New reflection metafunctions - is_structural_type (US NB comment 49) and is_destructurable_type

Document #:	P3856R3 [Latest] [Status]
Date:	2025-11-01
Project:	Programming Language C++
Audience:	SG7/LEWG

Reply-to:	Jagrut Dave (jdave12@bloomberg.net) Alisdair Meredith (ameredith1@bloomberg.net)

Contents

1. Revision History

From version 2

- Corrected version number

From version 1

- Added metafunction is_structural_type addressing US NB comment 49

From version 0

- Corrected paper number, added observations from the proof of concept (section 3.3)

2. Introduction

Several parts of the standard and library refer to structural types, including library mandates that types be structural, yet there is no way to query whether a type is structural. Library mandates clauses mean that library implementers must somehow have this functionality, but it is simply not exposed to users. We propose a new reflection metafunction - *is_structural_type(info, access_context)* as a solution, which would return *true* when *info* is the reflection of a structural type, and *false* otherwise.

We also propose another reflection metafunction - *is_destructurable_type(info, access_context)*. Complementary to the metafunction for structural types, this function would return *true* for a type that can participate in structured bindings, and going forward, could be decomposed into a pack ([P1061](P1061)).

Both these functions are proposed to be added to the *<meta>* header. We provide sample implementations using other reflection metafunctions from P2996 to demonstrate how far we can go with the existing batch of reflection type queries included in C++26. In addition, we compare the approaches of using traditional type traits vs. using reflection metafunctions to query type attributes.

3. Structural and destructurable types

3.1 is_structural_type

As per [temp.param] (13.2), a structural type is (a) a scalar type, (b) an lvalue reference type, or (c) a literal class type whose direct bases and non-static data members are all public and non-mutable, and whose types are themselves structural (or arrays thereof). Captureless lambdas are also considered structural, per [expr.prim.lambda.closure] (7.5.6.2).

3.1.1 Motivating examples

The *is_structural_type check* is valuable wherever values (including small user-defined structs) have to be treated as compile-time constants, since NTTPs must be structural types. E.g., better diagnostics can be produced for *template<auto>* instantiations, and APIs can be cleanly separated.

```
C/C++
template<auto V>
requires is_structural_type_v<decltype(V)>
struct const_wrapper { static constexpr auto value = V; };

template<auto V>
requires (!is_structural_type_v<decltype(V)>)
struct const_wrapper<V>; // = delete; // or static_assert with a helpful note

template<auto V>
requires is_structural_type_v<decltype(V)>
void register_token() { /* Register type V */ }

template<class T>
void register_token(type_tag<T>) { /* Register type_tag<V> value */ }
```

A single logging API could be used for compile and run time logging.

```
C/C++
template<auto Tag>
requires is_structural_type_v<decltype(Tag)>
inline void log_event() { /* Tag is known to be NTTP */ }

// Fall back, run-time.
inline void log_event(std::string_view tag);
```

3.1.2 Proof of concept implementation

The code below shows a possible implementation of *is_structural_type,* using Bloomberg's Clang fork.

