Document Number: P0149R0
Date: 2016-2-12
Reply to: jeff-isocpp@caffeinated.me.uk
Audience: Evolution

Generalised member pointers

Jeff Snyder

February 12th, 2016

1 Introduction

C++ member pointers are currently limited to expressing the relationship between a class type and one of its direct members. They are typically represented by storing the offset of the member from the start of the class, since there is always a fixed offset between the pointer to a class and a member of that class.

There are other objects that exist at a fixed offset from the start of the class, but whose offsets cannot be represented by member pointers. These include non-virtual bases, members of members, non-virtual bases of members, and so on. There is no clear distinction between the use cases for offsets to members and the use cases for offsets to other sub-objects, yet the language only permits the formation and use of the former, in the form of "member pointers".

Furthermore, the language only permits member pointers to be used in one direction. From a member pointer to an **int** in class A you can use a member pointer to get a pointer to an **int** within the object of type A from pointer to an object of type A, but given a pointer to an **int** and *a priori* knowledge that that **int** is a specific member of an object of type A, you cannot use the same member pointer in reverse to get a pointer to the full object of type A.

At least some of these limitations have been raised previously as deficiencies in the language, giving us Core Issue 794 [1] and Evolution Issue 94 [2], and have also been questioned on *std-discussion*, yielding the following explanation:

> *"The default state for a language feature is "not present"; this is not a natural consequence of the existing rules, and no-one has proposed adding it."* —Richard Smith, responding to "Why are member data pointers to inner members prohibited?" [3]

In summary, the current constraints on member pointers unnecessarily limit the expressiveness of the language, and the abstractions that can be created with it. This paper proposes extensions to the language to remediate this.

2 Proposal

2.1 Member type upcasts

Currently, the last line of the example below is not valid C++, even though the the intent is clear.

It should be possible to implement an upcast of the member type of a member pointer without difficulty—it would result in applying the same fixed offset as a normal upcast, if any. In the case of virtual bases, doing such an upcast requires runtime type information, which makes implementing the corresponding upcast for a

member pointer difficult. It may also require a breaking change to some ABIs, so supporting upcasts to virtual bases is not proposed here.

Proposal: Upcasts of the member type of member pointers to non-virtual bases of the member type should be permitted. The corresponding downcasts should also be permitted via **static_cast**.

```
struct A {};
struct B : A{};
struct C { B b; };

C c;
B*      to_b   = &c.b;   // OK, Normal pointer
B C::*  c_to_b = &C::b;  // OK, C++98 member pointer
A*      to_a   = to_b;   // OK, C++98 implicit upcast
A C::*  c_to_a = c_to_b; // Valid under P0419
```

2.2 Forming pointers to members of members

To allow member pointers to reference members of members as we as direct members, we need a syntax to form such member pointers. The most natural syntax for this is to take the set of operators that can be applied to an object to get another object which exists at a fixed offset from the first, and allow those operators to be applied to member pointers as well as concrete objects. The three such operators currently in the language are dot (**.**), subscript (**[]**) and member pointer application (**.***).

- **Proposal:** The operator '**.**', when applied to an expression of type "Pointer to member of T1 of class type T2" and an identifier naming a member of T2 of type T3, should result in a value of type "Pointer to member of T1 of type T3". The expression E1.*(E2.*identifier*) should be equivalent to (E1.*E2).*identifier*, where E1 and E2 have types T1 and T2 respectively, and the whole expression has type T3.

- **Proposal:** The operator '**[]**', when applied to an expression of type "Pointer to member of T1 of type array of T2", should result in a value of type "Pointer to member of T1 of type T2". The expression E1.*(E2[E3]) should be equivalent to (E1.*E2)[E3], where E1 and E2 have types T1 and T2 respectively, and E3 is a valid index for T2.

- **Proposal:** The operator '**.***', when applied to an expression of type "Pointer to member of T1 of class type T2" and an expression of type "Pointer to member of T2 of type T3", should result in a value of type "Pointer to member of T1 of type T3". The expression E1.*(E2.*E3) should be equivalent to (E1.*E2).*E3, where E1 and E2 have types T1 and T2 respectively, and the whole expression has type T3.

```
struct A { int i; };
struct B {
    constexpr B(){};
    A a{};
    int is[42]{};
};
constexpr A B::* ap = &B::a;
constexpr int (B::*isp)[42] = &B::is;
constexpr int A::*ip = & A::i;
constexpr B b;
constexpr auto& i_1 = (b.*ap).i;      // OK, C++98
constexpr auto& i_2 = b.*(ap.i);      // Valid under P0149
constexpr auto& is7_1 = (b.*isp)[7]; // OK, C++98
```

```
constexpr auto& is7_2 = b.*(isp[7]);  // Valid under P0149
constexpr auto& i_3 = (b.*ap).*ip;    // OK, C++98
constexpr auto& i_4 = b.*(ap.*ip);    // Valid under P0149
static_assert (&i_1 == &i_2);         // Valid under P0149
static_assert (&i_1 == &i_3);         // OK, C++1z
static_assert (&i_1 == &i_4);         // Valid under P0149
static_assert (&is7_1 == &is7_2);     // Valid under P0149
```

2.3 Forming pointers to bases

With the extensions to member pointers introduced so far, we can form member pointers to direct members, members of members, and bases of members. However, we cannot form a member pointer to a base class. This may be useful in situations where a class inherits multiple copies of a base class via non-virtual inheritance. For example:

```
struct A {};
template <int N> struct B : A {};
struct C : B<0>, B<1> {};
```

It may be useful to create a member pointer of type A C::*, which may point to either of the instances of A that C contains.

We could extend the grammar to allow specifying a base rather than a member in order to form such "member" pointers. However, there is a simpler way of achieving this, at least from the perspecitve of the grammar: we can add a way of forming "identity" member pointers, which would have a type of T T::*. Member pointers to bases could then be formed via upcasts of the member type, just like member pointers to bases of members are formed using earlier parts of this proposal.

```
A    A::* a_to_a    = &A::this;   // P0149 formation of 'identity' member pointer
A B<1>::* b1_to_a   = a_to_a;     // C++98 class type downcast
A    C::* c_to_b1_a = b1_to_a;    // C++98 class type downcast

C    C::* c_to_c    = &C::this;   // P0149 formation of 'identity' member pointer
B<1> C::* c_to_b1   = c_to_c;     // P0149 member type upcast
A    C::* c_to_b1_a = c_to_b1;    // P0149 member type upcast

A    C::* c_to_b1_a = &B<1>::this; // Combo!
```

Proposal: Expressions of the form &T::**this** have type "Pointer to member of T of type T". The expression E1.*&T::**this** should be valid if E1 has type T, and should be equivalent to the exprssion E1.

2.4 Class-from-member pointers

Given a pointer to an object of type A and a priori knowledge that that object is a specific subobject of another type B, it is often useful to be able to obtain a pointer to the complete B object. In the absence of virtual inheritance, this operation is a simple negative offset.

The usefulness of this is exemplified by node-based containers which store their header node as a member of the container type, such as Boost.intrusive's tree data structures. Given a pointer to a node in such a container, it is possible to navigate up the tree and identify the header node. Since the header node is at a fixed offset in memory from the container object, it should then be possible to get a pointer to the actual container object. Unfortunately this cannot be portably done in today's C++, and so Boost.intrusive uses compiler-specific hacks to implement its **container_from_iterator** functions.

We can straightforwardly allow portable class-from-member conversions by allowing the negation of member pointers. With this extension, "member pointer" is no longer an appropriate term. Moreover, via composition of a negated member with a normal member pointer, it would be possible to form "member-to-member" pointers.

To avoid the confusion associated with the term "member pointer" in this context, this paper will use the phrase "Pointer to object of type T1 from object of type T2" as a generalisation of the phrase "Pointer to member of T2 of type T1". Note that the order in which the types appear in the phrase is reversed.

Example:

```
struct A{};
struct B
{
    A a1;
    A a2;
};

auto b_to_a1 = &B::a1;
auto b_to_a2 = &B::a2;
auto a1_to_b = ~b_to_a1; // a1_to_b has type ''Pointer to object of type B from object of type A''
auto a1_to_a2 = a1_to_b .* b_to_a2; // a1_to_a2 hsa type ''Pointer to object of type A from object of type A''
```

Proposal: Expressions of the form ~E1, where E1 has type "Pointer to object of type T1 from object of type T2", should have the type "Pointer to object of type T2 from object of type T1". The expression E2.*E1.*~E1 should be valid if E2 is of type T2, should have type T2, and should be equivalent to E2.

Acknowledgements

Many thanks to Richard Smith for his feedback during its development on the ideas beind the proposal, the syntax used and its implementability.

References

[1] Detlef Vollman. Base-derived conversion in member type of pointer-to-member conversion. Technical Report CWG794, March 2009.
`http://wg21.link/cwg794`.

[2] Detlef Vollman. Base-derived conversion in member type of pointer-to-member conversion. Technical Report EWG94, March 2009.
`http://wg21.link/ewg94`.

[3] Richard Smith and "Myriachan". *[std-discussion] Re: Why are member data pointers to inner members prohibited?*, June 2015.
`https://groups.google.com/a/isocpp.org/d/msg/std-discussion/8tehjvbLEWQ/1oPQyuYw2k8J`.