P3707R0 – A std::is_always_exhaustive trait

Document number: P3707R0

Date: 2025-05-18

Authors:

- Patrice Roy: patricer@gmail.com
- Gregoire Angerand: gregoire.angerand@gmail.com

Reply to: patricer@gmail.com

Target audience: LEWG, SG14

Introduction

C++17 introduced a trait named `std::has_unique_object_representations<T>` that lets code ensure that a type is uniquely represented by its values. Informally, this can be seen as a way to verify that objects of that type are exempt of padding bits, but the requirements are stricter than this: `T` has to be *TriviallyCopyable* and any two objects of type `T` with the same value have to have the same object representation.

This definition disqualifies types with data members of floating point types because NaNs do not have a unique object representation. For this reason, this trait cannot be used in practice to ensure an object that has floating point data members is exempt of padding bits.

Sadly, this is a limiting factor for programs that can ensure their floating point numbers are not NaNs or that, if they are NaNs, they have the same canonical representation. There are use cases for programs that want to statically ensure that some types can be used on a GPU without fear of padding bits having indeterminate values and complicating the bitwise comparison of otherwise identical objects.

This proposal is for a trait that provides this missing facility. This trait is tentatively named **`std::is_always_exhaustive<T>`**.

Motivation for the proposed name

The intent is to provide a type trait which describes types that have no “padding”. Discussions with Mark Hoemmen indicate that `mdspan` layout uses `is_always_exhaustive()`. Quoting the esteemed Mr. Hoemmen from private conversation:

> « "Exhaustive" means that the layout mapping is surjective, that is, that every index in the mapping's codomain has a corresponding multidimensional index in the mapping's domain. To say that a type has no "padding" means that every byte of an object of that type corresponds to some member of the type. That is, the mapping from member name to bytes of the object is surjective. »

Thus, we tentatively propose **`std::is_always_exhaustive<T>`** as name for this trait, and will adjust in time if that name is found to be problematic.

Motivation for the proposed traits

Consider the following `Int16` and `Float16` types:

```
#include <type_traits>
struct Int16 { // note: 16 bytes, not bits
   int i0, i1, i2, i3;
};
static_assert(std::is_trivially_copyable_v<Int16>);
static_assert(
   sizeof(Int16) == sizeof(std::declval<Int16>().i0) +
                    sizeof(std::declval<Int16>().i1) +
                    sizeof(std::declval<Int16>().i2) +
                    sizeof(std::declval<Int16>().i3)
);
struct Float16 { // note: 16 bytes, not bits
   float f0, f1, f2, f3;
};
static_assert(std::is_trivially_copyable_v<Float16>);
static_assert(
   sizeof(Float16) == sizeof(std::declval<Float16>().f0) +
                      sizeof(std::declval<Float16>().f1) +
                      sizeof(std::declval<Float16>().f2) +
                      sizeof(std::declval<Float16>().f3)
);
struct Chunk {
   alignas(16) Int16 i16;
   alignas(16) Float16 f16;
};
int main() {
   static_assert(sizeof(Int16)==16);
   static_assert(sizeof(Float16)==16);
   // Ok
   static_assert(
```

```
        std::has_unique_object_representations_v<Int16>
    );
    // Sad
    static_assert(
        !std::has_unique_object_representations_v<Float16>
    );
}
```

Supposing that the various `static_assert` expressions in the above example succeed, we know that:

- An `Int16` object is trivially copyable.
- An `Int16` object's representation is entirely made of its data members and there are no padding bits between these members.
- A `Float16` object is trivially copyable
- A `Float16` object's representation is entirely made of its data members and there are no padding bits between these members.

An application that seeks to use an `Int16` object in a situation that requires bitwise copies and bitwise comparisons could do so.

An application that seeks to use a `Float16` object in a situation that requires bitwise copies and bitwise comparisons could do so as long as its data members are not NaNs or, if they are NaNs, that they have the same canonical representation.

For type `Int16`, we could have inferred that information through the use of the `std::has_unique_object_representations_v<Int16>` trait. We could not have done so with `std::has_unique_object_representations_v<Float16>`.

Intended usage

Users of trait **`std::is_always_exhaustive<T>`** are looking for a compile-time guarantee that type `T` is exempt of padding.

If `T` has data members of floating point types, these users can guarantee that these data members are either (a) not NaN or (b) are NaN that share a canonical representation such that for two objects `t0`, `t1` of type `T` and some data member `T::m`, the underlying representation of `t0.m` and `t1.m` would compare equal.

Possible approach

The text for `std::has_unique_object_representations<T>` found in [meta.unary.prop] p10 states (emphasis mine):

> « The predicate condition for a template specialization has_unique_object_representations<T> shall be satisfied if and only if
>
> (10.1) T is trivially copyable, and

(10.2) any two objects of type T with the same value have the same object representation, where

(10.2.1) two objects of array or non-union class type are considered to have the same value if their respective sequences of direct subobjects have the same values, and

(10.2.2) two objects of union type are considered to have the same value if they have the same active member and the corresponding members have the same value.

The set of scalar types for which this condition holds is implementation-defined.

[Note 9: If a type has padding bits, the condition does not hold; otherwise, the condition holds true for integral types. — end note] »

This text mostly seems to meet the expectations for `std::is_always_exhaustive<T>`. A non-exhaustive (!) list of approaches to adjust the wording would include:

- adding a phrase to the boldface text to accept floating point scalars (with a caveat that undefined behavior will ensue if at least one such scalar is a NaN, unless said NaN representations compare equal),
- adding (10.2.3) to provide a provision for floating point scalars (which could be preferable if the text required for this provision is more complicated), or
- adding a note such as "[*Note:* If NaN is avoided, T can be a floating-point type. —*end note*]" as can be found in [alg.clamp].

Prior art

We have a very similar trait in `std::has_unique_object_representations<T>`. It just slightly misses the mark for some real-life applications. Hopefully, adding a trait such as `std::is_always_exhaustive<T>` would be a useful complement.

FAQ

Question 00: have you considered alternative spellings?

Answer: of course, but let's work with this for now. If this name seems unsatisfying, we will examine alternatives in more detail.

Question 01: the potential of adding undefined behavior through a trait is a source of discomfort. Is this necessary?

Answer: well, the occurrence of a NaN in an object of some floating point type is a possibility. Users of this trait would have to be aware of this obligation of avoid NaN or ensure that the underlying representations can be compared. In practice, what can be expected is that two objects of type `T` for which `std::is_always_exhaustive<T>::value` is `true` will be used in ways where their representation will be compared bitwise, and as such failure to conform to the comparability requirement of floating point data members will lead to such comparisons producing erroneous results. It's unclear to us whether erroneous behavior is an option here (we are under the impression that it isn't).

Question 02: would it be sufficient to simply change the behavior of `std::has_unique_object_representations<T>`?

Answer: we fear this could lead to bad surprises for existing usages of that trait. This explains why we are proposing a new trait instead.