Meeting C++ 2023

Let’s make a library that uses Reflection

Konstantinos Kanavouras

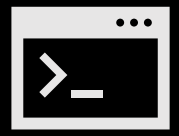
Monday 2023-11-13

UNIVERSITY OF LUXEMBOURG

Agenda

Exploring the future of reflection for the C++ end user

1  Overview of reflection

2  Overview of (de)serialization

3  A reflective serialization library

4  Conclusions

Learn more about reflection

This is not a reflection talk!

Reflection in C++ - Past, Present and Hopeful Future

Andrei Alexandrescu, CppCon 2022

Reflection: Compile-Time Introspection of C++

Andrew Sutton, ACCU 2021

The C++ Reflection TS

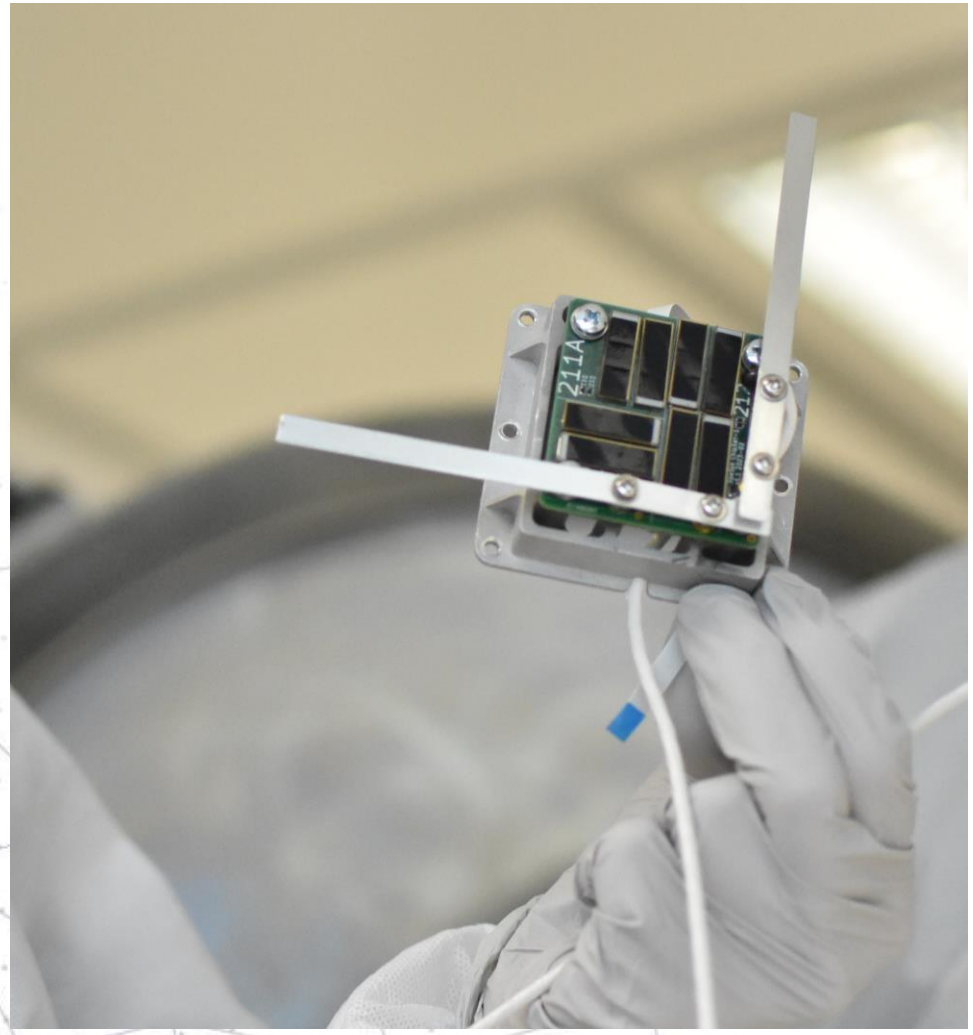
David Sankel, C++Now 2019

Design of a C++ reflection API

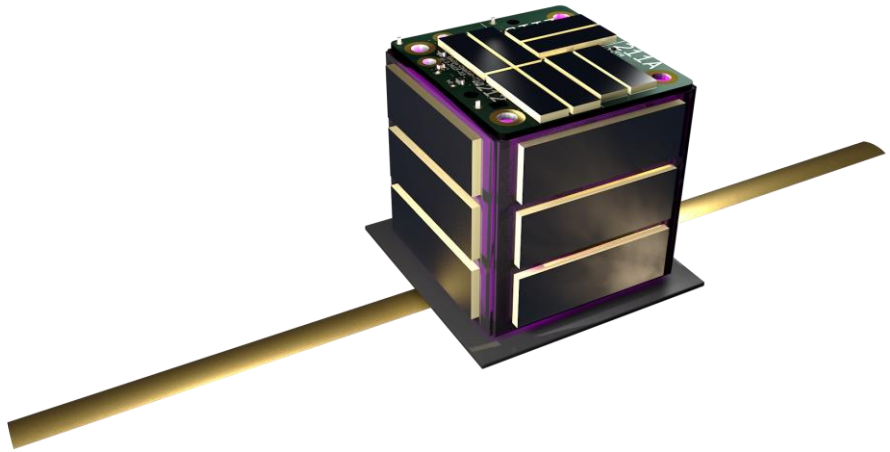
Matúš Chochlík, MeetingC++ Online 2022

About Me

Konstantinos Kanavouras



- Electrical Engineer
- PhD student in the University of Luxembourg
- Working on centimeter-size satellites



What is Reflection?

From P0385R2 (Matúš Chochlík, Axel Naumann, David Sankel):

Reflection is the process of obtaining *metadata*.

Metadata is [...] data conceptually describing some other, primary data.

