Capturing [*this] in Lambda Expressions

Let's look at how to capture [*this] in Lambda Expressions.

When you write a lambda inside a class method, you can reference a member variable by capturing `this`. For example:

```
#include <iostream>
#include <string>

struct Test  {
    void foo() {
        std::cout << m_str  << '\n';
        auto addWordLambda = [this]() { m_str += "World"; };
        addWordLambda ();
        std::cout << m_str  << '\n';
    }

    std::string m_str {"Hello "};
};
int main() {
    Test test;
    test.foo();

    return 0;
}
```

In the line with `auto addWordLambda = [this]() {... }` we capture `this` pointer and later we can access `m_str`.

Please notice that we captured `this` by value… to a pointer. You have access to the member variable, not its copy. The same effect happens when you capture by `[=]` or `[&]`. That's why when you call `foo()` on some `Test` object then you'll see the following output:

```
Hello
Hello World
```

`foo()` prints `m_str` two times. The first time we see `"Hello"`, but the next time foo() prints m_str two times. The first time we see "Hello", but the next time it's "Hello World" because the lambda addWordLambda changed it.

Let's look at a more complicated case. Do you know what will happen with the following code?

```
#include <iostream>

struct Baz {
    auto foo() {
        return [=] { std::cout << s << '\n'; };
    }
    std::string s;
};

int main() {
   auto f1 = Baz{"ala"}.foo();
   f1();
}
```

The code declares a Baz object and then invokes foo(). Please note that foo() returns a lambda that captures a member of the class.

A capturing block like [=] suggests that we capture variables by value, but if you access members of a class in a lambda expression, then it does this implicitly via the this pointer. So we captured a copy of this pointer, which is a dangling pointer as soon as we exceed the lifetime of the Baz object.

In C++17 you can write: [*this] and that will capture **a copy** of the whole object.

```
auto lam = [*this]() { std::cout << s; };
```

In C++14, the only way to make the code safer is init capture *this:

```
auto lam = [self=*this] { std::cout << self.s; };
```

Capturing this might get tricky when a lambda can outlive the object itself. This might happen when you use async calls or multithreading.

In C++20 (see P0806) you'll also see an extra warning if you capture [=] in a method. Such expression captures the this pointer, and it might not be exactly what you want.

exactly what you want.

> *Extra Info:* The change was proposed in: P0018.

Another step towards cleaner code in C++ is the way it now allows nested namespaces. We will examine that next.