Document Number: P0312R0
Date: 2016-03-20
Audience: Evolution Working Group
Reply-To: Barry Revzin
barry.revzin@gmail.com

Make Pointers to Members Callable

Introduction

Member functions play an important role in C++. However, pointers to members are still second-class citizens to pointers to non-members. They're *INVOKE*-able, but not actually invokable. As a result, lots of code exists to handle this exceptional case. But there's nothing exceptional about pointers to member functions. Or at least, there shouldn't be.

Motivation

In Item 41 of Effective STL, Scott Meyers summarizes a key difference between pointers to member functions and pointers to functions:

> Now, suppose I have a function that can test `Widgets`,
> ```
> void test(Widget&);
> ```
> and I have a container of `Widgets`:
> ```
> vector<Widget> vw;
> ```
> To test every `Widget` in `vw`, I can use `for_each` in the obvious manner:
> ```
> for_each(vw.begin(), vw.end(), test); // Call #1 (compiles)
> ```
> But imagine that test is a member function of `Widget` instead of a non-member function, i.e., that `Widget` supports self-testing:
> ```
> class Widget {
> public:
>     …
>     void test();
>     …
> };
> ```
> In a perfect world, I'd also be able to use `for_each` to invoke `Widget::test` on each object in vw:
> ```
> for_each(vw.begin(), vw.end(), &Widget::test); // Call #2 (won't compile)
> ```
> In fact, if the world were really perfect, I'd be able to use `for_each` to invoke `Widget::test` on a container of `Widget*` pointers, too:
> ```
> list<Widget*> lpw;
> for_each(lpw.begin(), lpw.end(), &Widget::test); // Call #3 (also won't compile)
> ```

There exist library mechanisms to still get these calls to work (namely `std::mem_fn`), but that is not a "perfect world." There are four categories of types that can be invokable: functions, pointers to functions, function objects, and pointers to members. The *INVOKE* concept includes all of these, so `std::function`, `std::bind`, and `std::invoke` work transparently with any of them. But consider the different syntaxes that a programmer must use in all four cases:

```
void fun(Widget& );
void (*pfun)(Widget& ) = fun;
struct Object {
    void operator()(Widget& );
} fun_obj;
void (Widget::*pmem)() = &Widget::test;

Widget w;

fun(w);
pfun(w);
fun_obj(w);
(w.*pmem)(); // huh?
```

The first three are identical, but that last one is jarringly different. It's a syntax that is difficult for beginners to grok and even experienced C++ programmers find it cryptic. And it's that difference in syntax that breaks Scott Meyers' examples: the algorithmic function templates work with functions, pointers to functions and arbitrary function objects ([function.objects]), but not with pointers to members.

In Eric Niebler's range-v3 library, the predicates and functions passed to algorithms are wrapped in the library side so that the difference is transparent to the user and anything can be passed in. But this simply shifts the burden from the user to the library developer to handle this case separately. It would be far simpler to just do away with the special case entirely.

Proposal

Rather than either placing the burden on the user to know how to handle pointers to member functions, or placing the burden on the library implementers to take care of everything for the user, or suggesting a rewrite of every template in `<algorithm>`, this paper is simply proposing to support the "normal" call syntax for pointers to member functions. That is:

```
X x;
X* px = &x;
void (X::*p_mem)(int ) = …;
p_mem(x, 1);  // equivalent to (x.*p_mem)(1)
p_mem(px, 1); // equivalent to (px->*pmem)(1) or ((*px).pmem)(1)
```

The same would apply to pointers to member data:

```
struct Y { int val; };
int (Y::*p_val) = &Y::val;
Y y{42};
p_val(y) = 17;           // equivalent to (y.*p_val) = 17;
p_val(&y) = 17;          // same as above
```

This syntax is natural and intuitive. You already know how to interpret it correctly. There are no surprises here. The call syntax can also support *cv*- and reference-qualified member functions, conditioned on the first argument being qualified according to what the member function requires. The following cases should all compile or not compile as expected:

```
void (Z::*lval)() & = …;
void (Z::*rval)() && = …;
void (Z::*const_)() const = …;

Z z;
const Z cz;

lval(z);    // OK
lval(&z);   // OK
lval(Z{});  // error: Z{} is not an lvalue
rval(z);    // error: z is not an rvalue
rval(Z{});  // OK
const_(cz); // OK
lval(cz);   // error: can't call non-const member function on const object
```

This proposal is focused solely on allowing pointers to member functions to be used as operands in a function call expression. It is not proposing to make the various function pointers convertible to each other. The two will remain unconvertible to each other:

```
int (*p)(X*, int) = non_member; // OK
int (X::*q)(int) = &X::member;  // OK
int (*r)(X*, int) = &X::member; // still an error
int (X::*s)(int) = non_member;  // still an error
```

Note that this proposal is not related to the unified call syntax proposal. The latter proposal is about supporting the syntax `member(x, a1, …, aN)` – calling member functions by name using the non-member call syntax. This proposal is about calling pointers to members via `pmf(x, a1, …, aN)`.

Impact

In most cases, the adoption of this proposal will not affect existing code. This paper is not proposing to remove the current pointer-to-member call syntax. The implementations of the aforementioned `std::function` and `std::bind()` could change to be simpler (but don't have to). `std::mem_fn()` and `std::invoke()` would effectively become redunant (but can still be used).

The value of this proposal would be the syntax becomes more regular, more predictable, more uniform, and considerably less confusing. It will be easier for beginners to understand how to use pointers to members. The standard algorithms will become easier to use as now pointers to members can be passed in without decoration, and this code will be easier to read as it will become less cluttered with `mem_fn`'s.

The cost would be there may be examples using SFINAE that would break as a result of this proposal. One example might be having two overloads taking an object, one enabled if the object is callable and the other if the object is a member pointer. These two conditions were previously disjoint, but now would become overlapping and thus ambiguous. This paper suggests that the breakages, such as they are (and there may not be many), are worth it in order to simplify the language rules.

Prior Art

As pointed out by Mr. Meredith, this idea was previously proposed by Peter Dimov in N1695[1]. This proposal however predates the committee recording discussion and the precise reason why it was rejected is unclear. This paper believes that paper was a good idea eleven years ago and continues to be a good idea today.

Acknowledgements

Thanks to Eugene Yakubovich, Jonathan Wakely, Robert Douglas, Alisdair Meredith, Thiago Macieira, Hyman Rosen, Vicente J. Botet Escriba, T.C., and Daniel Krügler for focusing and improving this proposal.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1695.html