Generalised member pointers

Document #:	P0149R3
Date:	2025-05-19
Project:	Programming Language C++
Audience:	Evolution

Reply-to: Jeff Snyder <jeff-isocpp@caffeinated.me.uk>

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".

Some of these limitations have been raised previously as deficiencies in the language, giving us Core Issue 794 [CWG794] and Evolution Issue 94 [EWG94], 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?" [std-discussion-20150605]

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 Revision History

2.1 R0: Summer 2016

Initial revision

2.2 R1: January 2025

The class-from-member pointers were removed from the proposal removed based on EWG's feedback. Wording for the remaining features has been added, along with a section on implementability.

2.3 R2: February 2025 (as presented in Hagenberg)

The '`->*`' and unary '`*`' operators were added to the proposal section, having previously only been mentioned in the wording. Various phrasing and formatting fixes and improvements were made in proposal section. The wording was updated based on feedback from Thomas Köppe.

2.4 R3: May 2025

Updated to reflect a clarifying poll in EWG that made `E1->E2` equivalent to `(*(E1)).E2` and `E1->*E2` equivalent to `(*(E1)).*E2` in all cases. Updated wording based on feedback from Thomas Köppe and Brian Bi.

3 Proposal

3.1 Member type upcasts

Currently, the last line of the example below is not valid C++, even though 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
A*       to_a  = to_b;                    // OK, C++98 implicit upcast
B*       to_b2 = static_cast<B*>(to_a);   // OK, C++98 static downcast

B C::* c_to_b  = &C::b;                       // OK, C++98 member pointer
```

```
A C::* c_to_a  = c_to_b;                          // Valid under P0419
B C::* c_to_b2 = static_cast<B C::*>(c_to_a); // Valid under P0419
```

3.2 Forming pointers to members of members

To allow member pointers to reference members of members as well 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 pointer to member of `T1` of type `T2` respectively, and the whole expression has type `T3`.

  ```
  struct A { int i; };
  struct B { A a; };
  constexpr A B::* ap = &B::a;

  B b;
  constexpr int& i_1 = (b.*ap).i; // OK, C++98
  constexpr int& i_2 = b.*(ap.i); // Valid under P0149
  static_assert(&i_1 == &i_2);    // Valid under P0149
  ```

- **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 pointer to member of `T1` of type array of `T2` respectively, `E3` is a valid index for the array identified by `E2`, and the whole expression has type `T2`.

  ```
  struct A { int is[42]; };
  constexpr int (A::*isp)[42] = &A::is;

  A a;
  constexpr int& is7_1 = (a.*isp)[7]; // OK, C++98
  constexpr int& is7_2 = a.*(isp[7]); // Valid under P0149
  static_assert(&is7_1 == &is7_2);    // Valid under P0149
  ```

- **Proposal:** The unary 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)` should be equivalent to

`*(E1.*E2)`, where `E1` and `E2` have types `T1` and pointer to member of `T1` of type array of `T2` respectively, and the whole expression has type `T2`.

The operator '`->`', when applied to expressions `E1` and `E2`, is equivalent to `(*(E1)).E2` in cases that exercise the new functionality of unary operator '`*`', just as it is for existing use cases. Note that this applies both in the case of `E1` being a pointer to a pointer to member and in the case of `E1` being a pointer to member of array type.

```
struct A { int is[42]; };
constexpr int (A::*isp)[42] = &A::is;

A a;
constexpr int& is0_1 = *(a.*isp);     // OK, C++98
constexpr int& is0_2 = a.*(*isp);     // Valid under P0149
static_assert(&is0_1 == &is0_2);      // Valid under P0149

struct B { int i; }
struct C { B b; }
constexpr B C::*bp = &C::b;

C c;
constexpr int& i_1 = &(c.*bp)->i;     // OK, C++98
constexpr int& i_2 = c.*((&bp)->i);   // Valid under P0149
static_assert(&i_1 == &i_2);          // Valid under P0149

struct C { int i; }
struct D { C cs[42]; };
constexpr C (D::*csp)[42] = &D::cs;