C/C++

```
#include <experimental/meta>


// Forward-declare so we can recurse.
consteval bool is_structural_type(std::meta::info,
       std::meta::access_context ctx);

namespace detail {

// Strip top-level cv, but not references (rvalue refs are *not* structural).
consteval std::meta::info canonicalize(std::meta::info t) {
  return remove_cv(dealias(t));
}

// Peel one array extent; P2996 has array transforms in
[meta.reflection.trans.arr].
consteval std::meta::info remove_one_extent(std::meta::info t) {
  return remove_extent(t);
}

consteval bool is_array_of_structural_types(std::meta::info t,
std::meta::access_context ctx) {
  t = canonicalize(t);

  if (!is_array_type(t)) return false;

  // Recursively strip all extents, then check the element type
  std::meta::info elem = t;
  while (is_array_type(elem)) {
    elem = remove_one_extent(elem);
  }
  return is_structural_type(canonicalize(elem), ctx);
}

// Return false if any direct base is not public *or* its type isn't
structural.
consteval bool bases_ok(std::meta::info t, std::meta::access_context ctx) {
  constexpr auto unchecked = std::meta::access_context::unchecked();
  for (std::meta::info rel : bases_of(t, unchecked)) {
    if (!is_public(rel))         return false;  // declared access
    std::meta::info rel_type = type_of(rel);

    // recursive check
    if (!is_structural_type(canonicalize(rel_type), ctx)) return false;
```

```
  }
  return true;
}

// Return false if any nsdm is not public or is mutable, or its type isn't
structural/array-of-structural.
consteval bool data_members_ok(std::meta::info t, std::meta::access_context
ctx) {
  constexpr auto unchecked = std::meta::access_context::unchecked();
  for (std::meta::info m : nonstatic_data_members_of(t, unchecked)) {
    if(!is_accessible(m, ctx)) return false;

    if (!is_public(m))         return false;

    if (is_mutable_member(m))  return false;

    std::meta::info MT = type_of(m);
    if (is_array_type(MT)) {
      if (!is_array_of_structural_types(MT, ctx)) return false;
    } else {
      if (!is_structural_type(canonicalize(MT), ctx)) return false;
    }
  }
  return true;
}


// Forward declaration for mutual recursion.
consteval bool is_literal_type(std::meta::info type,
                               std::meta::access_context ctx);

// Helper: at least one variant member of a union is a non-volatile literal
type.
// (Empty unions are allowed by DR 2598.)
consteval bool union_has_literal_variant(std::meta::info union_type,
                                         std::meta::access_context ctx) {
  auto ms = nonstatic_data_members_of(union_type, ctx);
  if (ms.empty()) return true; // union with 0 variant members
  for (auto m : ms) {

    if(!is_accessible(m, ctx)) return false;

    std::meta::info mt = dealias(type_of(m));
```

```
    if (!is_volatile_type(mt) && is_literal_type(mt, ctx)) {
      return true;
    }
  }
  return false;
}

// Helper: check “all bases and non-static data members are non-volatile
literal types”,
// and enforce the “anonymous union has a literal variant” rule for classes
with anon unions.
consteval bool class_members_are_literal(std::meta::info class_type,
                                         std::meta::access_context ctx =
std::meta::access_context::unchecked()) {
  // Bases
  for (auto b : bases_of(class_type, ctx)) {
    std::meta::info bt = dealias(type_of(b));
    if (is_volatile_type(bt) || !is_literal_type(bt, ctx)) {
      return false;
    }
  }
  // Non-static data members
  for (auto m : nonstatic_data_members_of(class_type, ctx)) {

    if(!is_accessible(m, ctx)) return false;

    std::meta::info mt = dealias(type_of(m));

    // Anonymous union members need a special check (P2996 exposes
has_identifier + union test).
    if (is_union_type(mt) && !has_identifier(m)) {
      if (!union_has_literal_variant(mt, ctx)) {
        return false;
      }
      continue;
    }

    if (is_volatile_type(mt) || !is_literal_type(mt, ctx)) {
      return false;
    }
  }
  return true;
}
```

```
// Probe for “constexpr destructor when default constructible” using
substitution.
template<class T, class = void>
constexpr bool __has_constexpr_dtor_if_default = false;

template<class T>
constexpr bool __has_constexpr_dtor_if_default<
  T,
  // This requires both a constexpr default construction *and* that the
destructor
  // is permitted in constant evaluation, which implies a constexpr destructor.
  decltype( []{
    constexpr T t{}; // ok only if T is literal *and* its dtor is constexpr
    (void)t;
  }(), void() )
> = true;

consteval bool is_literal_type(std::meta::info type,
                               std::meta::access_context ctx /* = unchecked */)
{
  type = dealias(type);

  // Primary cases
  if (is_void_type(type))     return true;
  if (is_reference_type(type))return true;
  if (is_scalar_type(type))   return true;

  // Arrays: literal iff the element type is literal.
  if (is_array_type(type)) {
    return is_literal_type(remove_all_extents(type), ctx);
  }

  // Unions: aggregate union with a literal variant (or empty union) and
non-volatile members.
  if (is_union_type(type)) {
    if (!is_aggregate_type(type)) return false; // matches the “aggregate
union” bullet
    return union_has_literal_variant(type, ctx);
  }

  // Class types: check members/bases first.
  if (is_class_type(type)) {
    if (!class_members_are_literal(type, ctx)) {
      return false;
```

```
    }

    // Destructor condition: C++20+ requires a constexpr destructor (trivial
counts).
    // P2996R13 doesn't expose “is this function constexpr?” for destructors,
    // so we implement a conservative check:
    //  - accept trivially-destructible (always OK);
    //  - otherwise, *attempt* to verify a constexpr destructor in the
default-constructible case
    //    via a substitution probe (__has_constexpr_dtor_if_default).
    if (!is_trivially_destructible_type(type)) {
      bool ok = false;
      // Try the default-constructible constexpr-dtor probe.
      // (If T isn't default-constructible but still has a constexpr dtor, we
can't detect it.)
      ok = extract<bool>(substitute(^^__has_constexpr_dtor_if_default, { type
}));
      if (!ok) return false; // conservative: fail if we cannot prove constexpr
dtor
    }

    // Final disjunct for class types:
    //   * aggregates are fine (no need to find a constexpr ctor explicitly),
    //   * otherwise the standard allows “has at least one constexpr
non-copy/move ctor”
    //     or “lambda type”. P2996R13 doesn’t yet expose “is this constructor
constexpr?”
    //     nor “is this a lambda closure type?”, so be conservative and only
accept aggregates.
    if (is_aggregate_type(type)) {
      return true;
    }

    // Without constructor-constexpr/lambda queries, we can't soundly accept
    // non-aggregate classes here. Be conservative.
    return false;
  }

  // Everything else (functions, etc.) — not literal.
  return false;
}

consteval bool is_literal_class_type(std::meta::info t,
std::meta::access_context ctx) {
```

```
  if (!(is_class_type(t) || is_union_type(t))) return false;

  return is_literal_type(t, ctx);
}
} // namespace detail

consteval bool is_structural_type(std::meta::info t0, std::meta::access_context
ctx) {
  // 1) lvalue reference types are structural (top-level cv is ignored)
  if (is_lvalue_reference_type(t0)) {
    return true;
  }

  // Work with cv-stripped, de-aliased type for the rest
  std::meta::info t = detail::canonicalize(t0);

  // 2) scalar types are structural
  if (is_scalar_type(t)) {
    return true;
  }

  // 3) literal class/union with the "public & non-mutable &
structural-or-array-of-structural" condition
  if (is_class_type(t) || is_union_type(t)) {

    if (!detail::is_literal_class_type(t, ctx)) {
        return false;
    }

    return detail::bases_ok(t, ctx) && detail::data_members_ok(t, ctx);
  }

  // Other kinds (function types, arrays as such, rvalue refs, etc.) are not
structural
  return false;
}

constexpr auto ac = std::meta::access_context::unchecked();

struct S { public: int x; };
struct Bad1 { private: int x; };
struct Bad2 { public: mutable int x; };
struct A { public: int a[2][3]; };
struct B : public S {};           // OK: public base of structural type
```

```
struct C : private S {};          // not structural: private base

static_assert( is_structural_type(^^int, ac));
static_assert( is_structural_type(^^int&, ac));
static_assert(!is_structural_type(^^int&&, ac));// rvalue ref is not structural
static_assert( is_structural_type(^^S, ac));
static_assert(!is_structural_type(^^Bad1, ac));
static_assert(!is_structural_type(^^Bad2, ac));
static_assert( is_structural_type(^^A, ac));
static_assert( is_structural_type(^^B, ac));
static_assert(!is_structural_type(^^C, ac));

int main() {

}
```

