How can I prevent myself from accessing a lambda captured variable or a parameter after I'm done with it?

devblogs.microsoft.com/oldnewthing/20211104-00

November 4, 2021

Raymond Chen

We saw last time that you can have your cake and eat it too: You can have a capturing lambda coroutine provided you copy the captures to the coroutine frame before the coroutine's first suspension point. From that point onward, you use the copies in the frame instead of the original captured variables, because the original captured variables may no longer exist!

You therefore must exercise willpower to resist the temptation to access those captured variables, since they are now poison. Is there a way to help stop yourself from doing a bad thing? Like, put a "Do not touch" sticker on the captured variable?

We can do this by taking advantage of *name hiding*, which also goes by the name *shadowing*. Name hiding is normally something you are scared of, because it often happens by mistake:

```
int open_or_alt(char const* name, char const* altname)
{
    int fd = open(name, 0);
    if (fd == -1 && errno == ENOENT) {
        // accidentally shadowed outer "fd" variable
        int fd = open(altname, 0);
    }
    return fd;
}
```

In our case, we can use it on purpose. Suppose we define some type, call it `hide_name`, and we use it to hide the names of things we don't want to see any more.

```
void RegisterClickHandler(Button const& button, int key)
{
    button.Click([key](auto&&, auto&&)
        -> winrt::fire_and_forget
        {
            auto copiedKey = key;
            hide_name key; // "key is now dead to me"
            co_await winrt::resume_background();
            NotifyClick(key); // want this to be an error
        });
}
```

What we're doing is declaring a local variable named `key` which hides the captured variable with the same name. Any attempt to use `key` after the point of declaration will use the local variable version and not the captured variable version. Let's assume that `hide_name` is carefully defined so that using it instead of the hidden variable has maximum likelihood of generating a compiler error. (We'll look at how we can do this later.)

You can also hide parameters, which is handy if you have a coroutine that must accept parameters by reference (say, for some compatibility reason), but needs to access them after a suspension point.

```
// This used to refresh synchronously, but now we refresh
// asynchronously. The parameter is still passed by reference
// for compatibility.
winrt::fire_and_forget Refresh(winrt::com_ptr<Widget> const& widget)
{
    auto copiedWidget = widget;
    hide_name widget;
    co_await winrt::resume_background();
    copiedWidget->Reset();
    copiedWidget->Reload();
}
```

Note that you can hide names from outer scopes. You cannot hide a name from the same scope.

```
winrt::fire_and_forget GiveAwayTheWidget()
{
    Widget widget;
    widget.Initialize();
    GiveAway(std::move(widget));

    // Try to prevent accidentally using a moved-from variable.
    hide_name widget; // error

    ...
}
```

Since `hide_name blah;` is a declaration, you can hide multiple names by separate them with commas.

```
hide_name a, b, c; // all three names are hidden
```

Okay, so the big question is "What should `hide_name` be in order to maximize the likelihood that using it will result in a compiler error?"

I came up with this:

```
using hide_name = void(struct hidden_name);
```

This says that `hide_name` is an alias for a function that accepts an incomplete type called `hidden_name` and which returns a `void`. When you write

```
hide_name a;
```

what you're really saying is "`a` is a function that accepts an object of type `hidden_name` and which returns `void`." It's just a function declaration, not a variable declaration. But function names are still names, so they participate in name hiding.

Using the name `a` by itself will decay to a function pointer, which is not convertible to anything interesting. In particular, you can't even do this:

```
void* p = &a;
```

Function pointers are not compatible with data pointers, so this catches the case where somebody accidentally tried to pass the address of a captured variable to something that accepts a `void*`.

You can't do arithmetic on function pointers, so those will generate errors. And you can't call the function pointer because its sole parameter is `hidden_name` which is an incomplete type.

Functions cannot be assigned to, so that blocks any attempt to write to `a`.

If somehow somebody gets past all these barriers (say by explicitly casting to `uintptr_t` on an implementation that supports it), they will fail at link time with an unresolved external complaining that there is no function named `a` that accepts a `hidden_name` as a parameter. (No such function can exist because `hidden_name` is an incomplete type.)

Here's what I found in my experiments:

Accessing the variable in various ways.

```
void test(int);
int i;
hide_name a;

// All accept this. "v" is a pointer to function.
// You will probably get errors when you try to use "v", though.
auto v = a;

// msvc: C2440: cannot convert from 'hide_name (__cdecl *)' to 'int'
// gcc: invalid conversion from 'void (*)(hidden_name)' to 'int' in assignment
// clang: assigning to 'int' from incompatible type 'hide_name'
// icc: a value of type "hide_name *" cannot be assigned to an entity of type "int"
i = a;

// msvc: C2659: '=': function as left operand
// gcc: assignment of function 'void a(hidden_name)'
// clang: non-object type 'hide_name' is not assignable
// icc: expression must be a modifiable lvalue
a = i;

// msvc: cannot convert argument 1 from 'hide_name (__cdecl *)' to 'int'
// gcc: invalid conversion from 'void (*)(hidden_name)' to 'int'
// clang: no matching function for call to 'test'
// icc: argument of type "hide_name *" is incompatible with parameter of type "int"
test(a);

// msvc: C2296: '+': illegal, left operand has type 'hide_name (__cdecl *)'
// gcc: pointer to a function used in arithmetic
// clang: arithmetic on a pointer to the function type 'hide_name'
// icc: a value of type "hide_name *" cannot be assigned to an entity of type "int"
i = a + 1;

// msvc: C2296: '+': illegal, left operand has type 'hide_name (__cdecl *)'
// gcc: pointer to a function used in arithmetic
// clang: arithmetic on a pointer to the function type 'hide_name'
// icc: NO ERROR AT COMPILE TIME! (Fails to link.)[1]
auto p = a + 1;

// msvc: C2070: 'hide_name': illegal sizeof operand
// gcc: ISO C++ forbids applying 'sizeof' to an expression of function type[2]
// clang: invalid application of 'sizeof' to a function type
// icc: warning: operand of sizeof may not be a function
auto size = sizeof(a);

// msvc: C228: left of '.method' must have class/struct/union
// gcc: request for member 'method' in 'a', which is of non-class type 'hide_name'
// clang: member reference base type 'hide_name' is not a structure or union
// icc: expression must have class type
a.method();
```

Furthermore, since this is a function declaration and not a variable declaration, you are less likely to get "unused local variable" errors. gcc, clang, and icc ignore the unused function declaration.

Unfortunately, the place where this trick is most useful is in coroutines, and that's where compilers today are the weakest. MSVC 19.28 was okay with it, but it seems that MSVC 19.29 does complain about unused function declarations in coroutines. As of this writing, a fix is pending release. [**Update February 15, 2022**: It is fixed in Visual Studio 2022 Version 17.1.] gcc, on the other hand, totally freaks out about function declarations in coroutines: Version 11 produces the inscrutable error "data member 'enclosing_function()::_Z18enclosing_functionv.frame::__a.2.3' invalid declared function." Version 12 blows up with an internal compiler error.

So maybe you should hold off on this for a bit longer.

[1] For some reason, icc treats the size of a function as 1. Sometimes it warns you that it's making this nonstandard decision. Other times, it just does it without telling you.

[2] I find it amusing that one of the gcc error messages says "ISO C++ forbids…", as if it's saying "Hey, I totally would let you get away with this, but those pesky C++ standard folks tell me I'm not allowed to. So don't blame me!"

Raymond Chen

Follow