C++26 Static Reflection

P2996R13 + others

Agenda

- What is (static) reflection
- Why we want reflection
- How to use static reflection in C++26
- In other languages: Rust, Java, Python, JS

What is Reflection

“Can a string be a variable name?”

Informally, reflection is language feature that allows the language to modify itself.

- Assembly
- Compiled languages
- Interpreted languages

Static reflection is the (restricted) ability to do this during compile time

Reflection over fields, methods, types, annotations, code

Applications of Reflection

- Object Relational Mapping (ORM)
- Serialisation Deserialisation (Serde)
- Struct of Arrays (SoA)
- Transpilers (code reflection)

Pre-C++26 Runtime Type Information (RTTI)

C++ already has (very limited) runtime reflection

- `typeid` gives you mangled type names
- `std::dynamic_cast` helps detect castability

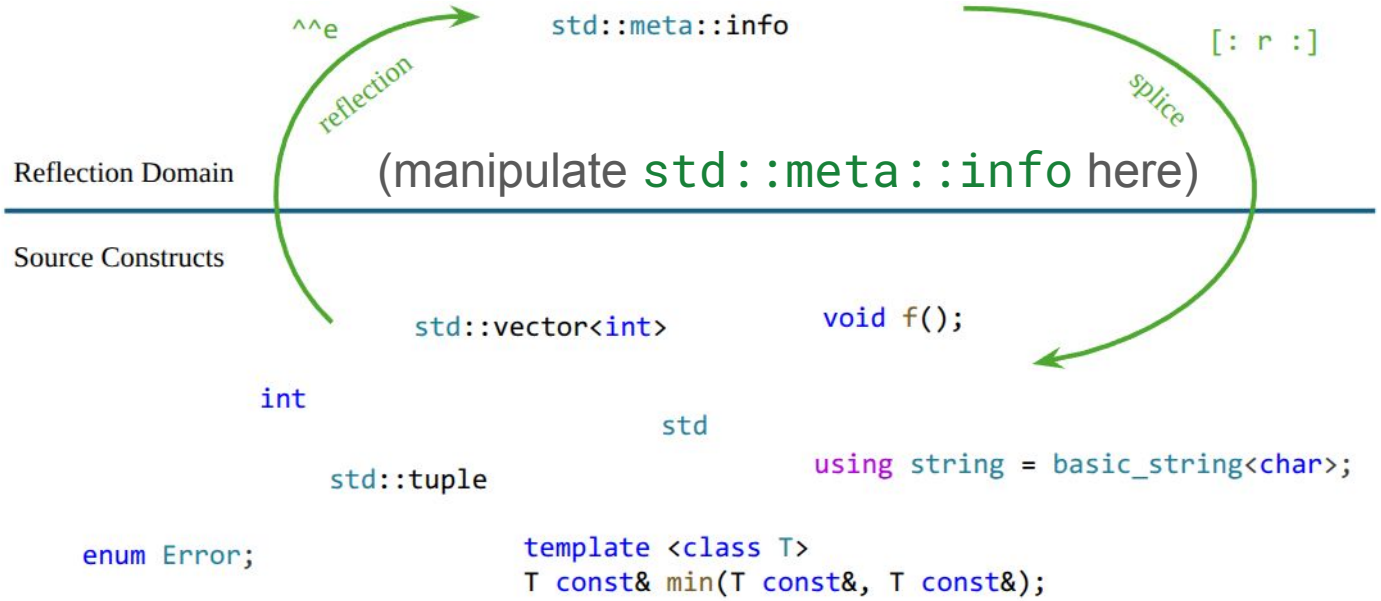
This can’t let us do much sadly

C++26 Static Reflection Proposals

1. P2996R13: Reflection for C++26
2. P3394R4: Annotations for Reflection
3. P3293R3: Splicing a Base Class Subobject
4. P3491R3: `define_static_{string,object,array}`
5. P1306R5: Expansion Statements
6. P3096R12: Function Parameter Reflection in Reflection for C++26
7. P3560R2: Error Handling in Reflection