Godbolt link

3.2 is_destructurable_type

Today, the “decomposability” of a type can be approximated by checking the tuple protocol (presence of *std::tuple_size/std::tuple_element* and *get<I>*), which covers tuple-like types but does not reliably detect aggregate-member decomposition for user-defined aggregates. *decomposable()* from P0931 was meant to close this gap in the pre-Reflection era. *std::is_aggregate<T>* in *<type_traits>* can identify an aggregate type but does not determine whether the type can participate in decomposition declarations. Similarly, the metafunction *is_structured_binding(info r)* can determine whether a variable or pack initialization was a structured binding initialization, but not whether a type can participate in such an initialization to begin with. For example, private/protected members or base classes of the type could prevent it from participating in decomposition declarations. Such a complex trait could be implemented using Reflection metafunctions.

3.2.1 Motivating examples

Consider a custom hash function used to hash standard or user-defined types. For destructurable types, it returns the sum of the sizes of the first two data members, while for other types, it calls *sizeof()* on the type. If an invalid *meta::info* is passed, the behavior is implementation-defined, as with other Reflection-based type queries.

```C/C++
template<class INPUT_TYPE, bool is_destructurable =
is_destructurable_type(^^INPUT_TYPE, ac)>

constexpr size_t custom_hash(INPUT_TYPE input, std::meta::access_context ac) {
  if constexpr (is_destructurable) {
    const auto& [first_member, second_member, ...rest] = input;
    return sizeof(first_member) + sizeof(second_member);
  }
  else {
    return sizeof(INPUT_TYPE);
  }
}
```

[Godbolt link](#)

Another application could be in code that relies on the *is_aggregate* type trait or equivalent compiler intrinsic to determine if a type could be decomposed, such as [Boost.PFR](#). *is_destructurable_type* would be a better fit, since the programmer would like to know if the type can be decomposed using structured bindings rules, which include ADL-available functions, rather than if it can be aggregate-initialized, along with some other checks, which taken together are a subset of decomposability.

