Document : N3772
Date : 2013-09-05
Project : Programming Language C++
Addresses : Evolution Working Group
Reply To : dibeas@ieee.org

Changing the type of address-of-member expression

This proposal addresses the same issue as CWG closed issue 203 "Type of *address-of-member*" [1] by providing a broader view of the issue and the implications that it has on other parts of the C++ standard.

I will try to provide a rationale for the change, examples of code that the current standard allows and should probably not compile and code that does not compile in the current standard and should arguably compile.

Type of *address-of-member*

The N3376 draft of the standard (latest as of this writing) section 5.3.1 paragraph 3 states:

> *The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a qualified-id. If the operand is a qualified-id naming a non-static member m of some class C with type T, the result has type "pointer to member of class C of type T" and is a prvalue designating C::m. Otherwise[...] [Example:*
>
> ```
> struct A { int i; };
> struct B : A { };
> ... &B::i ... // has type int A::*
> ```
>
> *— end example]*

The standard clearly states that the type of the expression *address-of-member* is a pointer to a member of the class that declares the member, and not a pointer to a member of the class in the qualified-id.

Lisa Lippincott considers that behavior unintuitive, and I must agree with her. The notes after the 04/00 meeting offer a rationale for the decision:

> *The rationale for the current treatment is to permit the widest possible use to be made of a given address-of-member expression.*

According to that rationale, if the DR or this proposal was to be accepted it would unnecessarily limit the uses of the *address-of-member* expression. My position, contrary to that opinion, is that the current treatment is not strictly more generic, as it inhibits some code patterns that we might want to allow. Furthermore, the extra generality allows for code that I don't believe we want to support.

Generality of the current behavior

As pointed in the comment after the 04/00 meeting, the pointer-to-member referring to a member of a base type can be implicitly converted to a *pointer-to-member* to the derived type under most common circumstances. But this requires a conversion.

Consider the definitions of A and B from the example in 5.3.1 and the following code that does not compile:

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

template <typename T, int T::*PTR>
struct example1 { ... };

example1<B, &B::i> x;       // error
```

The problem with the example is that for a non-type template argument the set of allowed implicit conversions are limited, and does not include the conversion needed. This can be fixed in code through explicit casting:

```
example1<B, static_cast<int B::*>(&B::i)> x;
```

In the example above the impact of the rule is just a bit of surprise and unnecessarily ugliness in the code. But the rule has direct impact on the design of components, including some present in current standard library implementations.

In the C++ language, and lacking a *static-if* [2][3] construct a common approach to specialize the behavior of templates is the abuse of inheritance. The public interface of a template will take a set of arguments, the parameters will be passed on to a base template together with the result of the evaluation of some traits. The base template can then be specialized on the different values of the traits. This technique is used in Bloomberg's Basic Standard Library[4]: the `bsl::pair<>` template inherits from `bsl::Pair_Imp<>` template to specialize the behavior for types that use the allocators[1].

For a different reason, the implementation of the standard library that ships with Microsoft's Visual Studio compiler (VS2010 and over) also splits the implementation of the `std::pair<>` template into a base and derived types, with the base holding the actual data members.

---

[1] In BSL, the default allocator in containers is polymorphic in nature[6] and similar to `std::scoped_allocator<>` in that it *propagates* (by lack of a better term) to the contained objects for their memory allocation, so in a `bsl::vector<bsl::string>` the strings use the same allocator that the container itself. At the same time, the allocators don't propagate on copy, providing the invariant that all objects stored in a container using the default allocator share the same allocator. The `bsl::Pair_Imp<>` specializations help managing the transfer of allocators from containers while blocking the transfer of allocators on assignment and copy construction.

In both cases, the inheritance is an implementation detail. The types are not polymorphic and Liskov's Substitution Principle does not apply. Inheritance is used to model *implemented-in-terms-of* rather than *is-a*. The idiom in C++ would be is using non-public inheritance[5].

For most uses of `std::pair<>` and `bsl::pair<>` private inheritance together with a *using-declaration* suffices:

```
template <typename T1, typename T2>
class pair : private Pair_Imp<T1,T2,
             BloombergLP::bslma::UsesBslmaAllocator<T1>::value,
             BloombergLP::bslma::UsesBslmaAllocator<T2>::value>
{
   typedef Pair_Imp<T1,T2,
              BloombergLP::bslma::UsesBslmaAllocator<T1>::value,
              BloombergLP::bslma::UsesBslmaAllocator<T2>::value>
                                 Base;
   ...
public:
   using Base::first;
   using Base::second;
   ...
};
```

The using declaration brings the members first and second from the base type and makes them accessible in the public section of the derived type. This works in most situations, except when *pointers-to-member* are used.

With the current definition of the address-of-member expression, the expression `&bsl::pair<int,int>` yields a *pointer-to-member* of `bsl::Pair_Imp` that cannot be converted to a pointer-to-member of `bsl::pair`, as the inheritance relationship is private:

