Document: P0466R2
Date: 2018-03-29
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Evolution; LEWG; LWG

Layout-compatibility and Pointer-interconvertibility Traits

Lisa Lippincott

Abstract

Over dinner at CppCon, Marshall Clow and I discussed a bit of code that relied on two types being layout-compatible. As it happened, the types weren't layout-compatible after all. I opined that there should be a way to statically assert layout-compatibility, so that the error would be caught at compile time, rather than dinner time. Marshall replied, "Write a proposal." This is that proposal.

In addition to a test for layout-compatibility, I propose tests for correspondence in the initial common sequence of two types, and for situations in which objects are pointer-interconvertible.

Changes from r1 to r2: These changes are based on feedback in the second Core discussion at Jacksonville, 2018-03-16. Each of these changves is more directly relative to the draft presented there.

- Adding wording to insist on complete types as arguments to the traits.
- Correcting the order of template parameters in the synopsis of `is_corresponding_member`.
- When describing `is_pointer_interconvertible_with_class`, writing of each object in the singular. On my own initiative, likewise changing `is_pointer_interconvertible_base_of`.
- Changing "happily fails" to "fails, as desired."

These changes are based on feedback in the first Core discussion at Jacksonville, 2018-03-13.

- Rewriting the abstract and much of the front matter to remove incorrect blather about `reinterpret_cast`. Instead, I've tried to restrict the text to mostly true statements.
- Restoring the constexpr functions from revision 0, as core-preferred alternatives to the traits in revision 1. The traits wording is kept and updated as an alternative.
- Renaming pointer-interconvertibility tests to express their function, rather than their mechanism, and changing their definitions to refer to core definitions, rather than mimic core definitions.

  `is_initial_base_of` → `is_pointer_interconvertible_base_of`
  `is_initial_member` → `is_pointer_interconvertible_with_class`

  This renaming more directly expresses the intent of these facilities, simplifies their wording, and allows them to track future changes in core wording.

  More generally, it is better to say what one means, rather than say what means what one means.
- Using phrases, rather than declarator syntax, when naming pointer-to-member types.
- Rebasing on draft n4713 of the standard.
- Correction of various typographic errors.

These changes are on my own initiative:

- Moving the enclosing-class template parameters of `is_corresponding_member` to the front of the parameter list, for use with explicit template arguments.

- Removing the increasingly-pointless requirement that the functions be ill-formed when applied to pointers to member functions. They can return false instead.
- Consolidating the notes about pointer to member literals.
- Adding _v definitions to the synopses where needed.

Changes from r0 to r1: These changes are based on the Library Evolution discussion at Kona in 2017. First, renaming the plural traits:

`are_layout_compatible` → `is_layout_compatible`
`are_common_members` → `is_corresponding_member`

Second, changing `is_initial_member` and `is_corresponding_member` from constexpr functions to ordinary traits using `template <auto>`. My thanks go to Louis Dionne for the sample implementation code.

On my own initiative, I have added a discussion and notes on the dangers of deducing the containing type from a member pointer constant.

1 Introduction

Currently, a program may rely on layout-compatibility, but cannot assert that the layout-compatibility it relies upon pertains. Even when a programmer carefully verifies layout-compatibility, a future change to the types involved may break the compatibility, silently introducing a bug.

A compiler, having full information about the types, can easily check layout-compatibility. But the compiler currently has no way to determine which types need to be layout-compatible. This gap can be bridged straightforwardly with a type trait expressing the layout-compatibility relationship:

```
template <class T, class U> struct is_layout_compatible;
```

Using this trait, a function may statically assert the layout-compatibility it relies upon.

Delving deeper into the problem, I found another situation where a programmer might rely on a fact about the type system that can't be asserted: the pointer-interconvertiblity of an object and an initial base or member subobject. A simple type trait handles the base subobject case:

```
template <class Base, class Derived>
struct is_pointer_interconvertible_base_of;
```

The initial member subobject case turns out to be trickier. The test should take a member pointer as a parameter:

```
template <class S, class M, M S::*m>
struct is_pointer_interconvertible_with_class;
```

That works, but with three template parameters, it's really cumbersome. In use, the first two parameters are redundant — the type of `m` determines `S` and `M`. But, because this is a class template, the earlier parameters can't be inferred. A function template is easier to use:

```
template <class S, class M>
constexpr bool
is_pointer_interconvertible_with_class( M S::*m ) noexcept;
```

The use of this function is a little more broad: it can be called in a non-`constexpr` context. An alternative formulation retains the traits syntax, at the expense of this breadth:

```
template <auto m> struct is_pointer_interconvertible_with_class;
```

Such a trait can be implemented by forwarding `decltype(m)`.