Example:

```
  metadata
class Toast {
    int id = 7;
    float temperature = 32;  primary data
    std::string review = "very tasty";
public:
    void bake();
}
```

- name of type
- name of variable
- type of variable
- scope
- list of arguments
- list of template arguments
- list of base classes
- list of data members
- list of nested types
- access specifiers
- source location information
- list of member functions
- comments/attributes

What is Reflection?

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = "very tasty";
public:
    void bake();
}
```

Very simple reflection:

```
get_member_variables<Toast>()
-> std::array { "id", "temperature", "review" }
```

```
get_member_functions<Toast>()
-> std::array { std::member_function{"Toast::bake"} }
```

What is Reflection?

```
class Toast {
    void caller() {
        function();
    }

    void callee() {
        std::print("I am {}\n", std::who_am_i());
        std::print("I'm in {}\n", std::where_am_i());
        std::print("Called by {}\n", std::who_called_me());
    }
}

toast->caller();
// I am callee()
// I'm in Toast
// Called by Toast::caller()
```

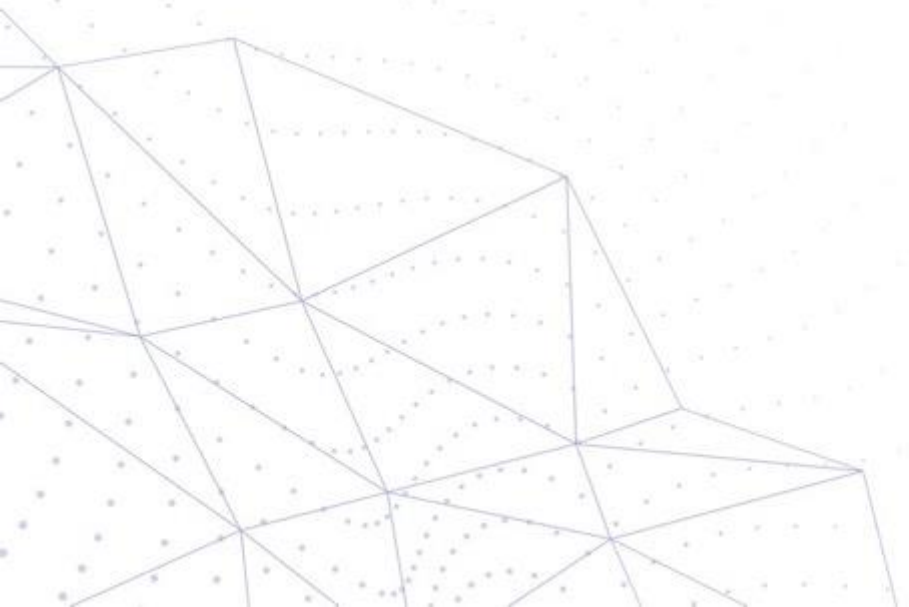
Why do we need reflection?

> *We must look at increasing leverage: more (correct) behaviours from fewer lines of code*
>
> - Andrei Alexandrescu

- Less repetition on complex projects
- More powerful, generic libraries
- Catch errors at compile time
- Reimplement much-needed features without compiler hacks
- Even more flexible concepts/constraints
- …

Your compiler can already do reflection!

```cpp
int main() {
    Toast breakfast;

    std::cout << typeid(breakfast).name() << std::endl;
}

// Output:
// 5Toast
```



Your compiler can already do reflection!

```cpp
int main() {
    Toast breakfast;

    static_assert(typeid(breakfast).name() == "5Toast"sv);
}
```

Not usable at compile-time :(

and requires RTTI?

```
<source>:18:19: error: static assertion expression is not an integral constant expression
18 | static_assert(typeid(breakfast).name() == "5Toast"sv);
   |               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:18:37: note: non-constexpr function 'name' cannot be used in a constant expression
18 | static_assert(typeid(breakfast).name() == "5Toast"sv);
   |                                     ^
```

Your compiler can already do reflection!

Static reflection with standard C++

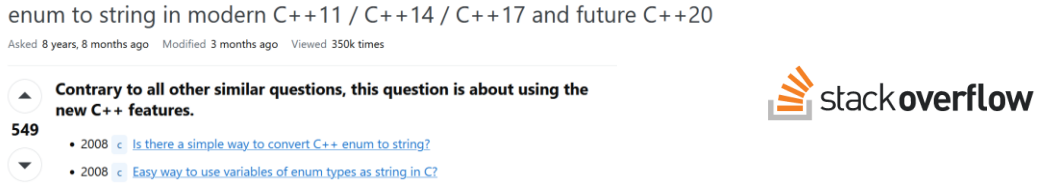
```
#include <type_traits> // C++11

namespace std {
  template<class T> inline constexpr bool is_void_v = ...
  template<class T> inline constexpr bool is_integral_v = ...
  template<class T> inline constexpr bool is_floating_point_v = ...
  template<class T> inline constexpr bool is_array_v = ...
  template<class T> inline constexpr bool is_pointer_v = ...
  template<class T> inline constexpr bool is_enum_v = ...
  template<class T> inline constexpr bool is_union_v = ...
  template<class T> inline constexpr bool is_class_v = ...
  template<class T> inline constexpr bool is_function_v = ...

  template<class T, class U> inline constexpr bool i_same_v =
  template<class B, class D> inline constexpr bool is_base_of_v =
  template<class F, class T> inline constexpr bool is_convertible_v =
  template<class Fn, class... A> inline constexpr bool is_invokable_v =
  // ...
}
```

Constraints and concepts (since C++20)

```
template<class T> concept Bakeable = requires(T t) {
    { t.bake() };
};
```

C++ Reflection Proposals

A few recent publications



P1240R1 Scalable Reflection in C++ 2019

Wyatt Childers
Andrew Sutton
Faisal Vali
Daveed Vandevoorde