D d;
constexpr int& cs0i_1 = (d.*csp)->i; // OK, C++98
constexpr int& cs0i_2 = d.*(csp->i); // Valid under P0149
static_assert(&cs0i_1 == &cs0i_2);   // Valid under P0149
```

- **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`, `E2` and `E3` have types `T1`, pointer to member of `T1` of type `T2`, and pointer to member of `T2` of type `T3` respectively, and the whole expression has type `T3`.

  The operator '`->*`', when applied to expressions `E1` and `E2`, is equivalent to `(*(E1)).*E2` in cases that exercise the new functionality of operator '`.*`', just as it is for existing use cases. Note that this applies both in the case of `E1` being a pointer to a pointer to member and in the case of `E1` being a pointer to member of array type.

  ```
  struct A { int i; };
  struct B { A a{}; };
  ```

```
constexpr A   B::*ap = &B::a;
constexpr int A::*ip = &A::i;

B b;
constexpr int& i_1 = (b.*ap).*ip;      // OK, C++98
constexpr int& i_2 = b.*(ap.*ip);      // Valid under P0149
static_assert(&i_1 == &i_2);           // Valid under P0149

constexpr int& i_3 = (&(b.*ap))->*ip; // OK, C++98
constexpr int& i_4 = b.*((&ap)->*ip); // Valid under P0149
static_assert(&i_3 == &i_4);           // Valid under P0149

struct C { int i; }
struct D { C cs[42]; };
constexpr int A::*ip       = &A::i;
constexpr C  (D::*csp)[42] = &D::cs;

D d;
constexpr int& cs0i_1 = (d.*csp)->*ip; // OK, C++98
constexpr int& cs0i_2 = d.*(csp->*ip); // Valid under P0149
static_assert(&cs0i_1 == &cs0i_2);     // Valid under P0149
```

3.3 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 {};
struct B1 : A {};
struct B2 : A {};
struct C : B1, B2 {};
```

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 can 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.

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 expression `E1`.

```
B1 B1::* b1_to_b1   = &B1::this; // P0149 formation of 'identity' member pointer

B1  C::*  c_to_b1   = b1_to_b1;  // C++98 class type downcast
A  B1::* b1_to_a    = b1_to_b1;  // P0149 member type upcast

A   C::*  c_to_b2_a = &B2::this; // Combo!
```

4 Implementation

Implementation of this proposal on most platforms (including all Itanium ABI platforms) is believed to be straightforward. Member pointers are generally represented as offsets, and the features in this paper adjust and combine those offsets in the same way that corresponding parts of the language dealing with concrete addresses adjust those addresses.

For example, given an array `A` with element type `T` and an index `I`, calculating the address of `A[I]` involves adding `sizeof(T)*I` to `&A`; whereas given a similar pointer to member `PTM` of type "pointer to member of class `C` of type array of `T`", the equivalent operation is to add `sizeof(T)*I` to the offset that is `PTM`'s representation.

Finding the address of a member in a virtual base class requires far more than just an offset, and for this reason the standard disallows formation of pointers to members in virtual base classes. However, MSVC goes above and beyond in this regard to support pointers to members of virtual bases, and as a result has a more complex pointer-to-member representation.

In its most general form, the representation of a pointer to a data member is:

1. An offset from the object address to the relevant vtable pointer
2. A virtual base table index
3. An offset from the start of the virtual base to the member

The representation of a pointer to a member function is the same as the above, with the addition of a function address or virtual function index.

This extension is fundamentally incompatible with using the `.*` operator to form transitive pointers to members. If only the first operand involves a virtual base table index, we can take the first operand and add the second operand's offset to (3). Similarly, if only the second operand involves a virtual base table index, we can take the second operand and add the first operand's offset to (1). However, if both operands involve a virtual base index, we're out of luck—the resulting pointer-to-member representation would have to store three offsets and two virtual base table indices. To handle this, the implementation would likely need a dynamically sized representation allocated in the free store, possibly forcing an ABI break.

