```
    auto c6 = [&, b]{ return a * b; };
        // a is implicitly captured by reference, and b is explicitly
        // captured by copy.
    auto c7 = [=, &b]{ return a * b; };
        // a is implicitly captured by copy, and b is explicitly
        // captured by reference.
    auto c8 = [a]{ return a * b; };
        // Error, a is explicitly captured by copy, but b is not captured.
}
```

When a lambda expression appears within a nonstatic member function, the **this** pointer can be captured as a special case:

```
class Class1
{
public:
    void mf()
    {
        auto c12 = [this]{ return this; };  // Explicitly capture this.
        auto c13 = [=]   { return this; };  // Implicitly capture this.
    }
};
```

Both implicit and explicit capture of **this** capture the pointer value of **this** and do not make a copy of the object pointed to by **this**. Redundant captures are not allowed; the same name (or **this**) cannot appear twice in the lambda capture. Moreover, if the capture default is `&`, then none of the explicitly captured variables may be captured by reference, and if the capture default is `=`, then any explicitly captured entities can be neither captured by copy nor **this**[4]:

```
class Class2
{
public:
    void mf()
    {
        int a = 0;
        auto c1 = [a, &a]{ /*...*/ }; // Error, a is captured twice.
        auto c2 = [=, a]{ /*...*/ };
            // Error, explicit capture of a by copy is redundant.
        auto c3 = [&,&a]{ /*...*/ };
            // Error, explicit capture of a by reference is redundant.
        auto c4 = [=, this]{ return this; };
            // Error, explicit capture of this with = capture default
    }
};
```

[4]C++20 removed the prohibition on explicit capture of **this** with an `=` capture default. In fact, C++20 deprecated implicit capture of **this** when the capture default is `=` and instead requires `[=, this]` to capture **this** in such situations.