P2996R13

- A single reflection type `std::meta::info` for good reason
- Diagram credit: Tim Song's cpp meetup 2 slides



Playground

Godbolt, or to set up locally using cursor

1. git clone https://github.com/bloomberg/clang-p2996
2. build p2996 compiler
3. **build libc++** with p2996 compiler because we need to `#include <meta>`
4. compile your program and link against libc++

```
llvm/build/bin/clang++ -std=c++26 -freflection-latest
-Ilibcxx/include -nostdinc++ -isystem libcxx/include
-Llibcxx/build/lib -lc++ -lm -lc -pthread your_program.cc
-o your_program
```

P2996R13 Key features

4.4.11 identifier_of, display_string_of, source_location_of
4.4.12 type_of, parent_of, dealias
4.4.13 object_of, constant_of
4.4.14 template_of, template_arguments_of
4.4.15 members_of, static_data_members_of, nonstatic_data_members_of, bases_of, enumerators_of
4.4.16 substitute
4.4.17 reflect_constant, reflect_object, reflect_function
4.4.18 extract<T>
4.4.19 data_member_spec, define_aggregate

P2996R13 Examples: `^^` and `[: :]`

`typename` is only required if compiler might be unsure if the expression is a type

```
constexpr auto r = ^^int;
typename[:r:] x = 42;        // Same as: int x = 42;
typename[:^^char:] c = '*';  // Same as: char c = '*';
```

`r` is of type `std::meta::info`

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
  if (n == 0) return ^^S::i;
  else if (n == 1) return ^^S::j;
}

int main() {
  S s{0, 0};
  s.[:member_number(1):] = 42;  // Same as: s.j = 42;
  s.[:member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

P2996R13 Examples: `^^` and `[: :]`

```
constexpr auto r = ^^int;
typename[:r:] x = 42;        // Same as: int x = 42;
typename[:^^char:] c = '*';  // Same as: char c = '*';
```

`r` is of type `std::meta::info`

Proof:

```
// trick to print types
template<typename T>
struct PrintType;
```

`typeid(r)` doesn't seem to work (yet?)

```
woof.cc:40:26: error: implicit instantiation of undefined template 'PrintType<const meta::info>'
   40 |    PrintType<decltype(r)> _;
      |                          ^
woof.cc:34:8: note: template is declared here
   34 | struct PrintType;
      |        ^
1 error generated.
```

P2996R13 Examples: `^^` and `[: :]`

`typename` is only required if compiler might be unsure if the expression is a type

```
constexpr auto r = ^^int;
typename[:r:] x = 42;        // Same as: int x = 42;
typename[:^^char:] c = '*';  // Same as: char c = '*';
```

`r` is of type `std::meta::info`

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
  if (n == 0) return ^^S::i;
  else if (n == 1) return ^^S::j;
}

int main() {
  S s{0, 0};
  s.[:member_number(1):] = 42;  // Same as: s.j = 42;
  s.[:member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

P2996R13 Examples

How to not hardcode ??

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
  if (n == 0) return ^^S::i;
  else if (n == 1) return ^^S::j;
}

int main() {
  S s{0, 0};
  s.[:member_number(1):] = 42;  // Same as: s.j = 42;
  s.[:member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

P2996R13 Examples: `nonstatic_data_members_of`

access_context basically helps prevent scoping issues
https://isocpp.org/files/papers/P2996R13.html#meta.reflection.access.context-access-control-context

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
  auto ctx = std::meta::access_context::current();
  return std::meta::nonstatic_data_members_of(^^S, ctx)[n];
}

int main() {
  S s{0, 0};
  s.[:member_number(1):] = 42;  // Same as: s.j = 42;
  s.[:member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

P2996R13 Examples: `access_context`

access_context basically helps prevent scoping issues https://isocpp.org/files/papers/P2996R13.html#meta.reflection.access.context-access-control-context

`std::meta::access_context::unchecked()` gives reflection access to private members

Most of the time we want to use `std::meta::access_context::current()`

https://godbolt.org/z/zM4E39Yah illustrating differences between the 2

P2996R13 Examples: `identifier_of`

```
struct S { unsigned i:2, j:6; };

consteval auto member_named(std::string_view name) {
  auto ctx = std::meta::access_context::current();
  for (std::meta::info field : nonstatic_data_members_of(^^S, ctx)) {
    if (has_identifier(field) && identifier_of(field) == name)
      return field;
  }
}

int main() {
  S s{0, 0};
  s.[:member_named("j"):] = 42;  // Same as: s.j = 42;
  s.[:member_named("x"):] = 0;   // Error (member_named("x") is not a constant).
}
```

Life before p2996

If you wanted to map a list of **types** to a list of **sizes**. Need to use variadic templates

```
template<class...> struct list {};

using types = list<int, float, double>;

constexpr auto sizes = []<template<class...> class L, class... T>(L<T...>) {
    return std::array<std::size_t, sizeof...(T)>{{ sizeof(T)... }};
}(types{});
```

https://godbolt.org/z/Y114rsbTb fixes a small typo

Life after p2996

If you wanted to map a list of **types** to a list of **sizes**. Need to use variadic templates

```
template<class...> struct list {};

using types = list<int, float, double>;

constexpr auto sizes = []<template<class...> class L, class... T>(L<T...>) {
    return std::array<std::size_t, sizeof...(T)>{{ sizeof(T)... }};
}(types{});
```

`std::meta::info` can work with `std::ranges::transform` 😀

```
constexpr std::array types = {^^int, ^^float, ^^double};
constexpr std::array sizes = []{
  std::array<std::size_t, types.size()> r;
  std::ranges::transform(types, r.begin(), std::meta::size_of);
  return r;
}();
```

Template Metaprogramming

C++11 parameter pack and std::tuple

Qn: Why did C++11 introduce tuple? Isn’t struct enough?

C++11 parameter pack and std::tuple

Qn: Why did C++11 introduce tuple? Isn't struct enough?

(Opinionated) Ans:

- We probably introduced parameter packs first https://en.cppreference.com/w/cpp/thread/async.html
- We wanted to reify / bundle (`std::make_tuple`) parameter packs
- `std::tuple` is the type of the reified parameter pack
- `std::apply` (in C++17) helps unbundle this parameter pack

With `std::tuple`

```cpp
template <class F, class... Args>
auto async_later(F&& f, Args&&... args) {
    // Reify pack so it can be moved into the async task.
    auto captured = std::make_tuple(std::forward<Args>(args)...);
    return std::async(std::launch::async,
        [fn = std::forward<F>(f), captured = std::move(captured)]() mutable {
            return std::apply(f: std::move(fn), t: std::move(captured));
        });
}

#include <string>
#include <iostream>

int main() {
    auto fut = async_later(
        f: [](std::string s, int x) {
            return s + std::to_string(val: x);
        },
        a1: std::string(s: "value="), a2: 42
    );
    std::cout << fut.get() << "\n"; // prints "value=42"
}
```

Without `std::tuple`

```cpp
template <class F, class... Args>
auto async_later(F&& f, Args&&... args) {
    return std::async(std::launch::async,
        [fn = std::forward<F>(f), ... xs = std::forward<Args>(args)]() mutable {
            return fn(std::move(xs)...);
        });
}
```

```cpp
#include <string>
#include <iostream>

int main() {
    auto fut = async_later(
        f: [](std::string s, int x) {
            return s + std::to_string(val: x);
        },
        a1: std::string(s: "value="), a2: 42
    );
    std::cout << fut.get() << "\n"; // prints "value=42"
}
```

How to print a tuple? (C++11)