Instead, on the basis that use of pointers to members is fairly rare, use of virtual bases is also rare, and combining them is especially rare (not to mention non-standard), the

easier path forward here would be to phase out support for this extension in MSVC. Concretely, this would involve:

- Deprecating the formation of pointers to members in virtual bases in MSVC, with a warning
- Disallowing formation of pointers to members in virtual bases when compiling with /std:c++26
- Supporting the cases that are implementable with the current representation
- Trapping in the `.*` operator if both operands are pointers to members in virtual bases

Disallowing formation of pointers to members in virtual bases when compiling with `/std:c++26` makes it hard to hit the trapping case. Doing so would require having new C++26 code that uses the `.*` operator on two operands supplied from one or more different translation units which were compiled with an older standard.

5 Wording

Change 7.3.13 **[conv.mem]** paragraph 2 as follows:

> A prvalue expression `E` of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to a prvalue of type "pointer to member of `D` of type *cv* ~~`T`~~`U`", where `D` is a complete class derived (11.7) from B, and `T` is the same as or a complete class derived (11.7) from `U`.
>
> If either `B` or `U` is an inaccessible (11.8), ambiguous (6.5.2), or transitively virtual ~~(11.7.2)~~ base class (11.7.2) of `D` or `T` respectively~~, or a base class of a virtual base class of D~~, a program that necessitates this conversion is ill-formed.
>
> If class `D` does not contain the original ~~member~~subobject and is not a base class of the class containing the original ~~member~~subobject, the behavior is undefined. Otherwise, the ~~result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The~~ result refers to the ~~member in D's instance of B~~instance of U that is the member designated by E or is a base subobject thereof, as if it were a subobject of D.
>
> ~~Since the result has type "pointer to member of D of type cv T", indirection through it with a D object is valid. The result is the same as if indirecting through the pointer to member of B with the B subobject of D.~~
>
> The null member pointer value is converted to the null member pointer value of the destination type.

Change 7.6.1.2 **[expr.sub]** paragraph 2 as follows:

> With the built-in subscript operator, an expression-list shall be present, consisting of a single assignment expression. One of the expressions shall be a glvalue of type "array of `T`" ~~or~~, a prvalue of type "pointer to `T`" or a prvalue of type "pointer to member of class `C` of type array of `T`", and the other shall be a prvalue of unscoped enumeration or integral type.
>
> ~~The~~Let `E1` and `E2` denote these two expressions, respectively. The type "`T`" shall be a completely-defined object type. If `E1` is of pointer to member type and `E2` is within the bounds of the member array identified by `E1` and not past its end, the result is a prvalue of type "pointer to member of class `C` of type `T`" and `E` designates the element of the member array designated by `E1` at index `E2` as if it were a subobject of `C`. If `E1` is of pointer to member type and `E2` is not within the bounds of the member array identified by `E1`, behaviour is undefined. Otherwise, the result is of type "`T`", and~~. The type "`T`" shall be a completely-defined object type. The expression~~ `E1[E2]` is identical (by definition) to `*((E1)+(E2))`, except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise.

Change 7.6.1.5 **[expr.ref]** paragraph 2 as follows:

> For the first option (dot), if the id-expression names a static member or an enumerator, the first expression is a discarded-value expression (7.2.3); if the id-expression names a non-static data member, the first expression shall be a glvalue or a prvalue having pointer-to-member type. For the second option (arrow), the first expression shall be a prvalue having either pointer type or pointer-to-member type. The expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of 7.6.1.5 will address only the first option (dot).

Change 7.6.1.5 **[expr.ref]** paragraph 5 as follows:

> Otherwise, the object expression shall be of class type X or type "pointer to member of class `C` of class type `X`". ~~The class type~~In either case, `X` shall be complete unless the class member access appears in the definition of that class.

Change 7.6.1.5 **[expr.ref]** paragraph 6 as follows:

> If `E1` does not have pointer-to-member type and `E2` is a bit-field, `E1.E2` is a bit-field. The type and value category of `E1.E2` are determined as follows. In the remainder of 7.6.1.5, cq represents either const or the absence of const and vq represents either volatile or the absence of volatile. cv represents an arbitrary set of cv-qualifiers, as defined in 6.8.5.
>
> Furthermore, let the notation *vq12* stand for the "union" of *vq1* and *vq2*;

that is, if *vq1* or *vq2* is `volatile`, then *vq12* is `volatile`. Similarly, let the notation *cq12* stand for the "union" of *cq1* and *cq2* ; that is, if *cq1* or *cq2* is `const`, then *cq12* is `const`.