```
template <Enum T>
std::string to_string(T value) {
    for constexpr (auto e : std::meta::members_of(reflexpr(T)) {
        if (exprid(e) == value) {
            return std::meta::name_of(e);
        }
    }
    return "<unnamed>";
}
```

N4856 Technical Specification: C++ Extensions for Reflection

David Sankel

2020

ISO/IEC TS 23619:2021 C++ extensions for reflection

For a more complete list of papers: Meeting C++ Blog, Andrew Sutton's presentation

C++ Reflection Proposals

A few recent publications

P1240R2 Scalable Reflection in C++ 2022

Wyatt Childers
Andrew Sutton
Faisal Vali
Daveed Vandevoorde

```
template <Enum T>
std::string to_string(T value) {
    template for (constexpr auto e : std::meta::members_of(^T)) {
        if ([:e:] == value) {
            return std::string(std::meta::name_of(e));
        }
    }
    return "<unnamed>";
}
```

P2996R0 Reflection for C++26 2023

Wyatt Childers, Peter Dimov, Barry Revzin, Andrew Sutton, Faisal Vali, Daveed Vandevoorde

For a more complete list of papers: Meeting C++ Blog, Andrew Sutton's presentation

C++ Reflection TS

Also available in cppreference.com!

Extensions for reflection

The C++ Extensions for Reflection, ISO/IEC TS 23619:2021, specifies modifications to the core language and defines new components for the C++ standard library listed on this page.

The Reflection TS is based on the C++20 standard (except that the definition of concepts are specified in the style of Concepts TS).

Core language changes

reflexpr-specifier

A *reflexpr-specifier* is of form **reflexpr (** *reflexpr-operand* **)**, and specifies a meta-object type (see below).

reflexpr-operand can be one of following:

::	(1)
type-id	(2)
nested-name-specifier(optional) *namespace-name*	(3)
id-expression	(4)
(*expression* **)**	(5)
function-call-expression	(6)
functional-type-conv-expression	(7)

where *function-call-expression* is

postfix-expression **(** *expression-list*(optional) **)**

and *functional-type-conv-expression* are following sorts of expressions which perform explict cast:

Keywords

`reflexpr`

Predefined feature testing macros

__cpp_reflection (reflection TS)	a value of at least `201902` indicates that the Reflection TS is supported (macro constant)

Library support

Concepts

Defined in header `<experimental/reflect>`
Defined in namespace `std::experimental::reflect`
Defined in inline namespace `std::experimental::reflect::v1`

Object (reflection TS)	specifies that a type is a meta-object type (concept)
ObjectSequence (reflection TS)	specifies that a meta-object type is a meta-object sequence type (concept)
TemplateParameterScope (reflection TS)	specifies that a meta-object type reflects a template parameter scope (concept)
Named (reflection TS)	specifies that a meta-object type reflects an entity or alias with an associated (possibly empty) name (concept)
Alias (reflection TS)	specifies that a meta-object type reflects a type alias, namespace alias, or an alias introduced by a using-declaration (concept)
RecordMember (reflection TS)	specifies that a meta-object type reflects a *member-declaration* of a class (concept)
Enumerator (reflection TS)	specifies that a meta-object type reflects an enumerator (concept)
Variable (reflection TS)	specifies that a meta-object type reflects a variable or data member (concept)
ScopeMember (reflection TS)	specifies that a meta-object type satisfies `RecordMember`, `Enumerator`, or `Variable`, or reflects a namespace other than the global namespace (concept)
Typed (reflection TS)	specifies that a meta-object type reflects an entity with a type (concept)
Namespace (reflection TS)	specifies that a meta-object type reflects a namespace (concept)
GlobalScope (reflection TS)	specifies that a meta-object type reflects the global namespace

SNT

Reflection Currently

UNIVERSITY OF LUXEMBOURG

A simple example

stackoverflow

enum to string in modern C++11 / C++14 / C++17 and future C++20

Asked 8 years, 8 months ago Modified 3 months ago Viewed 350k times

549

Contrary to all other similar questions, this question is about using the new C++ features.

- 2008 c Is there a simple way to convert C++ enum to string?
- 2008 c Easy way to use variables of enum types as string in C?
- 2008 c++ How to easily map c++ enums to strings
- 2008 c++ Making something both a C identifier and a string?
- 2008 c++ Is there a simple script to convert C++ enum to string?
- 2009 c++ How to use enums as flags in C++?
- 2011 c++ How to convert an enum type variable to a string?
- 2011 c++ Enum to String C++
- 2011 c++ How to convert an enum type variable to a string?
- 2012 c How to convert enum names to string in c
- 2013 c Stringifying an conditionally compiled enum in C

After reading many answers, I did not yet find any:

- Elegant way using C++11, C++14 or C++17 new features
- Or something ready-to-use in Boost
- Else something planned for C++20

Solutions:

Store the strings in an `std::map`

`#define ENUM(…)`
to store strings automatically

Use a build tool script

Abuse user-defined literals

Top answer:

Magic Enum C++

A simple example

Magic Enum C++ by Neargye

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

Without any extra code!

Enum-to-string

```
Color color = Color::RED;
auto color_name = magic_enum::enum_name(color);
// color_name == "RED"
```

String-to-enum

```
std::string color_name{"GREEN"};
auto color = magic_enum::enum_cast<Color>(color_name);
// color = Color::GREEN
```

Magic Enum C++

How does it work?

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

```
template<Color C>
constexpr std::string enum_name() {
    return std::string(__PRETTY_FUNCTION__);
}
```


```
enum_name<RED>()
-> "std::string enum_name() [C = RED]"
```

Other reflection implementations

boostorg/describe

Boost Describe

Reflection based on macros

```
enum E2 { a, b, c, d };
BOOST_DESCRIBE_ENUM(E2, a, b, c, d)
BOOST_DESCRIBE_STRUCT(X, (), (m1, m2))
BOOST_DESCRIBE_CLASS(Y, (X), (m3, f), (m4), (m5))
```

boostorg/pfr

Boost PFR

Simple reflection based on TMP (template meta-programming)

