In C++, how can I make a default parameter be the this pointer of the caller?, revisited

devblogs.microsoft.com/oldnewthing/20241122-00

November 22, 2024

Some time ago, we looked at making the default parameter of a method be the `this` pointer of the caller. The scenario was something like this:

```
struct Property
{
    Property(char const* name, int initial, Object* owner) :
        m_name(name), m_value(initial), m_owner(owner) {}

    ⟦ other methods elided - use your imagination ⟧

    char const* m_name;
    Object* m_owner;
    int m_value;
};

struct Widget : Object
{
    Property Height{ "Height", 10, this };
    Property Width{ "Width", 10, this };
};
```

and we didn't want to have to type `this` as the last parameter to all the `Property` constructors. We came up with this:

```
template<typename D>
struct PropertyHelper
{
    Property Prop(char const* name, int initial)
    { return Property(name, initial, static_cast<D*>(this)); }
};

struct Widget : Object, PropertyHelper<Widget>
{
    Property Height = Prop("Height", 10);
    Property Width = Prop("Width", 10);
};
```

Or, if you have access to deducing this,

```
struct PropertyHelper
{
    template<typename Parent>
    Property Prop(this Parent&& parent, char const* name, int initial)
    { return Property(name, initial, &parent); }
};

struct Widget : