New Reflection metafunction - is_destructurable_type

Document #:	DXXXXR0 [Latest] [Status]
Date:	2025-08-15
Project:	Programming Language C++
Audience:	LEWG

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

Contents

1. Revision History

(TBD)

2. What does destructurable mean?

We propose a new Reflection-based metafunction *is_destructurable_type(info, access_context)* in <*meta*> that returns a *bool*. It returns *true* for a type that can participate in structured bindings,

and going forward, could be decomposed into a pack (P1061). In addition, we compare traditional type traits vs. Reflection metafunctions to query type attributes.

3. 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.

3.1 Motivating example

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

3.2 Sample implementation

The code below shows a possible implementation of *is_destructurable_type,* using Bloomberg's Clang fork. The access-context parameter is necessary because the implementation needs to check access to the non-static data members of the type's base classes.

```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) {
    constexpr auto unchecked_ctx = std::meta::access_context::unchecked();
    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

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)*. 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.

We think that type traits and metafunctions should implement type queries 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. Rather than trying to make type traits compatible with metafunctions or vice versa, both should be allowed to evolve independently.

Metafunctions offer a simpler interface, reduced memory footprint, and reduced reliance on compiler intrinsics. Thus, type queries in C++29 and beyond should be implemented as metafunctions if possible, rather than as type traits.

5. Proposed Library wording

Add the new metafunction 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_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_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. Acknowledgements

We would like to thank Dan Katz for the sample implementation of *is_destructurable_type* shown above.

7. 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