```
struct my_struct {
    int i; char c; double d;
};

my_struct s{100, 'H', 3.141593};
std::cout << "my_struct has "
          << boost::pfr::tuple_size<my_struct>::value
          << " fields: " << boost::pfr::io(s) << "\n";

// my_struct has 3 fields: {100, H, 3.14159}
```

Serialization libraries in C++

Boost.PFR

```
struct my_struct {
    int i; char c; double d;
};

my_struct s{100, 'H', 3.141593};
std::cout << "my_struct has "
          << boost::pfr::tuple_size<my_struct>::value
          << " fields: " << boost::pfr::io(s) << "\n";

// my_struct has 3 fields: {100, H, 3.14159}
```

The answer: C++ structured bindings and SFINAE!

```
struct my_struct {
    int a, float b, std::string c;
};

int main() {
    auto [a, b, c] = my_struct{15, 120, "beep"};
    std::cout << c << std::endl; // beep
}
```

How does it work?

Works with pure C++17

Capabilities limited

Cannot retrieve names

Complex implementation

Only works with aggregates

More information:
Serializing structs with C++17 structured bindings
Björn Fahller

Other reflection implementations

rttrorg/rttr

RTTR

Runtime reflection based on manual definitions

```
struct MyStruct {
    MyStruct() {};
    void func(double) {};
    int data;
};

RTTR_REGISTRATION
{
    registration::class_<MyStruct>("MyStruct")
        .constructor<>()
        .property("data", &MyStruct::data)
        .method("func", &MyStruct::func);
}
```

veselink1/refl-cpp

refl-cpp

Reflection based on macros

```
struct A {
    int foo;
    void bar();
    void bar(int);
};

REFL_AUTO(
    type(A),
    field(foo, my::custom_attribute("foofoo")),
    func(bar, property(), my::custom_attribute("barbar"))
)
```

Compiler implementations

seanbaxter/circle

Circle compiler

Experimental C++ compiler with reflection features

```
template<typename type_t>
const char* name_from_enum(type_t x) {
  static_assert(std::is_enum<type_t>::value);

  switch(x) {
    @meta for(int i = 0; i < @enum_count(type_t); ++i) {
      case @enum_value(type_t, i):
        return @enum_name(type_t, i);
    }
    default:
      return nullptr;
  }
}
```

MSVC

Runtime reflection implementation

```
using namespace System;

enum class Options {
   Option1, Option2, Option3
};

int main() {
   array<String^>^ names = Enum::GetNames(Options::typeid);

   Console::WriteLine("there are {0} options in enum '{1}'",
               names->Length, Options::typeid);

   for (int i = 0 ; i < names->Length ; i++)
      Console::WriteLine("{0}: {1}", i, names[i]);

   Options o = Options::Option2;
   Console::WriteLine("value of 'o' is {0}", o);
}
```

Compiler implementations

lock3/meta

Lock3 LLVM Reflection & Metaprogramming Clang Fork

Based on P1240 with extra features

```
template<typename T>
char const* to_string (T value) {
    constexpr meta::info type = reflexpr(T);
    constexpr auto members = meta::members_of(type);
    template for (constexpr auto member : members) {
        if (valueof(member) == value)
            return meta::name_of(member);
    }
    return "<unknown>";
}

to_string(Color::RED) -> "RED"
to_string(Color::GREEN) -> "GREEN"
```

Reflection: Compile-Time Introspection of C++

Andrew Sutton, ACCU 2021

Compiler implementations

matus-chochlik/llvm-project

Matúš Chochlík's LLVM fork

Implementation of static reflection TS

```
template <typename E>
std::string_view enum_to_string(E e) {
  return select(get_enumerators(mirror(E)),
    [](auto& result, auto mo, auto e) {
      if (get_constant(mo) == e) {
        result = get_name(mo);
      }
    }, std::string_view{}, e);
}
```

Design of a C++ reflection API

Matúš Chochlík, MeetingC++ Online 2022

SNT

(De)serialization

UNIVERSITY OF LUXEMBOURG

What is serialization?

(de-)serialization

(un-)marshalling

(un-)packing

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```

serialization →

Storage or Channel

→ deserialization

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```

Serialization applications

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```

serialization →

Storage or Channel

→ deserialization

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```

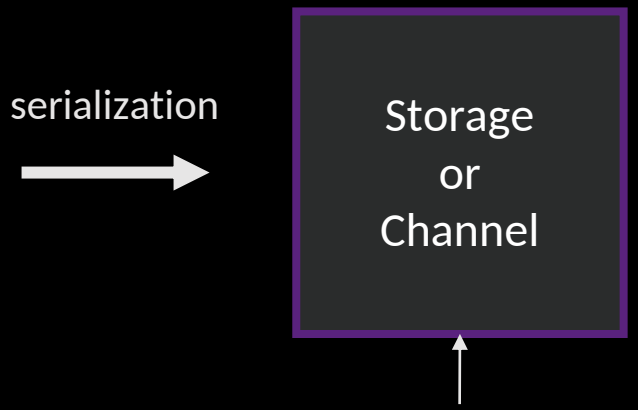
permanent storage
transmission to another application
transmission over a network
interface with a database
remote procedure call
wireless telemetry and telecommands
diagnostics
identifying changes
...

Storage

Communication

Serialization protocols

```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```




```
class Toast {
    int id = 7;
    float temperature = 32;
    std::string review = “tasty";
public:
    void bake();
}
```

Naive id = 7, temperature = 32, review = tasty

JSON {“id”: 7, “temperature”: 32, “review”: "tasty"}

BSON 4����id�������temperature�������@@review�����tasty��

XML <toast id="7"><temperature>32</temperature><review>tasty</review></toast>

Python Pickle ��4}�(�id�K temperature�G@@review��tasty�u.

