Document: P0466R1
Date: 2018-02-12
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Core → Library

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 a `reinterpret_cast` between pointers to layout-compatible types. 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 corresponding to `reinterpret_cast` to and from the initial subobject of a class type, and for correspondence in the common initial sequence of two class types.

Changes since r0: 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.

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 the user of a `reinterpret_cast` might rely on a fact about the type system that can't be asserted: casting between a pointer to an object and a pointer to its initial base or member subobject. A simple type trait handles the base subobject case:

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

The member subobject case turns out to be trickier. The pattern suggests a trait like this:

```
template <class S, class M> struct initial_member_has_type;
```

But that's not really useful. A programmer relying on such a cast almost certainly has a particular member in mind. The test should take a member pointer as a parameter:

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

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`. A template that deduces these types is easier to use:

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

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

```
template <auto m>
struct is_initial_member: is_initial_member_impl< decltype(m), 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 (9.2 [class.mem]) can only be relied upon if the programmer is sure that particular members correspond. So I'm proposing another trait for testing correspondence in the common initial sequence:

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

Like `is_initial_member`, this trait can be implemented by forwarding `decltype(m1)` and `decltype(m2)`.

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

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

static_assert( is_initial_member_v< &C::b > );   // succeeds!
   // &C::b has type int B::*, not int C::*.
```

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.

1 is_layout_compatible

Add to table 40 in 20.15.6 [meta.rel]:

Template	Condition	Comments
`template <class T, class U> struct is_layout_compatible;`	`T` and `U` are layout-compatible (3.9 [basic.types])	

Add to 20.15.2 [meta.type.synop], in the section corresponding to 20.15.6 [meta.rel]:

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

2 is_initial_base_of

Add to table 40 in 20.15.6 [meta.rel]:

Template	Condition	Comments
`template <class Base, class Derived> struct is_initial_base_of;`	`Derived` is a standard-layout class with no non-static data members, and `Base` is the first base of `Derived`.	An object is pointer-interconvertible (3.9.2 [basic.compound]) with its initial base subobject.

Add to 20.15.2 [meta.type.synop], in the section corresponding to 20.15.6 [meta.rel]:

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

3 is_initial_member

This pretty clearly belongs in `<type_traits>` (20.15 [meta]), but I don't see a clear choice of subsection to put it in. Perhaps it goes in 20.15.6 [meta.rel], or perhaps a new subsection, "Member relationships" is appropriate.

Wherever it fits, here is some text to add:

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

A UnaryTypeTrait with a BaseCharacteristic of `true_type` if all of the following conditions hold, and `false_type` otherwise.

- `m` is a member pointer `D S::*m`.
- `S` is a standard-layout type.
- `D` is an object type.
- Either `S` is a union or `m` points to the first non-static data member of `S`. [*Note:* An object is pointer-interconvertible (3.9.2 [basic.compoind]) with its initial member subobjects. —*end note*]

A program which instantiates this template where `D` is not an object type is ill-formed.

[*Note:* The type of a pointer-to-member constant is not always as it appears, and this may lead to errors in using `is_initial_member` in conjunction with inheritance. Consider the following example:

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

static_assert( is_initial_member_v< &C::b > );   // succeeds!
   // &C::b has type int B::*, not int C::*.
```

—*end note*]

Add to 20.15.2 [meta.type.synop], in the corresponding section:

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

4 is_corresponding_member

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

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

A UnaryTypeTrait with a BaseCharacteristic of `true_type` if all of the following conditions hold, and `false_type` otherwise.

- `m1` and `m2` are member pointers `D1 S1::*m1` and `D2 S2::*m2`, respectively.
- `S1` and `S2` are standard-layout types.
- `D1` and `D2` are object types.
- `m1` and `m2` point to corresponding members of the common initial sequence (9.2 [class.mem]) of `S1` and `S2`.

A program which instantiates this template where either `D1` or `D2` is not an object type is ill-formed.

[*Note:* The type of a pointer-to-member constant is not always as it appears, and this may lead to errors in using `is_corresponding_member` in conjunction with inheritance. Consider the following example:

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

static_assert( is_corresponding_member_v< &C::a, &C::b > ); // succeeds!
   // &C::a and &C::b have types int A::* and int B::*, respectively.
```

—*end note*]

Add to 20.15.2 [meta.type.synop], in the corresponding section:

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