Reference-Qualified Member Functions

Qualifying a nonstatic **member function** with either an `&` or `&&` refines its **signature** based on the **value category** — i.e., *lvalue* or *rvalue*, respectively — of the **expression** used to evoke it, thus enabling two distinct **overloaded** implementations of that member function.

Description

C++ has always supported decorating nonstatic member functions with **cv-qualifiers** and allowed overloading on those **qualifiers**:

```
struct Class1
{
    void mf1() const;     // (1) const-qualified member function
    void mf2();           // (2) member function with no qualifiers
    void mf2() volatile;  // (3) volatile-qualified overload of (2)
};
void f1()
{
             Class1 uobj;
       const Class1 cobj;
    volatile Class1 vobj;
    uobj.mf1();  // calls function (1)
    cobj.mf1();  // calls function (1)
    uobj.mf2();  // calls overloaded function (2)
    vobj.mf2();  // calls overloaded function (3)
    vobj.mf1();  // Error, no mf1 overload matches a volatile object.
    cobj.mf2();  // Error,  " mf2     "       "    " const       "
}
```

The cv-qualifiers, **const** and **volatile**, optionally appearing after the **parameter list** of a nonstatic member function prototype apply to the object on which the **member** is called and allow us to overload on the **cv-qualification** of that object. **Overload resolution** will select the closest match whose cv-qualifiers are the same as, or more restrictive than, the object's cv-qualification; hence, `uobj.mf1()` calls a **const**-qualified member even though `vu` is not **const**. A qualifier cannot be dropped during overload resolution, however, so `vobj.mf1()` and `cobj.mf2()` are **ill formed**.

C++11 introduced a similar feature, adding optional qualifiers that indicate the valid value categories for the expression a member function may be invoked on. **Declaring** a member function overload specifically for *rvalue* expressions, for example, allows library writers to make better use of **move semantics**. Note that readers of this feature are presumed to be familiar with value categories and, in particular, the distinction between **lvalue** and **rvalue references** (see Section 2.1."*Rvalue* References" on page 710):