ProtoBuf ??? tasty

...or a custom protocol

Serialization libraries in C++

fraillt/**bitsery**

capnproto/**capnproto**

USCiLab/**cereal**

felixguendling/**cista**

mikeloomisgg/**cppack**

google/**flatbuffers**

kaitai-io/**kaitai_struct**

protocolbuffers/**protobuf**

boostorg/**serialization**

KonanM/**tser**

eyalz800/**zpp_bits**

Important distinctions:

Pre-defined data format

User-defined data format

Run-time

Compile-time
(constexpr)

Compilation
+
Code generation

Run within C++

Multi-language implementation

C++ only

Serialization libraries in C++

Example: Google ProtoBuf

toast.proto

```
syntax = "proto2";

package tutorial;

message Toast {
  required int32 id = 1;
  required float temperature = 2;
  required string review = 3;
}
```

```
> protoc toast.proto –cpp_out=pout
```

Compilation
+
Code generation

```
// Generated by the protocol buffer compiler.  DO NOT EDIT!
// source: toast.proto

#include "toast.pb.h"
#include <algorithm>
#include <google/protobuf/io/coded_stream.h>
#namespace tutorial {
class ToastDefaultTypeInternal {
 public:
  ::PROTOBUF_NAMESPACE_ID::internal::ExplicitlyConstructed<Toast> _instance;
} _Toast_default_instance_;
}  // namespace tutorial
class Toast PROTOBUF_FINAL :
    public ::PROTOBUF_NAMESPACE_ID::Message /*
@@protoc_insertion_point(class_definition:tutorial.Toast) */ {
 public:
  inline Toast() : Toast(nullptr) {};
  virtual ~Toast();

  Toast(const Toast& from);
  Toast(Toast&& from) noexcept
    : Toast() {
    *this = ::std::move(from);
  }

  inline Toast& operator=(const Toast& from) {
    CopyFrom(from);
    return *this;
  }
  inline Toast& operator=(Toast&& from) noexcept {
    if (GetArena() == from.GetArena()) {
      if (this != &from) InternalSwap(&from);
    } else {
      CopyFrom(from);
    }
    return *this;
  }

  inline const ::PROTOBUF_NAMESPACE_ID::UnknownFieldSet& unknown_fields() const {
    return
_internal_metadata_.unknown_fields<::PROTOBUF_NAMESPACE_ID::UnknownFieldSet>(::PROTOBUF_NAMESPACE_I
D::UnknownFieldSet::default_instance);
  }
  inline ::PROTOBUF_NAMESPACE_ID::UnknownFieldSet* mutable_unknown_fields() {
    return _internal_metadata_.mutable_unknown_fields<::PROTOBUF_NAMESPACE_ID::UnknownFieldSet>();
  }

  static const ::PROTOBUF_NAMESPACE_ID::Descriptor* descriptor() {
    return GetDescriptor();
  }
  static const ::PROTOBUF_NAMESPACE_ID::Descriptor* GetDescriptor() {
    return GetMetadataStatic().descriptor;
  }
  static const ::PROTOBUF_NAMESPACE_ID::Reflection* GetReflection() {
    return GetMetadataStatic().reflection;
  }
  static const Toast& default_instance();

…
```

Serialization libraries in C++

Example: cereal

Approach used by many serialization libraries!

```
struct MyRecord
{
  uint8_t x, y;
  float z;

  template <class Archive>
  void serialize( Archive & ar )
  {
    ar( x, y, z );
  }
};

// And then you can call all library
serialization/deserialization functions
```

A function or other instruction to the library, to let it know what and how to serialize

Runs within C++

Serialization libraries in C++

Example: bitsery

```cpp
enum class MyEnum:uint16_t { V1,V2,V3 };
struct MyStruct {
    uint32_t i;
    MyEnum e;
    std::vector<float> fs;
};
```

```cpp
template <typename S>
void serialize(S& s, MyStruct& o) {
    s.value4b(o.i);
    s.value2b(o.e);
    s.container4b(o.fs, 10);
}
```

Specify binary types by hand if you want more control and a stable interface

Runs within C++

Serialization libraries in ~~C++~~ Python

Example: Python's construct library (construct/construct)

```python
seq = Sequence(
    Int16ub,
    CString("utf8"),
    GreedyBytes,
)

object = (7, "foo", b"bar")

seq.compile(object)
# b'\x00\x07foo\x00bar'
```

User-defined data format

Serialization libraries in C++

Example: Cista

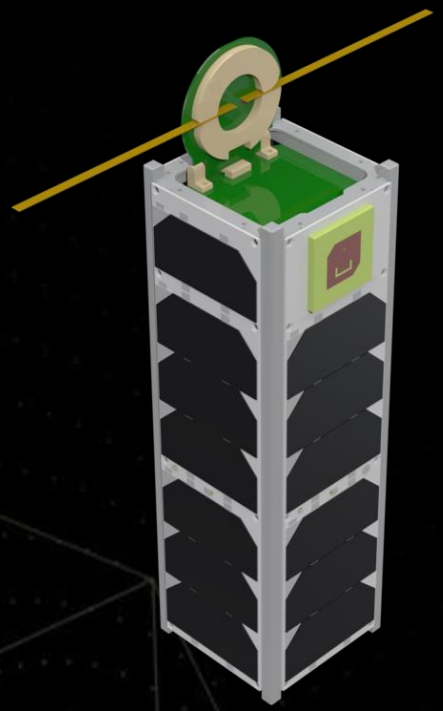
```
struct my_struct {
  int a_ {0};
  float b_ {1};
  cista::raw::string c_;
};

my_struct obj{7, 32.0f, cista::raw::string{"toast"}};

std::vector<unsigned char> buf = cista::serialize(obj);
-> "p???!toast?????"
```

Serializes without any other information!

