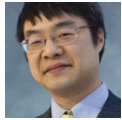


How can I have a C++ function that returns different types depending on what the caller wants?

 devblogs.microsoft.com/oldnewthing/20191106-00

November 6, 2019



Raymond Chen

Here's something crazy: You have a function that has two different callers. One of them expects the function to return a widget. The other expects the function to return a doodad.

```
class widget;
class doodad;

class experiment
{
public:
    doodad get_entity();
    widget get_entity();
};
```

This is not allowed in C++. The return type of a function cannot be overloaded.

But Kenny Kerr taught me how to fake it. What you do is return an object that doesn't yet know whether it's a widget or doodad.

```

class experiment
{
public:
    auto get_entity()
    {
        struct result
        {
            operator widget()
            {
                return experiment->get_entity_as_widget();
            }

            operator doodad()
            {
                return experiment->get_entity_as_doodad();
            }

            experiment* experiment;
        };
        return result{ this };
    }
};

```

The thing that is returned is neither a widget nor a doodad, but observing it will trigger a collapse to one or the other.

```

widget w = exp.get_entity();
doodad p = exp.get_entity();

```

In the first call, the `get_entity()` returns the private `result` type, and then immediately assigns it to a variable of type `widget`. This triggers the `operator widget()` conversion operator, which calls `get_entity_as_widget`.

Similarly, the second call obtains the private `result` type and converts it to a `doodad`, which winds up calling `get_entity_as_doodad`.

The wave function collapse could be triggered by anything that accepts a conversion.

```

move_widget(exp.get_entity()); // will call get_entity_as_widget
signal_doodad(exp.get_entity()); // will call get_entity_as_doodad

```

If you take the return value of `get_entity` and save it in an `auto`, then the wave function hasn't collapsed yet. It's still not sure which thing it is.

```

auto entity = exp.get_entity();

```

The thing doesn't become a widget or doodad until you convert it.

```

move_widget(entity); // calls get_entity_as_widget

```

Note that the call to `get_entity_as_widget` is delayed until the conversion occurs.

```
auto entity = exp.get_entity();  
exp.replace_entity();  
move_widget(entity); // calls get_entity_as_widget
```

Between calling `get_entity` and converting the result to a widget, we changed the entity in the experiment. Not until the conversion occurs does the call to `get_entity_as_widget` happen, at which point it will get the new entity rather than the original one. And of course, if you destroy the experiment, then the unresolved `entity` has a dangling pointer, and the behavior is undefined.

This trick works best if the caller will convert the result *immediately* to its final type (widget or doodad).

Of course, you could try to fix these problems, say by taking a strong reference to the `experiment` to prevent it from being destructed prematurely. Or you could call both `get_entity_as_widget` and `get_entity_as_doodad` as part of the constructor, and then hand out the appropriate type during the conversion. That would fix the “delayed evaluation” problem, but at a cost of doing eager evaluation of both branches, even if only one will end up being used.

In the case where Kenny used it, it was to permit the `First` method to return a different iterator depending on who’s asking for it. The underlying problem is the object wants to be able to produce a stream of `T` objects or `IInspectable` objects, so it implements both the `IIterable<T>::First()` and `IIterable<IInspectable>::First()` methods. The projection for those interfaces forward to the implementation’s `First()`, which forces `First()` to serve two masters. And the way he solved it was to return an ambiguous object, so that each master sees what it wants.

Raymond Chen

Follow

