The std::invoke function does more than invoke functions

devblogs.microsoft.com/oldnewthing/20220401-00

April 1, 2022

Raymond Chen

The `std::invoke` function in the C++ standard library is usually used to call a functor with parameters.

```
std::function<void(int)> func = ...;

// same as func(42)
std::invoke(func, 42);
```

What `std::invoke` brings to the table is that you can use it for other things beyond just functors.

```
struct S
{
    void do_something(int);
    int v;
};

S s;
// same as s.do_something(42)
std::invoke(&S::do_something, s, 42);
std::invoke(&S::do_something, std::ref(s), 42);

S* p = &s;
// same as p->do_something(42)
std::invoke(&S::do_something, p, 42);
```

But wait, what about this?

```
struct S
{
    std::function<void()> do_something;
    int v;
};

S s;
s.do_something = []() { std::cout << "hello"; };

// does not print anything
std::invoke(&S::do_something, s);
```

What's going on here?

One thing that often goes overlooked is that you can also use `std::invoke` with pointers to non-static data members.

```
S s;

// same as s.v = 42
std::invoke(&S::v, s) = 42;

// same as "auto x = s.v;"
auto x = std::invoke(&S::v, s);
```

Invoking a pointer to a non-static data member is the same as dereferencing the pointer, when applied to the second argument.

The statement

```
std::invoke(&S::do_something, s);
```

is therefore equivalent to

```
s.do_something;
```

which, despite its name, does nothing: It accesses the member and throws it away.

If you want to access the memory and then invoke it, you'll have to follow up the `std::invoke` with the function call.

```
std::invoke(&S::do_something, s)();
```

Or, if you really like to show off,

```
std::invoke(std::invoke(&S::do_something, s));
```

Taken to an extreme, you get invoke-oriented programming!

```
// Old and busted
this->dict.find(3)->second = "meow";

// New hotness
std::invoke(
    static_cast<std::map<int, std::string>::iterator
        (std::map<int, std::string>::*)(int const&)>(
        &std::map<int, std::string>::find),
        std::invoke(&MyClass::dict, this), 3)->second = "meow";

// Beyond hot
std::invoke(
    static_cast<std::string& (std::string::*)(char const*)>
        (&std::string::operator=),
    std::invoke(&std::pair<int const, std::string>::second,
        std::invoke(
            static_cast<std::pair<int const, std::string>& (
                std::map<int, std::string>::iterator::*)() const noexcept>
                (&std::map<int, std::string>::iterator::operator*),
        std::invoke(
            static_cast<std::map<int, std::string>::iterator
                (std::map<int, std::string>::*)(int const&)>
                (&std::map<int, std::string>::find),
            std::invoke(&MyClass::dict, this), 3))), "meow");
```

The above code is technically non-portable thanks to [member.functions], which says

> For a non-virtual member function described in the C++ standard library, an implementation may declare a different set of member function signatures, provided that any call to the member function that would select an overload from the set of declarations described in this document behaves as if that overload were selected.

This means basically that you cannot form pointers to non-virtual member functions, because the implementation's signature for the member function is permitted to differ from the formal definition (say, by the addition of default template arguments or parameters), as long as the behavior is the same. In practice, these extra default arguments or parameters are used for things like SFINAE.

To make the code portable, we'll have to wrap the member function pointers into program-provided versions.

```
namespace mfptr
{
    template<typename Object, typename...Args>
    decltype(auto) find(Object&& object, Args&&...args) {
        return std::forward<Object>(object).find(std::forward<Args>(args)...);
    }

    template<typename Object>
    decltype(auto) dereference(Object&& object) {
        return *std::forward<Object>(object);
    }

    template<typename Object, typename Arg>
    decltype(auto) assign(Object&& object, Arg&& arg) {
        return std::forward<Object>(object) = arg;
    }
}

std::invoke(
    &mfptr::assign<std::string&, char const*>,
    std::invoke(&std::pair<int const, std::string>::second,
        std::invoke(
            &mfptr::dereference<std::map<int, std::string>::iterator>,
            std::invoke(
                &mfptr::find<std::map<int, std::string>&, int>,
                std::invoke(&MyClass::dict, this), 3))), "meow");
```

Raymond Chen

Follow