A similar situation can occur with layout-compatibility: a programmer may rely on particular members of layout-compatible types overlaying each other. More generally, the overlap of the common initial sequence of two types (12.2 [class.mem]) can only be relied upon if the programmer is sure that particular members correspond. So I'm proposing a second function for testing correspondence in the common initial sequence:

```
template <class S1, class S2, class M1, class M2>
constexpr bool
is_corresponding_member( M1 S1::*m1, M2 S2::*m2 ) noexcept;
```

As above, an alternative would be to stick to traits:

```
template <auto m1, auto m2> struct is_corresponding_member;
```

Note: There is a danger in deducing the type of the containing class from the type of a pointer-to-member literal. Consider the following example:

```
struct A { int a; };
struct B { int b; };
struct C: public A, public B {};

static_assert( is_pointer_interconvertible_with_class( &C::b ) );
   // Succeeds because, despite its appearance, &C::b has type
   // "pointer to member of B of type int."
static_assert( is_pointer_interconvertible_with_class<C>( &C::b ) );
   // Forces the use of class C, and happily fails.

static_assert( is_corresponding_member( &C::a, &C::b ) );
   // Succeeds because, despite appearances, &C::a and &C::b have types
   // "pointer to member of A of type int" and
   // "pointer to member of B of type int," respectively.
static_assert( is_corresponding_member<C,C>( &C::a, &C::b ) );
   // Forces the use of class C, and happily fails.
```

The awkwardness of the deduced type of pointer-to-member constants was discussed in core language issue 203; no action was taken for fear of breaking existing code.

2 is_layout_compatible

Add to table 40 in 23.15.6 [meta.rel]:

Template	Condition	Comments
`template <class T, class U> struct is_layout_compatible;`	`T` and `U` are layout-compatible (6.7 [basic.types])	`T` and `U` shall be complete types.

Add to 23.15.2 [meta.type.synop], in the section corresponding to 23.15.6 [meta.rel]:

```
template <class T, class U> struct is_layout_compatible;
template<class T, class U>
  inline constexpr bool is_layout_compatible_v
    = is_layout_compatible<T,U>::value;
```

3 is_pointer_interconvertible_base_of

Add to table 44 in 23.15.6 [meta.rel]:

Template	Condition	Comments
`template <class Base, class Derived> struct is_pointer_interconvertible_base_of;`	`Derived` is unambiguously derived from `Base`, and each object of type `Derived` is pointer-interconvertible (6.7.2 [basic.compound]) with its `Base` subobject.	`Base` and `Derived` shall be complete types.

I note here that it may be possible to relax the requirement that `Base` be complete.

Add to 23.15.2 [meta.type.synop], in the section corresponding to 23.15.6 [meta.rel]:

```
template <class Base, class Derived>
struct is_pointer_interconvertible_base_of;
template<class Base, class Derived>
  inline constexpr bool is_pointer_interconvertible_base_of_v
    = is_pointer_interconvertible_base_of<Base,Derived>::value;
```

4 is_pointer_interconvertible_with_class

This pretty clearly belongs in `<type_traits>` (23.15 [meta]), but I don't see a clear choice of subsection to put it in. Perhaps it goes in 23.15.6 [meta.rel], or perhaps a new subsection, "Member relationships" is appropriate.

Wherever it fits, here is some text to add:

```
template <class S, class M>
constexpr bool
is_pointer_interconvertible_with_class( M S::*m ) noexcept;
```

Requires: `S` shall be a complete type.

Returns: `true` if and only if each object `s` of type `S` is pointer-interconvertible (6.7.2 [basic.compound]) with its subobject `s.*m`.

Add to 23.15.2 [meta.type.synop], in the corresponding section:

```
template <class S, class M>
constexpr bool
is_pointer_interconvertible_with_class( M S::*m ) noexcept;
```

5 is_corresponding_member

Add this text to the same subsection as `is_pointer_interconvertible_with_class`:

```
template <class S1, class S2, class M1, class M2>
constexpr bool
is_corresponding_member( M1 S1::*m1, M2 S2::*m2 ) noexcept;
```

Requires: `S1` and `S2` shall be complete types.

Returns: `true` if and only if `m1` and `m2` point to corresponding members of the common initial sequence (12.2 [class.mem]) of `S1` and `S2`.