Add the following as new paragraphs following 7.6.1.5 **[expr.ref]** paragraph 6:

For a type "*cq1 vq1* `A`" and the type "*cq2 vq2* `B`" of `E2`, the merged type of "*cq1 vq1* `A`" and `E2` is

- "*vq12* `B`" if `E2` is declared to be a `mutable` member,
- "*cq12 vq12* `B`" otherwise.

If the type of `E1` is "pointer to member of class `C` of type `X`", `E2` shall name a non-static member of non-reference type. If `E1` is the null member pointer value, behaviour is undefined. If `E2` is a member of an inaccessible (11.8), ambiguous (6.5.2), or transitively virtual base class (11.7.2) of `X`, the program is ill-formed. The expression `E1.E2` is a prvalue of type "pointer to member of class `C` of type `M`", where `M` is the merged type of `X` and `E2`, and designates the member `E2` of `X` as if it were a subobject of `C`. [Note—If `E2` names an overload set, the expression `E1.E2` can be used only in a context that uniquely determines which function in the overload set is selected (see 12.3 **[over.over]**)]

Change 7.6.1.5 **[expr.ref]** paragraph 7 as follows:

If `E1` does not have pointer-to-member type and `E2` is declared to have type "reference to `T`", then `E1.E2` is an lvalue of type `T`. If `E2` is a static data member, `E1.E2` designates the object or function to which the reference is bound, otherwise `E1.E2` designates the object or function to which the corresponding reference member of `E1` is bound. Otherwise, one of the following rules applies.

- If `E2` is a static data member and the type of `E2` is `T`, then `E1.E2` is an lvalue; the expression designates the named member of the class. The type of `E1.E2` is `T`.
- If `E2` is a non-static data member and the type of `E1` is ~~"*cq1 vq1* `X`", and the type of `E2` is "*cq2 vq2* `T`"~~`X`, the expression designates the corresponding member subobject of the object designated by the first expression. If `E1` is an lvalue, then `E1.E2` is an lvalue; otherwise `E1.E2` is an xvalue. ~~Let the notation *vq12* stand for the "union" of *vq1* and *vq2* ; that is, if *vq1* or *vq2* is `volatile`, then *vq12* is `volatile`. Similarly, let the notation *cq12* stand for the "union" of *cq1* and *cq2* ; that is, if *cq1* or *cq2* is `const`, then *cq12* is `const`. If `E2` is declared to be a mutable member, then the~~The type of `E1.E2` is the merged type of `X` and `E2`~~"*vq12* `T`". If `E2` is not declared to be a mutable member, then the type of `E1.E2` is "*cq12 vq12* `T`"~~.

Change 7.6.1.9 **[expr.static.cast]** paragraph 2 as follows:

> An lvalue of type "*cv1* `B`", where `B` is a class type, can be cast to type "reference to *cv2* `D`", where `D` is a complete class derived (11.7) from `B`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If `B` is a ~~virtual base class of `D` or a base class of a~~transitively virtual base class of `D` (11.7.2), or if no valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (7.3.12), the program is ill-formed. An xvalue of type "*cv1* `B`" can be cast to type "rvalue reference to *cv2* `D`" with the same constraints as for an lvalue of type "*cv1* `B`". If the object of type "*cv1* `B`" is actually a base class subobject of an object of type `D`, the result refers to the enclosing object of type `D`. Otherwise, the behavior is undefined. [Example: ...]