SFINAE / template specialisation

Downsides

- Verbose code
- Tuple size can’t >= 1024
- For old compilers might be O(N^2) at *compile time.* Modern ones use compiler intrinsics

```cpp
template<std::size_t I = 0, typename... Ts>
typename std::enable_if<I == sizeof...(Ts), void>::type
print_tuple_impl(std::ostream&, const std::tuple<Ts...>&) {
    // Base case: do nothing when we've reached the end
}

template<std::size_t I = 0, typename... Ts>
typename std::enable_if<I < sizeof...(Ts), void>::type
print_tuple_impl(std::ostream& os, const std::tuple<Ts...>& t) {
    if (I > 0) os << ", ";
    os << std::get<I>(t);
    print_tuple_impl<I + 1>(os, t);  // Recurse to next element
}

template<typename... Ts>
void print(const std::tuple<Ts...>& t) {
    std::cout << "(";
    print_tuple_impl(std::cout, t);
    std::cout << ")\n";
}
```

How to print a tuple? (C++14)

Replace recursion for the index with `std::integer_sequence`

We can emulate fold expressions using `std::initializer_list`

```cpp
template<typename Tuple, std::size_t... Is>
void print_tuple_impl(const Tuple& t, std::index_sequence<Is...>) {
    std::size_t n = 0;
    // Use initializer list trick for sequenced evaluation
    (void)std::initializer_list<int>{
        (std::cout << (n++ ? ", " : "") << std::get<Is>(t), 0)...
    };
}

template<typename... Ts>
void print(const std::tuple<Ts...>& t) {
    std::cout << "(";
    print_tuple_impl(t, std::index_sequence_for<Ts...>{});
    std::cout << ")\n";
}
```

C++14 std::integer_sequence

A compile-time list of integers, stored as template parameters

```
using seq2 = std::integer_sequence<int, 42, 17, 99, 3, 28>;
using sorted2 = bubble_sort_t<seq2>;
print_sequence<sorted2>::print();
```

How do compilers implement `std::make_integer_sequence` efficiently ?

https://stackoverflow.com/questions/25372805/how-exactly-is-stdmake-integer-sequence-implemented modern ones use intrinsics

P2996R13 Examples: `substitute`

Constructs `std::integer_sequence<T, 0, 1, 2, 3>`

```
#include <utility>
#include <vector>

template<typename T>
consteval std::meta::info make_integer_seq_refl(T N) {
  std::vector args{^^T};
  for (T k = 0; k < N; ++k) {
    args.push_back(std::meta::reflect_constant(k));
  }
  return substitute(^^std::integer_sequence, args);
}

template<typename T, T N>
  using make_integer_sequence = [:make_integer_seq_refl<T>(N):];
```

P2996R13 Examples: `extract`

`std::meta::info` may or may not contain the actual value, which can be extracted using `std::meta::extract`

4.4.18 `extract<T>`

```
namespace std::meta {
  template<typename T> consteval auto extract(info) -> T;
}
```

If `r` is a reflection for a value of type `T`, `extract<T>(r)` is a prvalue whose evaluation computes the reflected value.

Otherwise, it's *ill-formed*.

For other reflection values `r`, `extract<T>(r)` is ill-formed.

P2996R13 Examples: `define_aggregate`

Allows “programmatically defining” a struct by specifying a `vector<meta::info>`

e.g. Struct => Struct of Arrays https://godbolt.org/z/vqxzMoPcj

```
template<typename... Ts> struct Tuple {
  struct storage;
  consteval {
    define_aggregate(^^storage, {data_member_spec(^^Ts)...})
  }
  storage data;

  Tuple(): data{} {}
  Tuple(Ts const& ...vs): data{ vs... } {}
};
```

P1306R5: Expansion Statements

How to print a tuple? (C++14) (Revisited)

Replace recursion for the index with `std::integer_sequence`

We can emulate fold expressions using `std::initializer_list`

```cpp
template<typename Tuple, std::size_t... Is>
void print_tuple_impl(const Tuple& t, std::index_sequence<Is...>) {
    std::size_t n = 0;
    // Use initializer list trick for sequenced evaluation
    (void)std::initializer_list<int>{
        (std::cout << (n++ ? ", " : "") << std::get<Is>(t), 0)...
    };
}

template<typename... Ts>
void print(const std::tuple<Ts...>& t) {
    std::cout << "(";
    print_tuple_impl(t, std::index_sequence_for<Ts...>{});
    std::cout << ")\n";
}
```

How to print a tuple? (C++17)

`std::apply` helps unpack tuple, so no need for helper function anymore!

No need for `std::initializer_list` thanks to fold expressions

The instantiation of a *fold expression* expands the expression `e` as follows:

1) Unary right fold `(E op ...)` becomes $(E_1\ op\ (\dots\ op\ (E_{N-1}\ op\ E_N)))$
2) Unary left fold `(... op E)` becomes $(((E_1\ op\ E_2)\ op\ \dots)\ op\ E_N)$
3) Binary right fold `(E op ... op I)` becomes $(E_1\ op\ (\dots\ op\ (E_{N-1}\ op\ (E_N\ op\ I))))$
4) Binary left fold `(I op ... op E)` becomes $((((I\ op\ E_1)\ op\ E_2)\ op\ \dots)\ op\ E_N)$

How to print a tuple? (C++17)

`std::apply` helps unpack tuple, so no need for helper function anymore!

No need for `std::initializer_list` thanks to fold expressions

```
template<typename... Ts>
void print(const std::tuple<Ts...>& t) {
    std::cout << "(";
    std::apply(f: [](const auto&... args) {
        std::size_t n = 0;
        ((std::cout << (n++ ? ", " : "") << args), ...);
    }, t);
    std::cout << ")\n";
}
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p1306r5.html template for

How to print a tuple? (C++26)

`template for` helps to avoid using `std::apply`

```cpp
// Create an array of reflected index constants at compile time
template<std::size_t N>
consteval auto make_index_reflections() {
    std::vector<std::meta::info> result;
    result.reserve(N);
    for (std::size_t i = 0; i < N; ++i) {
        result.push_back(std::meta::reflect_constant(i));
    }
    return std::define_static_array(result);
}

template<typename... Ts>
void print(const std::tuple<Ts...>& t) {
    std::print("(");
    bool first = true;
    // template for iterates at compile-time over reflected indices
    template for (constexpr auto I : make_index_reflections<sizeof...(Ts)>()) {
        if (!first) std::print(", ");
        std::print("{}", std::get<([:I:])>(t));  // Parenthesize splice in template arg
        first = false;
    }
    std::println(")");
}
```

`template for` and `define_static_array`

```
template<typename E, bool Enumerable = std::meta::is_enumerable_type(^^E)>
  requires std::is_enum_v<E>
constexpr std::string_view enum_to_string(E value) {
  if constexpr (Enumerable)
    template for (constexpr auto e :
                  std::define_static_array(std::meta::enumerators_of(^^E)))
      if (value == [:e:])
        return std::meta::identifier_of(e);

  return "<unnamed>";
}
```

https://isocpp.org/files/papers/P3491R3.html define_static_{string, array, object}

is a workaround for the lack of *non-transient constexpr allocation*

Project Ideas

XML parser

JSON parser

ORM https://github.com/VolgaLabs/vORM

Reflection in other libraries

Reflection in other libraries

https://github.com/skypjack/meta - manual registration

Boost.PFR - tuple-like indexing for aggregate

Boost.Hana - macros

https://github.com/silverqx/TinyORM - “runtime reflection” + type erasure

https://thrift.apache.org/ - custom compiler for their DSL

skypjack/meta

Seems like you need to register your types with the library

Doesn’t really solve the code duplication issue

Boost.PFR (very limited reflection)

Boost.PFR allows you to index aggregates like tuples, but only up to 200

e.g. `get<0>(person)` instead of `person.name`

