Document: P0466R0
Date: 2016-10-15
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: Library Evolution Working Group

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.

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 are_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`. 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_initial_member( M S::*m ) noexcept;
```

The use of this function is a little more broad: it can be called in a non-`constexpr` context. But the implementation is simple. Knowing the internal structure of a pointer-to-member, an implementation may test that the offset of `m` is zero.

```
template <class S, class M>
constexpr bool is_initial_member( M S::*m ) noexcept
   {
    static_assert( is_object<M>::value,
                   "Only data members may be initial." );

    return is_standard_layout<S>::value
        && __member_offset(m) == 0;
   }
```

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 a second function for testing correspondence in the common initial sequence:

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

Once again, the runtime implementation of this function relies on turning the member pointers into an offsets. But this time a compiler intrinsic is required: the offset of the end of the common initial sequence.

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

```
{
 static_assert( is_object<M1>::value,
                "The common initial sequence is only data." );
 static_assert( is_object<M2>::value,
                "The common initial sequence is only data." );

 if (    !is_standard_layout<S1>::value
      || !is_standard_layout<S2>::value )
     return false;

 const auto offset1 = __member_offset(m1);
 const auto offset2 = __member_offset(m2);

 return offset1 == offset2
     && offset1 < __end_of_common_initial_sequence<S1,S2>();
}
```

1 are_layout_compatible

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

Template	Condition	Comments
`template <class T, class U> struct are_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 are_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 an 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 <class S, class M>
constexpr bool is_initial_member( M S::*m ) noexcept;
```

Returns `true` if and only if `S` is a standard-layout class type and 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 `M` is not an object type is ill-formed.

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

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

4 are_common_members

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

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

Returns `true` if and only if both `S1` and `S2` are standard-layout types, and `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 `M1` or `M2` is not an object type is ill-formed.

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

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