Change 7.6.1.9 **[expr.static.cast]** paragraph 12 as follows:

> A prvalue of type "pointer to member of ~~`D`~~`A` of type *cv1* `T`" can be converted to a prvalue of type "pointer to member of `B` of type *cv2* ~~`T`~~`U`", where ~~`D` is a~~`A` and `B` are complete class types~~ and `B` is a base class (11.7) of `D`~~, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If no valid standard conversion from "pointer to member of ~~`B`~~`A` of type `T`" to "pointer to member of ~~`D`~~`B` of type `T`" exists (7.3.13), and no valid standard conversion from "pointer to member of `B` of type `T`" to "pointer to member of `A` of type `T`" exists, the program is ill-formed. If no valid standard conversion from "pointer to member of `A` of type `T`" to "pointer to member of `A` of type `U`" exists, and no valid standard conversion from "pointer to member of `A` of type `U`" to "pointer to member of `A` of type `T`" exists, the program is ill-formed. The null member pointer value (7.3.13) is converted to the null member pointer value of the destination type. If class `B` contains the original ~~member~~subobject, or is a base class of the class containing the original ~~member~~subobject, and if `U` is either the same as `T` or is a base class of the most derived type of the original subobject, the resulting pointer to member points to the original ~~member~~subobject or one of its bases or derived classes, according to the conversion from `T` to `U`. Otherwise, the behavior is undefined.

Change 7.6.2 **[expr.unary]** paragraph 1 as follows:

Expressions with unary operators group right-to-left.

unary-expression:
postfix-expression
unary-operator cast-expression
`++` *cast-expression*
`--` *cast-expression*
`&` *pm-identity*
await-expression
sizeof *unary-expression*
sizeof `(` *type-id* `)`
sizeof `...` `(` *identifier* `)`
alignof `(` *type-id* `)`
noexcept-expression
new-expression
delete-expression
unary-operator: one of
`*` `&` `+` `-` `!`

Change 7.6.2.2 **[expr.unary.op]** paragraph 1 as follows:

> The unary `*` operator performs *indirection*: Its operand shall be either a prvalue of type "pointer to `T`", where `T` is an object or function type, or a prvalue of type "pointer to member of class `C` of type array of `T`". If its operand has pointer-to-member type, then the expression designates the first element of the array as if it were a subobject of `C`, and operator yields a prvalue of type "pointer to member of class `C` of type `T`". Otherwise, the~~The~~ operator yields an lvalue of type `T`~~. If~~, and if the operand points to an object or function, the result denotes that object or function; otherwise, the behavior is undefined except as specified in 7.6.1.8.

Add the following as a new section named **[expr.unary.pmidentity]** as a subsection of 7.6.2 **[expr.unary]**:

> A expression of the form `&`*pm-identity* is used to form an identity pointer-to-member. [Note—such member pointers can have their member type upcast and/or class type downcast (7.3.13 **[conv.mem]**) to form a pointer-to-member that identifies a base subobject.]
>
> *pm-identity:*
> *nested-name-specifier* `this`
>
> The *nested-name-specifier* in a *pm-identity* shall name a class. The expression `&` *pm-identity* is a prvalue of type "pointer to member of class `T` of type `T`", where `T` is the class named by the *nested-name-specifier*. The expression designates an object of type `T` as if it were a subobject of itself.

Drafting note: the *nested-name-specifier* is assumed to name a class name in a dependent context. Use of typename is intentionally not allowed.

Change 7.6.4 **[expr.mptr.oper]** paragraph 2 as follows:

The binary operator `.*` binds its second operand, which shall be a prvalue of type "pointer to member of `T` of type `M`" to its first operand, which shall be a glvalue of ~~class T~~type `U` or of type "pointer to member of class `C` of type `U`", where `U` is either `T` or ~~of~~ a class of which `T` is an unambiguous and accessible base class. ~~The~~If the first operand is of class type, the result is an object or a function of the type specified by the second operand. Otherwise, if the first operand is of pointer to member type, `T` shall not be a transitively virtual base class (11.7.2) of `U`, and the result is a prvalue of type "pointer to member of `C` of type `M`".

Change 7.6.4 **[expr.mptr.oper]** paragraph 3 as follows:

The binary operator `->*` binds its second operand, which shall be a prvalue of type "pointer to member of `T`" to its first operand, which shall be of type "pointer to `U`" , "pointer to pointer to member of class `C` of type `U`" or "pointer to member of class `C` of type array of `U`", where `U` is either `T` or a class of which `T` is an unambiguous and accessible base class. The expression `E1->*E2` is converted into the equivalent form `(*(E1)).*E2`. The remainder of 7.6.4 will address only expressions of the form `E1.*E2`.

Change 7.6.4 **[expr.mptr.oper]** paragraph 4 as follows:

Abbreviating *pm-expression* `.*`*cast-expression* as `E1.*E2`, `E1` is called the *object expression*. If `E1` is of class type and the result of `E1` is an object whose type is not similar to the type of `E1`, or whose most derived object does not contain the member to which `E2` refers, the behavior is undefined. The expression `E1` is sequenced before the expression `E2`.

Change 7.6.4 **[expr.mptr.oper]** paragraph 6 as follows:

If the result of `.*` ~~or ->*~~ is a function, then that result can be used only as the operand for the function call `operator ()`. [Example: ...] In a `.*` expression whose object expression is an rvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&`, unless its cv-qualifier-seq is const. In a `.*` expression whose object expression is an lvalue, the program is ill-formed if the second operand is a pointer to member function whose *ref-qualifier* is `&&`. If `E1` is of pointer to member type or `E2` is a pointer to a member function, the result of the expression is a prvalue. ~~The result of a .* expression whose second operand~~Otherwise, if `E1` is of class type and `E2` is a pointer to a data member, the result of the expression is an lvalue if ~~the first operand~~`E1` is an lvalue and an xvalue otherwise. ~~The result of a .* expression whose second operand is a pointer to a member function is a prvalue.~~ If ~~the second operand is the~~`E1` or `E2` is a null member pointer value (7.3.13), the behavior is undefined.

Add the following as new paragraph following 7.6.4 **[expr.mptr.oper]** paragraph 6:

If the type of `E1` is "pointer to member of class `C` of type `U`", the object expression refers to the member of `T` designated by `E2` within the subobject of `C` designated by `E1` as if it were a subobject of `C`. Otherwise, the object expression refers to the member of `T` designated by `E2` within the object referred to by `E1`. If the subobject denoted by E1 does not contain the subobject denoted by E2, behaviour is undefined.

Change 11.7.2 [**class.mi**] paragraph 4 as follows, and add a paragraph break before the example:

A base class specifier that does not contain the keyword `virtual` specifies a *non-virtual base class*. A base class specifier that contains the keyword `virtual` specifies a *virtual base class*. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (6.7.2) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type. A *transitively virtual base class* of a class `C` is a virtual base class of `C` or a base class of a virtual base class of `C`.

Change 12.2.4.3 [**over.ics.rank**] paragraph 4:

...

- If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from `B`,

  ...

- If `pm1`, `pm2` and `pm3` are pointer-to-member types, where `pm1` can be converted to `pm2` and `pm2` can be converted to `pm3` (7.3.13 [**conv.mem**]),
  - conversion of `pm1` to `pm2` is better than conversion of `pm1` to `pm3`, and
  - conversion of `pm2` to `pm2` is better than conversion of `pm1` to `pm3`

Change 12.3 [**over.over**] paragraph 1 as follows:

[...]

If the target type contains a placeholder type, placeholder type deduction is performed ([**dcl.type.auto.deduct**]), and the remainder of this subclause uses the target type so deduced. If the id-expression is the second operand of a class member access expression where, the first operand has pointer-to-member type after its conversion to the dot form, the class member access expression can be preceded by `&`. Otherwise, t~~T~~he id-expression can be preceded by the `&` operator.

Add an entry to 15.11 [**cpp.predefined**] table 22 [**tab:cpp.predefined.ft**]:

`__cpp_pointer_to_subobject 202502L`

Acknowledgements

Many thanks to Richard Smith for his feedback on this proposal and the ideas behind it, to Jens Maurer, Thomas Köppe and Brian Bi for their help and guidance with its wording, and to Jonathan Caves for his help with answering the question of MSVC implementability.

References

[CWG794] Detlef Vollman. *C++ Standard Core Language Issue 794, Base-derived conversion in member type of pointer-to-member conversion*, March 2009. http://wg21.link/cwg794.

[EWG94] Detlef Vollman. *C++ Standard Evolution Issue 94, Base-derived conversion in member type of pointer-to-member conversion*, March 2009. http://wg21.link/ewg94.

[std-discussion-20150605] 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.