```
int bsl::Pair_Imp<int,int,0,0>::*p1
            = &bsl::pair<int,int>::first; // OK
int bsl::pair<int,int>::*p2
            = &bsl::pair<int,int>::first  // Error
```

This is clearly an example where the current behavior in the standard is not more generic than if the expression yielded a pointer to member to the type on which the *address-of-member* was applied.

The solution taken in both libraries is making the inheritance public, and expecting users to explicitly cast the pointer to member if they need a *pointer-to-member* to the pair as a non-type template argument. The implementation detail is leaked to users and we end up with what is arguably a worse design.

When the extra generality is undesired

While the previous examples show where the current definition limits or complicates code that we want to support, the following code shows how the current definition enables code that we don't want to support and was raised as Defect Report 1007 [7]:

```
struct base {
    protected: int x;
};
struct derived : base {
    void foo(base* b) {
        b->x = 123; // not ok
        (b->*(&derived::x)) = 123; // ok?!
    }
};
```

The defect report was closed as not a defect with the following rationale:

> *Access applies to use of names, so the check must be done at the point at which the pointer-to-member is formed. It is not possible to tell from the pointer to member at runtime what the access was.*

I contend that the core issue in the code above is that the *address-of-member* yields a *pointer-to-member* to base, even though the access specifier is checked at the derived level. This dissociation is what allows the code above to compile. If the expression was modified to yield a *pointer-to-member* to the type on which the expression is applied, the code in the defect report would not compile, as you cannot apply the *pointer-to-member* to derived on a base object.

That is, by changing the semantics of the *address-of-member* to yield a *pointer-to-member* to derived the expression fails to compile. This does not really solve the issue of access specifiers, as user code can explicitly cast the *pointer-to-member* to derived to a pointer to a pointer-to-member to base, which at least makes more explicit that something funny is going on.

```
(b->*(static_cast<int base::*>(&derived::x))) = 123;
```

Impact on existing code

I expect the change to the type of the *pointer-to-member* expression to have little overall impact at the source code level. In the few cases where the old behavior might be desired, the code will fail to compile with a hard error and the programmer will be able to fix it by changing the type in the *address-of-member* expression to refer to the type that declares the member. For the few cases where modifying the *address-of-member* expression is not feasible due to the access specifiers, the programmer can still perform a conversion with a `static_cast<>` as shown above, even if in my opinion the design is wrong and should be corrected.

Another source code incompatibility will arise when trying to obtain a pointer to a member defined in an ambiguous base. While this was ill-formed in the previous standard, it is accepted in C++11 due to the fixes for Defect Report 1121[7], the solution in this case is still the same: change the type of the *address-of-member* expression to refer to the base type. In this particular

case, if that member is not accessible where the expression is used, or if the conversion from pointer to member of derived to pointer to member to base is not available there would be no simple workaround.

Besides issues at the source code level, this change can break binary compatibility across translation units that are compiled with the old and the new behavior in those cases where the type of the pointer to member is deduced by the compiler:

```
template <typename T, typename U, typename V>
void reset_member(T & t, V& U::*ptr) {
   (t.*ptr) = V();
}
...
reset_member(obj,&Type::member);
```

This case is more problematic as it causes an ODR violation, and can cause undefined behavior. This can be fixed by recompiling the whole program with a compiler implementing the new behavior. While this case is far more dangerous than the source code compatibility as it will silently fail. I am unaware of how often this type of code is used in production systems, and how often the systems that use this code would not recompile the whole program when moving from a standard version to another.

Proposal

Change 5.3.1 [expr.unary] paragraph 3

> *The result of the unary & operator is a pointer to its operand. The operand shall be either an lvalue of type other than "array of runtime bound" or a qualified-id. If the operand is a qualified-id in the form D::m, naming a non-static member m of some class C with type T, the result has type "pointer to member of class ~~C~~ D of type T" and is a prvalue designating ~~C~~ D::m; the program is ill formed if C is an ambiguous base (10.2) of the class designated by the nested-name-specifier of the qualified-id. Otherwise, if the type of the expression is T, the result has type "pointer to T" and is a prvalue that is the address of the designated object (1.7) or a pointer to the designated function. [Note: In particular, the address of an object of type "cv T" is "pointer to cv T", with the same cv-qualification. — end note] [Example:*
>
> ```
> struct A { int i; };
> struct B : A { };
> ... &B::i ... // has type int A B::*
> ```
>
> *— end example] [Note: a pointer to member formed from a mutable non-static data member (7.1.1) does not reflect the mutable specifier associated with the non-static data member. — end note]*

Change 10.2 [class.member.lookup] paragraph 13 as follows:

> *[Note: Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.11 [conv.mem], 5.2.5 [expr.ref], 5.3.1 [expr.unary.op], 11.2 [class.access.base]). —end note]...*

References

[1] “Type of address-of-member expression”, Lisa Lippincott
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed#203

[2] N3322 “A preliminary proposal for a static if”, Walter Brown
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3322.pdf

[3] N3329 “Proposal: static if declaration”, Walter Bright, Herb Sutter, Andrei Alexandrescu
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

[4] Bloomberg Basic Standard Library (BSL)
https://github.com/bloomberg/bsl

[5] “Uses and abuses of inheritance, part 1”, Herb Sutter
http://www.gotw.ca/publications/mill06.htm

[6] N3525 “Polymorphic Allocators”, Pablo Halpern
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf

[7] “Protected access and pointers to members”, Johannes Schaub,
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed.html#1007

[8] “Unnecessary ambiguity error in formation of pointer to member”,
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1121