Runs within C++

Serialization in practice

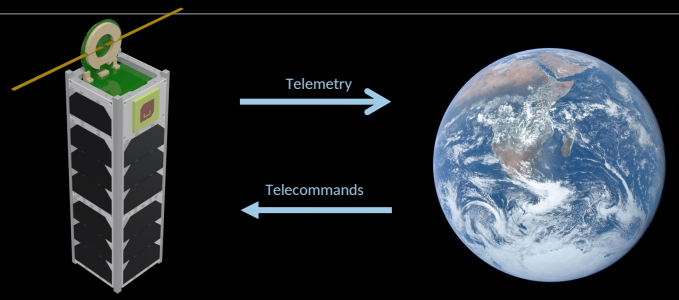
Use case: Telecommands and telemetry of AcubeSAT satellite



Telemetry

Telecommands

Serialization in practice



Use case: Telecommands and telemetry of AcubeSAT satellite

In telemetry parser:

```
message.skipBytes(2); // Skips reading the application ID
uint16_t eventDefinitionID = message.readEnum16();
uint16_t eventActionDefinitionID = message.readEnum16();
```

In internal command generator:

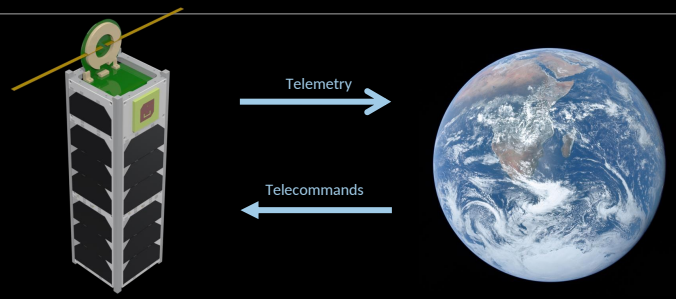
```
message.appendEnum16(APID);
message.appendEnum16(eventDefinitionID);
message.appendEnum16(eventActionDefinitionID);
```

In ground station operations declaration:

```
<CommandContainer name="ST19_Enable_Definition">
    <EntryList>
        <FixedValueEntry name="APID" binaryValue="0000" sizeInBits="16" />
        <ArgumentRefEntry argumentRef="Event_Definition_ID" />
        <FixedValueEntry name="Event_Action_Definition_ID" binaryValue="0000" sizeInBits="16" />
    </EntryList>
    <BaseContainer containerRef="AcubeSATTC" />
</CommandContainer>
```

Serialization in practice

Use case: Telecommands and telemetry of AcubeSAT satellite



Thoughts for the far future

What if:

- a C++ library could read this XML definition at compile-time
- generate the functions to parse and generate these messages
- report any errors/incompatibilities at compile time
- help the optimizer make this as fast as possible

```
<CommandContainer name="ST19_Enable_Definition">
    <EntryList>
        <FixedValueEntry name="APID" binaryValue="0000" sizeInBits="16" />
        <ArgumentRefEntry argumentRef="Event_Definition_ID" />
        <FixedValueEntry name="Event_Action_Definition_ID" binaryValue="0000" sizeInBits="16" />
    </EntryList>
    <BaseContainer containerRef="AcubeSATTC" />
</CommandContainer>
```

What do I want from my ideal serialization library?

User-defined data format

Compile-time
(constexpr)

Runs within C++

Minimal repetition

Supports other languages

High performance

Extensibility

SNT

Let's create ...yet another serialization library

UNIVERSITY OF LUXEMBOURG

ok-serializer

Available at: https://github.com/kongr45gpen/ok-serializer/

Just a proof-of-concept,
not meant for serious use

Design decisions and building blocks

Using Matúš Chochlík's LLVM compiler with Reflection TS support

+ updated to Clang 17 for more C++23 features

matus-chochlik/llvm-project

kongr45gpen/llvm-project

Requirement: Almost everything possible at compile-time

Requirement: No exceptions ← std::expected instead

Requirement: No dynamic memory allocation

Requirement: Only standard (or experimental..) C++

Short-term goal: Serialize into JSON with one line

Long-term goal: Take an XML definition and serialize at compile time

NOTE

There already are reflective implementations of serialization!

matus-chochlik/mirror

A Faster Serialization Library Based on Compile-time Reflection and C++ 20 - Yu Qi - CppCon 2022

Let’s get started!

by testing reflection

Enumeration to string, as always

```
enum class MyEnumeration {
    RED, GREEN, BLUE
};

auto main() -> int {
    static_assert(enum_to_string(MyEnumeration::RED) == "RED");
    std::cout << enum_to_string(MyEnumeration::RED) << std::endl;
}
```

Starting with a basis of a reflection library

Serialization to buffer

```
std::string string;
okser::out::dynamic out(string);

auto result = okser::serialize<okser::uint<2>>(out, 0x6869);
// string = “hi”
```

Deserialization from buffer

```
std::string string("\x70\xAF");
okser::in::range in{string};

auto result = okser::deserialize<okser::uint<2, okser::end::le>>(in);
std::cout << result.value() << std::endl;
// 44912
```

Starting with a basis of a reflection library

Convenience methods

Serialization to buffer

```
auto result = okser::serialize_to_string<okser::uint<2>>(0x6869);
// hi
```

Deserialization from buffer

```
auto result = okser::deserialize<okser::uint<2, okser::end::le>>("\x70\xAF");
std::cout << result.value() << std::endl;
// 44912
```

Input/Output types

Input

```
namespace okser {
namespace in {

template<std::ranges::input_range R = std::string>
class range {
private:
    typename C::const_iterator begin;
    typename C::const_iterator end;
public:
    constexpr range(Const_Iterator begin, Const_Iterator end) : begin(begin), end(end) {}
    constexpr range(const R &_range) : begin(_range.cbegin()), end(_range.cend()) {}
…
```