Add to 23.15.2 [meta.type.synop], in the corresponding section:

```
template <class S1, class S2, class M1, class M2>
constexpr bool
is_corresponding_member( M1 S1::*m1, M2 S2::*m2 ) noexcept;
```

6 Note about pointer to member literals

To the same section as the functions above, add a note:

[*Note:* The type of a pointer-to-member literal is not always as it appears, and this may lead to surprising results when using these functions in conjunction with inheritance in classes that are not standard-layout. Consider the following example:

```
struct A { int a; };
struct B { int b; };
struct C: public A, public B {};

static_assert( is_pointer_interconvertible_with_class( &C::b ) );
   // Succeeds because, despite its appearance, &C::b has type
   // "pointer to member of B of type int."
static_assert( is_pointer_interconvertible_with_class<C>( &C::b ) );
   // Forces the use of class C, and fails, as desired.

static_assert( is_corresponding_member( &C::a, &C::b ) );
   // Succeeds because, despite appearances, &C::a and &C::b have types
   // "pointer to member of A of type int" and
   // "pointer to member of B of type int," respectively.
static_assert( is_corresponding_member<C,C>( &C::a, &C::b ) );
   // Forces the use of class C, and fails, as desired.
```

—*end note*]

7 Alternative wording as traits

Instead of the above wording, `is_pointer_interconvertible_with_class` and `is_corresponding_member` can be provided as traits. I favor the function approach above, largely because it allows the enclosing classes to be easily specified as explicit template arguments. This wording replaces the wording in sections 4, 5, and 6 above.

7.1 General wording for traits of non-type parameters

First, it is necessary to introduce general wording for traits of non-type parameters. Rather than duplicate the already-duplicate requirements of UnaryTypeTrait and BinaryTypeTrait, I introduce the common notion of IntegralTrait.

Modify 23.15.1 [meta.rqmts]:

An IntegralTrait describes a property of or relationship between template parameters. It shall be a class template whose specializations are DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template `integral_constant` (23.15.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property or relationship being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the IntegralTrait.

A UnaryTypeTrait is an IntegralTrait with one primary type parameter. A BinaryTypeTrait is an IntegralTrait with two primary type parameters. In each case, other parameters of lesser importance may be present.

A UnaryTypeTrait describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template `integral_constant` (23.15.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the UnaryTypeTrait.

A BinaryTypeTrait describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template `integral_constant` (23.15.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the BinaryTypeTrait.

7.2 is_pointer_interconvertible_with_class trait

Add in place of the description of the `is_pointer_interconvertible_with_class` function above:

```
template <auto m> struct is_pointer_interconvertible_with_class;
```

An IntegralTrait with a BaseCharacteristic of `true_type` if `m` has type "pointer to member of `S` of type `D`," `D` is an object type, and each object `s` of type `S` is pointer-interconvertible (6.7.2 [basic.compound]) with its subobject `s.*m`. Otherwise, the BaseCharacteristic is `false_type`.

A program is ill-formed if it instantiates the definition of this template where `S` is incomplete.

Add to 23.15.2 [meta.type.synop], in place of the corresponding synopsis:

```
template <auto m>
struct is_pointer_interconvertible_with_class;
template<auto m>
  inline constexpr bool is_pointer_interconvertible_with_class_v
    = is_pointer_interconvertible_with_class<m>::value;
```

7.3 is_corresponding_member trait

Add in place of the description of the `is_corresponding_member` function above:

```
template <auto m1, auto m2> struct is_corresponding_member;
```

An IntegralTrait with a BaseCharacteristic of `true_type` if `m1` has type "pointer to member of `S1` of type `D1`," `m2` has type "pointer to member of `S2` of type `D2`," and `m1` and `m2` point to corresponding members of the common initial sequence (12.2 [class.mem]) of `S1` and `S2`. Otherwise, the BaseCharacteristic is `false_type`.

A program is ill-formed if it instantiates the definition of this template where `S1` or `S2` is incomplete.

Add to 23.15.2 [meta.type.synop], in place of the corresponding synopsis:

```
template <auto m1, auto m2>
struct is_corresponding_member;
template<auto m1, auto m2>
  inline constexpr bool is_corresponding_member_v
    = is_corresponding_member<m1,m2>::value;
```

7.4 Note about pointer to member literals, traits version

Add in place of the similar note above:

[*Note:* The type of a pointer-to-member literal is not always as it appears, and this may lead to surprising results when using these traits in conjunction with inheritance in classes that are not standard-layout. Consider the following example:

```
struct A { int a; };
struct B { int b; };
struct C: public A, public B {};

static_assert( is_pointer_interconvertible_with_class_v< &C::b > );
   // Succeeds because, despite its appearance, &C::b has type
   // "pointer to member of B of type int."

static_assert( is_corresponding_member_v< &C::a, &C::b > );
   // Succeeds because, despite appearances, &C::a and &C::b have types
   // "pointer to member of A of type int" and
   // "pointer to member of B of type int," respectively.
```

—*end note*]