Document: P0466R5
Date: 2019-07-19
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: 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 r5 (2019-07-16 PM draft for CWG) to r5 (2019-07-17 draft for LWG): In Cologne on 2019-07-17, CWG asked for one change:

- Remove "as desired" from the note in section 6.

I have updated the paper accordingly. Distaste was also expressed for the use of "is" or "are", rather than "shall be" in the "Comments" column of the table entries, but no change was requested.

Changes from r5 (2019-07-16 AM LWG draft) to r5 (2019-07-16 PM draft for CWG): In Cologne on 2019-07-16, an LWG small group asked for changes:

- Change `is_pointer_interconvertible_base_of<T,T>` to produce `true`, like `is_base_of`.
- Change "shall be" to "is" or "are" in four places.
- Provide more clarity for null member pointers, pointers to member functions, and non-standard-layout classes.
- In the note, avoid suggesting that people may not be confused about standard-layout classes.
- Reword the note to avoid using "this" with an unclear referent.
- Provide a feature-test macro.

I have updated the paper accordingly.

Changes from r5 (CWG draft) to r5 (2019-07-16 AM draft for LWG): In Cologne on 2019-07-15, CWG answered the three questions posed, accepting the changes proposed by LWG members, subject to two further conditions:

- That in the description of `is_pointer_interconvertible_base_of`, the words "possibly *cv*-qualified" be parenthesized.
- That I emphasize to LWG that `is_pointer_interconvertible_base_of_v<T,T>` is `false` under the current wording.

I have incorporated the CWG-approved text into the green editorial instructions, removed the CWG questions, and added text pointing out the `false` value.

Hubert Tong, in correspondence, also asks that LWG check that the functions taking pointers-to-members provide the intended answers when given null pointers, pointers to member functions, and members of classes that are not standard layout. I've added text describing my understanding of the wording in these cases.

Changes from r4 to r5 (draft shown to CWG): Addressing issues raised by Daniel Krügler and Marshall Clow:

- Changing "Requires" to "Mandates" in function descriptions, following new LWG practice.
- Aligning the text suggested to CWG for `is_pointer_interconvertible_base_of` with the existing constraint for `is_base_of`, allowing a trivial answer to be provided in some cases when the type of `Derived` is incomplete.

Changes from r3 to r4: Addressing questions posed by Alisdair Merideth:

- Layout compatibility is an equivalence relation. While the trait has been renamed to the less apparently symmetric `is_layout_compatible`, the description uses the phrase "are layout-compatible" following core usage.
- Pointer-interconvertibility does not respect access control, so `is_pointer_interconvertible_base_of` also does not respect access control.
- `is_pointer_interconvertible_with_class(m)` returns false when `m` is a pointer to member function. This should be clarified in normative text.
- Added three questions for CWG approval. Two involve allowing the traits to be used for *cv* `void` and for arrays of unknown length, despite the incompleteness of those types. The third involves the clarification that a member function cannot be found pointer-interconvertible with an object.

Also, rebasing on N4800.

Changes from r2 to r3: At the 2019 Kona meeting, LEWG approved the change back to `constexpr` functions instead of `template<auto>` for the tests on pointers-to-members. The section "Alternate Wording as Traits" is thus removed.

At the 2018 Batavia meeting of LWG, it was suggested that the two remaining traits can be extended in a trivial way to incomplete non-class types. This change was found acceptable in both LEWG and EWG discussions at Kona in 2019. However, to avoid changing wording that impacts CWG, I'm choosing not to make that change at this time.

Also, based on Batavia LWG feedback, the note about expressions of the form `&T::m` has been reworded to avoid calling such expressions "literals."

Changes from r1 to r2: These changes are based on feedback in the second Core discussion at Jacksonville, 2018-03-16. Each of these changes 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 expressions.

- 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 (10.3 [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 expression of the form `&T::m`. 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 49 in 19.15.6 [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` are complete types, *cv* `void`, or arrays of unknown bound.

To the best of my knowledge, layout compatibility is an equivalence relation. While the trait has been renamed to the less apparently symmetric `is_layout_compatible`, the phrase "are layout-compatible" follows core usage, *e.g.* 6.7 [basic.types]¶11:

> Two types *cv*1 `T1` and *cv*2 `T2` are layout-compatible types if `T1` and `T2` are the same type, layout-compatible enumerations, or layout-compatible standard-layout class types.

Add to 19.15.2 [meta.type.synop], in the section corresponding to 19.15.6 [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 49 in 19.15.6 [meta.rel]:

Template	Condition	Comments
`template <class Base, class Derived> struct is_pointer_interconvertible_base_of;`	`Derived` is unambiguously derived from `Base` without regard to *cv*-qualifiers, and each object of type `Derived` is pointer-interconvertible (6.7.2 [basic.compound]) with its `Base` subobject, or `Base` and `Derived` are not unions and name the same class type without regard to *cv*-qualifiers.	If `Base` and `Derived` are non-union class types and are not (possibly *cv*-qualified) versions of the same type, `Derived` is a complete type.

Here the derivation relationship need not be public, as pointer-interconvertibility does not respect access control. Note that `is_pointer_interconvertible_base_of_v<T,T>` is always `true` under this wording, even though `T` is not derived from itself.

Add to 19.15.2 [meta.type.synop], in the section corresponding to 19.15.6 [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>` (19.15 [meta]), but I don't see a clear choice of subsection to put it in. I suggest a new subsection, "Member relationships."

Add a new subsection after [meta.logical]:

19.15.9 Member relationships **[meta.member]**

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

Mandates: `S` is a complete type.
Returns: `true` if and only if `S` is a standard-layout type, `M` is an object type, `m` is not null, and each object `s` of type `S` is pointer-interconvertible (6.7.2 [basic.compound]) with its subobject `s.*m`.

Add to 19.15.2 [meta.type.synop], in the corresponding position:

// 19.15.9 Member relationships

```
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;
```

Mandates: `S1` and `S2` are complete types.
Returns: `true` if and only if `S1` and `S2` are standard-layout types, `M1` and `M2` are object types, `m1` and `m2` are not null, and `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 expressions

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

[*Note:* The type of a pointer-to-member expression `&C::b` is not always a pointer to member of `C`, leading to potentially surprising results when using these functions in conjunction with inheritance. Consider the following example:

```
struct A { int a; };              // a standard-layout class
struct B { int b; };              // a standard-layout class
struct C: public A, public B {};  // not a standard-layout class

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.

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.
```

—*end note*]

7 Feature-test macros

Add to the table in [support.limits.general]:

Macro name	Value	Header(s)
`__cpp_lib_is_layout_compatible`	*some value*	<type_traits>
`__cpp_lib_is_pointer_interconvertible`	*some value*	<type_traits>

It has been noted that this paper introduces two separate features, one relating to layout-compatibility and common initial sequence, and the other relating to pointer-interconvertibility. I have therefore included two macros.

The macro `__cpp_lib_is_layout_compatible` is intended to cover both `is_layout_compatible` and `is_corresponding_member`. The macro `__cpp_lib_is_pointer_interconvertible` is intended to cover both `is_pointer_interconvertible_base_of` and `is_pointer_interconvertible_with_class`.