Output

```
namespace out {

template<std::ranges::output_range<uint8_t> C> requires (std::ranges::forward_range<C>)
class range {
private:
    typename C::iterator current;
    typename C::const_iterator last;
public:
    constexpr range(C &container) : current(container.begin()), last(container.end()) {}

    template<typename T>
    constexpr empty_result add(const T &value) {
        if (current == last) {
            return std::unexpected(error_type::not_enough_output_bytes);
        }
        *current = value;
        current++;

        return {};
    }
```

Data Types

Simple data types

```
template<int Bytes, end Endianness = end::be>
struct okser::uint

template<int Bytes, end Endianness = end::be>
struct okser::sint

struct okser::varint

template<typename Enum, int Bytes = sizeof(std::underlying_type<Enum>), end Endianness = end::be>
struct okser::enumv

template<int Bytes = 4, end Endianness = end::be>
struct okser::floatp
```

Data Types

Simple data types

```
template<class Size = okser::uint<1>>
struct okser::pascal_string

template<uint8_t Terminator = 0>
struct okser::terminated_string
```

Data Types

Compound data types

```
template<class... Types>
class okser::bundle

template<typename T, typename Size = okser::uint<1>>
class okser::length_prefixed_vector
```

Compound Data Types

A little look behind the scenes

Using parameter pack expansions to iterate over tuples at compile-time

```
template<class... Types>
class bundle {
private:
    using IndexSequence = std::make_index_sequence<sizeof...(Types)>;
public:
    using TypesTuple = std::tuple<Types...>;
    using DefaultType = std::tuple<typename Types::DefaultType...>;

    template<OutputContext Context, typename... Values>
    requires (Serializer<Types>, ...)

    constexpr static empty_result serialize(Context &&output, Values... values) {
        std::tuple<serializable_value<Types, Values>...> typeValues{values...};

        std::apply(
                [&output](auto &&...v) { ((internal::serialize_one(output, v)), ...); },
                typeValues);

        if (output.error) {
            return std::unexpected(output.error.value());
        }

        return {};
    }
```

Let's talk about errors

Definition of return value types:

```
template<class T>
using result = std::expected<T, parse_error>;

using empty_result = std::expected<void, parse_error>;
```

Used for de-serialization

Used for Serialization

```
enum class error_type : uint8_t {
    not_enough_input_bytes,
    not_enough_output_bytes,
    io_error,
    malformed_input,
    overflow,
    redundant_mismatch
};
```

```
struct parse_error {
    error_type type;
//    std::any error;

    explicit(false) constexpr parse_error(error_type type) noexcept: type(type) {}

    error_type operator()() const noexcept {
        return type;
    }

    error_type operator*() const noexcept {
        return type;
    }
};
```

Testing

Fun with Catch2 and idempotence

```
TEST_CASE("floatp idempotence") {
    SECTION("single-precision") {
        float i = GENERATE(take(100, random(-1e30, 1e30)));

        auto str = serialize_to_string<okser::floatp<4>>(i);

        auto result = deserialize<okser::floatp<4>>(str);

        REQUIRE(result.has_value());
        CHECK_THAT(i, WithinRel(*result));
    }
}
```

```
SECTION("varint and signed_varint equivalence") {
    auto i = GENERATE(take(100, random <uint32_t> (0, std::numeric_limits<int32_t>::max())));

    auto str1 = serialize_to_string<okser::varint>(i);
    auto str2 = serialize_to_string<okser::signed_varint<>>(i);

    CHECK(str1 == str2);
}
```

Let's add some reflection to the mix!

Uses Matúš Chochlík's mirror library

```cpp
#if __cpp_reflection >= 201902L

template<class T, Output Out, auto config = configuration()>
constexpr void serialize_struct(Out &&output, const T &object) {
    auto mirrored_struct = mirror(T);

    for_each(get_data_members(mirrored_struct), [&](auto member) {
        const auto &value = get_value(member, object);
        using type = std::remove_cvref_t<decltype(value)>;

        using serializer = decltype(config)::template default_serializers<type>::ser;

        auto result = serialize<serializer>(output, value);
    });
}
```

Converts uint8_t to okser::uint

Let’s add some reflection to the mix!

End result

```
struct Structure {
    int8_t a;
    uint16_t b;
};

auto main() -> int {
    using namespace std::experimental;

    Structure s{104, 26913};

    using ss = reflexpr(Structure);
    std::cout << "Structure has " << reflect::get_size_v<reflect::get_data_members_t<ss>> << " data members"
              << std::endl;

    std::cout << "Result: " << okser::serialize_struct_to_string(s) << std::endl;
    // Result: hi!
}
```

Let’s add some reflection to the mix!

Implementing JSON

```
14  namespace json {
15
16  struct json_configuration;
17
18  struct boolean : public okser::internal::type {...};
30
31  struct null : public okser::internal::type {...};
40
41  struct number : public okser::internal::type {...};
92
93  struct string : public okser::internal::type {...};
134
135 template<class Configuration = json_configuration>
136 struct array : public okser::internal::type {...};
216
217 template<class Configuration = json_configuration>
218 struct object : public okser::internal::type {...};
270
```

Let’s add some reflection to the mix!

