

Static Reflection

Relevant Talks

Andrew Sutton, ACCU 2021

"Reflection: Compile-Time Introspection of C++"

Pavel Novikov, C++ on Sea 2020

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

David Sankel, C++Now 2019

"The C++ Reflection TS"

TS Draft (N4856)

```
template <typename T> std::string get_type_name() {  
    namespace reflect = std::experimental::reflect;  
    using meta_t = reflexpr(T);  
    using aliased_meta_t = reflect::get_aliased_t<meta_t>;  
    return reflect::get_name_v<aliased_meta_t>;  
}
```

```
get_type_name<std::string>() // -> "basic_string"  
get_type_name<int>() // -> "int"
```

TS Draft (N4856)

```
namespace reflect = std::experimental::reflect;
using meta_t = refl_expr(Date);
using members_t = reflect::get_accessible_data_members_t<meta_t>;
using member_t = reflect::get_element_t<members_t, 0>;
std::cout << reflect::get_name_v<member_t>
           << reflect::get_name_v<reflect::get_type_t<member_t>>
           << std::endl;
```

Constexpr Reflexpr (P0953)

```
template <typename T>
void to_json_impl(const T& object) {
    std::cout << '{';
    constexpr reflect::Class meta = reflexpr(T);
    constexpr auto members = meta.get_accessible_data_members();
    std::size_t count = 0;
    constexpr for(const RecordMember member : members) {
        std::cout << '"' << member.get_name() << '"' << ':';
        constexpr reflect::Constant member_ptr = member.get_pointer();
        to_json(object.*unreflexpr(member_ptr));
        if (++count != members.size()) {
            std::cout << ',';
        }
    }
    std::cout << '}';
}
```

Value-based Reflection (P2320)

```
Date date{.year=2021, .month=10, .day=29};
template for (constexpr meta::info member : meta::members_of(^Date)) {
    std::cout << '"' << meta::name_of(member) << '"'
                << ':' << date.[member:] << std::endl;
}
```

Value-based Reflection (P2320)

```
template <typename T>
void to_json(const T& object) {
    std::cout << '{';
    constexpr auto members = meta::data_members_of(^T);
    std::size_t count = 0;
    template for (constexpr meta::info member : members) {
        std::cout << '"' << member.get_name() << '"' << ':';
        to_json(object.[member]);
        if (++count != size(members)) {
            std::cout << ',';
        }
    }
    std::cout << '}';
}
```

Conclusion

- Template meta programming will continue to have its place and use cases.
- Designing good software requires deep understanding of use cases.
- Designing user-friendly extension points requires leveraging multiple language features.
- C++ is evolving into a simpler, more readable, more maintainable language.

Questions

<https://github.com/ghorbanzade/cppcon21>