```
template <class T>
constexpr auto tie_as_tuple(T& val, size_t_<200>) noexcept {
  auto& [
    a,b,c,d,e,f,g,h,j,k,l,m,n,p,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,
    aa,ab,ac,ad,ae,af,ag,ah,aj,ak,al,am,an,ap,aq,ar,as,at,au,av,aw,ax,ay,az,aA,aB,aC
    ba,bb,bc,bd,be,bf,bg,bh,bj,bk,bl,bm,bn,bp,bq,br,bs,bt,bu,bv,bw,bx,by,bz,bA,bB,bC
    ca,cb,cc,cd,ce,cf,cg,ch,cj,ck,cl,cm,cn,cp,cq,cr,cs,ct,cu,cv,cw,cx,cy,cz,cA,cB,cC
    da,db,dc,dd,de,df,dg,dh,dj,dk,dl,dm
  ] = const_cast<std::remove_cv_t<T>&>(val); // ====================> Boost.PFR: Use

  return ::boost::pfr::detail::make_tuple_of_references(
    args: detail::workaround_cast<T, decltype(a)>(&arg: a),detail::workaround_cast<T,
    detail::workaround_cast<T, decltype(d)>(&arg: d),detail::workaround_cast<T, declt
    detail::workaround_cast<T, decltype(g)>(&arg: g),detail::workaround_cast<T, declt
    detail::workaround_cast<T, decltype(k)>(&arg: k),detail::workaround_cast<T, declt
    detail::workaround_cast<T, decltype(n)>(&arg: n),detail::workaround_cast<T, declt
    detail::workaround_cast<T, decltype(r)>(&arg: r),detail::workaround_cast<T, declt
```

Boost.Hana

Introspection done via macros

```
// 1. Give introspection capabilities to 'Person'
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (int, age)
  );
};

// 2. Write a generic serializer (bear with std::ostream for the example)
auto serialize = [](std::ostream& os, auto const& object) {
 hana::for_each(hana::members(object), [&](auto member) {
    os << member << std::endl;
  });
};
```

TinyORM

```
post->getAttribute<qint64>("id");
```

```
template<typename Derived, AllRelationsConcept ...AllRelations>
QVariant
HasAttributes<Derived, AllRelations...>::getAttribute(const QString &key) const
{
    if (key.isEmpty() || key.isNull())
        return {};

    /* If the attribute exists in the attribute hash or has a cast we will
       get the attribute's value. Otherwise, we will return invalid QVariant.
       Also, don't user the hasCast(key) check here because there is always primary
       key cast and it would return null or invalid QVariant. */
    if (m_attributesHash.contains(x: key) || basemodel().getUserCasts().contains(key))
        return getAttributeValue(key);

    // FUTURE add getRelationValue() overload without Related template argument, after
    // NOTE api different silverqx
    return {};
      return $this->getRelationValue($key);
}
```

Apache Thrift

- **Build and Install the Apache Thrift compiler**
  You will then need to build the Apache Thrift compiler and install it. See the installing Thrift guide for any help with this step.
- **Writing a .thrift file**
  After the Thrift compiler is installed you will need to create a thrift file. This file is an interface definition made up of thrift types and Services. The services you define in this file are implemented by the server and are called by any clients. The Thrift compiler is used to generate your Thrift File into source code which is used by the different client libraries and the server you write. To generate the source from a thrift file run

  ```
  thrift --gen <language> <Thrift filename>
  ```

  The sample tutorial.thrift file used for all the client and server tutorials can be found here.

Reflection in other languages

Rust

https://effective-rust.com/reflection.html “Rust does not have static reflection”

However, Rust has procedural macros, which can read the AST

(Opinionated) To me, this *is* a form of **syntactic** static reflection

Which is in contrast to other languages’ **semantic** static reflection

Java

https://docs.oracle.com/javase/tutorial/reflect/ introspection since JDK 1.1 (1997)

https://openjdk.org/jeps/416 3 possible mechanisms for implementation

https://openjdk.org/projects/babylon/ code reflection

Python & JS

Interpreted languages that have runtime reflection

You can basically do anything

```
*--slides.end();
```

Misc. cpp lessons I learned

- Turns out `<iostream>` depends on LIBCXX_ENABLE_LOCALIZATION
-