3.2.2 Proof of concept implementation

The code below shows a possible implementation of *is_destructurable_type,* using Bloomberg's Clang fork.

```C/C++
#include <experimental/meta>

template <typename T, class = void>
constexpr bool has_tuple_size = false;

template <typename T>
constexpr bool has_tuple_size<T, decltype((void)std::tuple_size<T>::value)> =
                       is_integral_type(type_of(substitute(^^std::tuple_size_v,
                                                            {
                                                                ^^T}))) ||
                       is_enum_type(type_of(substitute(^^std::tuple_size_v,
                                                        {
                                                            ^^T})));
```

```
template <typename T, unsigned K, class = void>
constexpr bool has_member_get_k = false;

template <typename T, unsigned K>
constexpr bool
    has_member_get_k<T,
                     K,
                     decltype((void)std::declval<T>().template get<K>())> =
                                                                         true;

template <typename T, unsigned K, class = void>
constexpr bool has_adl_get_k = false;

template <typename T, unsigned K>
constexpr bool has_adl_get_k<T, K, decltype((void)get<K>(std::declval<T>()))> =
                                                                         true;
// is_destructurable_type
consteval bool is_destructurable_type(std::meta::info           r,
                                      std::meta::access_context ctx)
{
    if (!is_class_type(r))
        return false;

    // Case 1 - array check
    auto is_array_case = [](std::meta::info r) {
        return is_array_type(r);
    };

    // Case 2 - tuple-like check
    auto is_tuple_case = [](std::meta::info r) {
        if (!extract<bool>(substitute(^^has_tuple_size,
                                      {
                                          r}))) {
            return false;
      }

        std::size_t sz = extract<std::size_t>(substitute(^^std::tuple_size_v,
                                                         {
                                                             r}));
        bool has_member_get = false;

        for (unsigned idx = 0; idx < sz; ++idx) {
            bool check = extract<bool>(substitute(
                                          ^^has_member_get_k,
```

```
                           {
                                r,
                                std::meta::reflect_constant(idx)
                           }
                      )
                 );

            if (has_member_get && !check) {
              return false;
            }

           has_member_get = check;
       }

       if (has_member_get) {
           return true;
        }

       for (unsigned idx = 0; idx < sz; ++idx) {
           if (!extract<bool>(substitute(^^has_adl_get_k,
                                  {
                                      r,
                                      std::meta::reflect_constant(idx)
                                   }))) {
               return false;
            }
       }

      return true;
  };

  // Case 3 - check for bases, inheritance, and access controls.
  auto is_data_member_case = [](std::meta::info           r,
                                std::meta::access_context ctx) {
       constexpr auto unchecked_ctx = std::meta::access_context::unchecked();
       auto nsdms = nonstatic_data_members_of(r, unchecked_ctx);
       {
           std::meta::info class_with_members = nsdms.size() > 0
                                                ? r
                                                : std::meta::info{};
           for (auto base : bases_of(r, unchecked_ctx)) {
               auto base_nsdms = nonstatic_data_members_of(type_of(base),
                                                           unchecked_ctx);
               if (nsdms.size() > 0) {
```

```
                    if (class_with_members != std::meta::info{} &&
                        class_with_members != base) {
                        return false;
                    }
                    class_with_members = base;
                }
            }
        }

        for (auto mem : nsdms) {
            if (!is_accessible(mem, ctx)) {
                return false;
            }
            if (is_union_type(type_of(mem)) && !has_identifier(mem)) {
                return false;
            }
        }

        return true;
    };

    return is_array_case(r) || is_tuple_case(r) ||
           is_data_member_case(r, ctx);
}

struct S {
    int a;
    int b;
    int c;
};

struct C1 : S {
};

struct C2 : S {
    int d;
};

constexpr auto ac = std::meta::access_context::unchecked();
static_assert(!is_destructurable_type(^^int, ac));
static_assert(is_destructurable_type(^^S, ac));
static_assert(is_destructurable_type(^^C1, ac));
static_assert(!is_destructurable_type(^^C2, ac));
static_assert(is_destructurable_type(^^std::tuple<int, bool>, ac));
```

[Godbolt link]

3.3 Observations from the sample implementations

- The unchecked access context - *std::meta::access_context::unchecked()* - is used to obtain bases of a type, and then their non-static data members. Then, the caller's access context is used to verify access to those data members. The point in the program where the above functions are evaluated would determine the caller's access context, and thus, whether the type is classified as structural or not, or destructurable or not. There isn't a facility similar to access context in traditional type traits APIs.
- The implementation could be improved if metafunctions such as "is this a literal type", "is this a lambda closure type" or "is this function constexpr" were available (*std::is_literal_type<T>* was removed in C++20).
- The proof of concept implementations ignore unsuitable inputs. If the input argument isn't a reflection of a type, reflection metafunctions are supposed to throw an exception, as per the current specification. However, NB Comment US 129-191 proposes returning *false* instead. If the NB comment were accepted, the implementation of the new reflection metafunctions would change only for the case when the input argument is the reflection of a non-type. The core logic for the happy case, when the input arguments are reflections of types, remains unchanged.

4. Type traits or metafunctions?

P2996 introduced several metafunctions (*consteval* functions that operate on *meta::info*) in *<meta>* that mirror existing type traits, such as *is_const_type(info), is_volatile_type(info),* and *is_trivially_copyable_type(info)*. Type traits are typically implemented via templates (SFINAE) or compiler intrinsics, or a combination of both techniques. Metafunctions offer several benefits over traditional type traits when used to query the attributes of a type:

- Simple and fixed interface that accepts a single, light-weight *meta::info* type parameter.
- Avoid template instantiation and SFINAE-related memory bloat in most cases.
- Provide a wide range of building blocks, as they can not only query types, but also values, and non-type entities like data members, functions, parameters, namespaces, etc. E.g., *std::meta::is_final(info type)* isn't limited to "final class types"; it can also ask if a member function is final. Such a query is not expressible with type traits via a single trait.
- Improved error handling by means of *meta::exception* that includes an error message, source and line number, thanks to P3560.
- Access control checks while querying type information, thanks to P3547. Type queries could return different answers depending on the program point from where they're evaluated.