Testing JSON

```
TEST_CASE("json types") {
    SECTION("number") {
        auto result = okser::serialize_to_string<okser::json::number>(-1234567890);
        CHECK_THAT(result, Equals("-1234567890"s));
    }

    SECTION("string") {
        auto result = okser::serialize_to_string<okser::json::string>("toast"s);
        CHECK_THAT(result, Equals("\"toast\""s));

        result = okser::serialize_to_string<okser::json::string>("\"boop\\toast\":\t\r"s);
        CHECK_THAT(result, Equals(R"("\"boop\\toast\":\t\r")"s));

        result = okser::serialize_to_string<okser::json::string>("\x7F\x32\x00\x01\xFE"s);
        CHECK_THAT(result, Equals(R"("\x7F2\x00\x01\xFE")"s));
    }

    SECTION("array") {
        auto result = okser::serialize_to_string<okser::json::array<>>(std::vector<int>{1, 2, 3});
        CHECK_THAT(result, Equals("[1, 2, 3]"s));

        result = okser::serialize_to_string<okser::json::array<>>(std::vector<bool>{true, false, true, false});
        CHECK_THAT(result, Equals("[true, false, true, false]"s));

        result = okser::serialize_to_string<okser::json::array<>>(std::vector<std::string>{"toast"s, "is"s, "nice"s});
        CHECK_THAT(result, Equals(R"(["toast", "is", "nice"])"s));

        result = okser::serialize_to_string<okser::json::array<>>(std::make_pair(1, false));
        CHECK_THAT(result, Equals(R"([1, false])"s));
    }

    SECTION("object") {
        ThreeByteStructure object{119, 86};
        auto result = okser::serialize_to_string<okser::json::object<>>(object);
        CHECK_THAT(result, Equals(R"({"a": 119, "b": 86})"s));

        ComplexStructure complex{1.4f, "toast", {1, 2, 3}, {119, 86}};
        result = okser::serialize_to_string<okser::json::object<>>(complex);
        CHECK_THAT(result,
                   Equals(R"({"number": 1.40, "text": "toast", "numbers": [1, 2, 3], "structure": {"a": 119, "b": 86}})"s));
    }
```

SNT

Conclusions

UNIVERSITY OF LUXEMBOURG

ok-serializer

✓ Can convert arbitrary objects to JSON

✓ Serialization and deserialization at compile time

✓ Possibility to extend with little effort

✓ Useful for embedded systems

✗ Serializing with custom formats is janky

✗ Still cannot parse XML definitions, Kaitai structs etc.

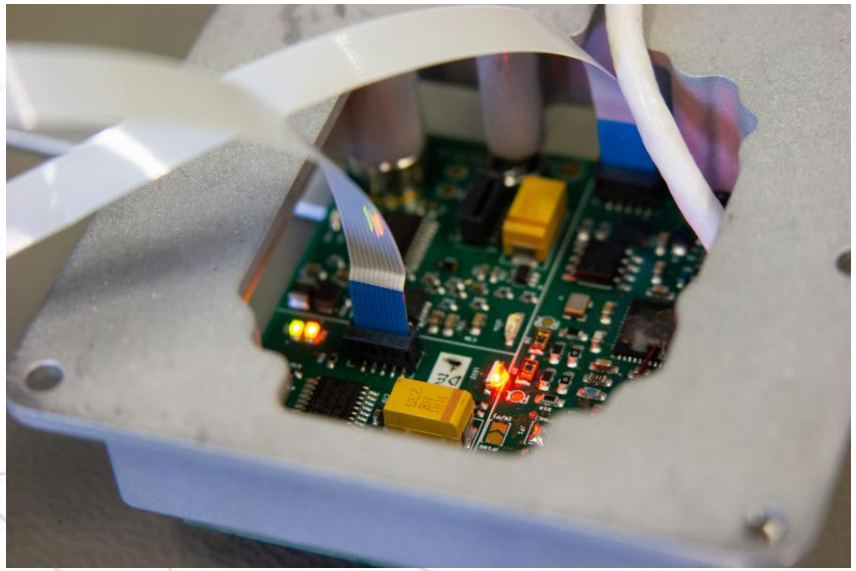
✗ Would be useful to *create* completely new structs, variables at compile time (metaprogramming)

Still far away from production-ready

Compare with annotations on other languages:

```
public class Toast {
    @Uint(size=2)
    public int id;
    @Float
    public float temperature;
    @PascalString
    public String review;
}
```

A day of life in ok-serializer



ok-serializer going to space?

Goal for future projects:

Single source of truth for generated and received data

```
file = Struct(
    "service" / Const(b"f"),
    "message" / Const(b"f"),
    "start" / Int32ub,
    "length" / Int32ub,
    "data" / Bytes(this.length)
) * "Contents of a file"

parameter_value = Struct(
     "service" / Const(b"p"),
     "message" / Const(b"v"),
     "parameter" / common.parameter,
     "value" / Float32b
) * "Single value of a parameter"

parameter_health = Struct(
    "service" / Const(b"p"),
    "message" / Const(b"h"),
    "cpu" / Float32b,
    "memory" / Float32b,
    "flash" / Float32b,
    "uptime" / Float32b,
) * "Health status of the satellite"
```

← Parsing
← Generation
Documentation
Interface Definition
Ground Station
Radio-amateur

(example from Python)

Another nice-to-have?

```
auto function = std::reflection::get_item_from_signature<"communication::get_timestamp">();

function->call(arguments);
```

Final Thoughts

Implementations exist, feel free to play around

and use llvm-compiler with reflection

Link: https://github.com/kongr45gpen/ok-serializer/
Mirror library reference: https://matus-chochlik.github.io/mirror/doxygen/
cppreference: https://en.cppreference.com/w/cpp/experimental/reflect

Not updated with new ^ operator and drafts!

Metaprogramming capabilities added by P2996R0

```
constexpr auto T = std::meta::synth_struct({
  nsdm_description(^int),
  nsdm_description(^char),
  nsdm_description(^double),
});

// T is a reflection of the type
// struct {
//   int _0;
//   char _1;
//   double _2;
// }
```

P2996 is active
https://github.com/brevzin/cpp_proposals/tree/master/2996_reflection

SnT

Fin

More information: https://snt.uni.lu/research/spasys

Contact:

konstantinos.kanavouras@uni.lu

@kongr45gpen@mastodon.social

https://github.com/kongr45gpen

Media Credits

FreePik

UNIVERSITY OF LUXEMBOURG