Removing exception in precedence rule(s) when using member pointer syntax

Document #:	P2904R0
Date:	2023-05-27
Project:	Programming Language C++
Audience:	Evolution Working Group

Reply-to:	Anoop Rana <ranaanoop986@gmail.com>

1 Introduction

This paper proposes that [expr.unary.op#4] be amended to allow the usage of the syntax `&(C::Bar)` for some class `C` with member function `Bar`.

The standard makes an exception to precedence rules when it comes to member function pointer syntax. For example, in the below shown program, the syntax `&(C::Bar)` is explicitly forbidden(disallowed) by [expr.unary.op#4]. Is there a reason why the normal operator precedence rules are not applied here. I mean I would expect that `&C::Bar` is grouped as(or equivalent to writing) `&(C::Bar)` and so the latter should be allowed(since the former is allowed and since they are equivalent in form). There doesn't seem to be a good reason to disallow the latter by making an exception to the normal precedence rules.

This exception also makes the language(this part of the language to be specific) little inconsistent and unintuitive in my understanding. Here is the example for which we notice implementation divergence:

```
struct C
{
    void Bar(int);
};
int main()
{
    void (C::*ptr)(int) = &(C::Bar); //compiles with msvc but both gcc and clang rejects this
}
```

2 Motivation and Scope

This would make this part of the language more consistent and intuitive. Since there doesn't seem to be a good motivation for having this exception in the normal precedence rules, I suggest this to be removed so that `&(C::Bar)` is well-formed and work just like `&C::Bar`.

3 Impact on the Standard

Again, this will make previously invalid program(s) involving `&(C::Bar)` to be well-formed.

4 Change in wording

Change [expr.unary.op#4] to as shown below:

> A pointer to member is ~~only~~ formed when an explicit `&` is used and its operand is a qualified-id ~~not enclosed in parentheses~~.
>
> [Note 4: ~~That is, the expression `&(qualified-id)`, where the qualified-id is enclosed in parentheses, does not form an expression of type *pointer to member*. Neither does~~. The expression `qualified-id` does not form an expression of type pointer to member, because there is no implicit conversion from a qualified-id for a non-static member function to the type "pointer to member function" as there is from an lvalue of function type to the type "pointer to function" ([conv.func]). ~~Nor~~ Neither is `&unqualified-id` a pointer to member, even within the scope of the unqualified-id's class. — end note]