- Improved maintainability as the code is easier to read vs. SFINAE-based template waterfall logic.
- Faster execution than a template-based implementation, though not as fast as using compiler intrinsics.

Given that we now have two similar facilities in the standard library for inspecting type attributes, how should programmers approach them? Should type traits and reflection metafunctions evolve independently? Should one be implemented in terms of the other? More research and discussion needs to be carried out, though our preliminary thinking is that type traits and metafunctions should implement the abstract library API specification independently. They are different paradigms for metaprogramming, with different side effects. Metafunctions should not use library trait templates in their implementation, as that may confuse a programmer who expects to avoid template instantiations and encounter a different set of error logs. Conversely, type traits should not use reflection under the covers as that may surprise programmers who expect to see and handle the side effects of template metaprogramming.

5. Proposed Library wording

Add the new metafunctions to <meta> in [meta.reflection], at the end of the list of unary type-property queries.

```None
21.4 Reflection [meta.reflection]

21.4.1 Header <meta> synopsis [meta.syn]
…

// associated with [meta.unary.prop], type properties
consteval bool is_scoped_enum_type(info type);
consteval bool is_structural_type(info type,access_context ctx);
consteval bool is_destructurable_type(info type,access_context
ctx);

template<reflection_range R = initializer_list<info>>
    consteval bool is_constructible_type(info type, R&&
type_args);
...
```

Add the new metafunction in 21.4.17 Reflection type traits [meta.reflection.traits]

```
None
21.4.17 Reflection type traits[meta.reflection.traits]
…
// associated with [meta.unary.prop], type properties
…
consteval bool is_scoped_enum_type(info type);
consteval bool is_structural_type(info type,access_context ctx);
consteval bool is_destructurable_type(info type,access_context
ctx);

template<reflection_range R = initializer_list<info>>
  consteval bool is_constructible_type(info type, R&& type_args);
…
```

6. Conclusion

Our experience in implementing two new reflection metafunctions shows how reflection type queries can be implemented using other reflection metafunctions only (no intrinsics or SFINAE), and how they differ from type traits (caller's access context/program point can be incorporated in the decision). It also shows the need for more "lower level" metafunctions not included in P2996, such as *is_constexpr(info)*, *is_literal_type(info)* and *is_lambda_closure_type(info)*. The approach of using reflection metafunctions to query type attributes appears promising, and likely to yield richer APIs than possible with prior techniques.

7. Acknowledgements

We would like to thank Dan Katz for the sample implementation of *is_destructurable_type* and Andrei Zissu for his review of the paper.

8. References

[P2996R13] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2025-01-13. Reflection for C++26.
https://wg21.link/p2996r9

[P3547R1] Dan Katz, Ville Voutilainen. 2025-02-09. Modeling Access Control With Reflection.
https://wg21.link/p3547r1

[P1061R10] Barry Revzin, Jonathan Wakely. 2024-11-24. Structured Bindings can introduce a Pack.
https://wg21.link/p1061r10

[P0144R2] Herb Sutter, Bjarne Stroustrup, Gabriel Dos Reis. 2016-03-16. Structured bindings.
https://wg21.link/p0144r2

[CppCon25Talk] Andrei Zissu. 2025-09. Reflection and Metaprogramming in Modern C++.
https://www.youtube.com/watch?v=EK74rV1M7uc

[N5028] 2025-10-2. SoV and Collated Comments - ISO_IEC CD 14882.
https://wg21.link/N5028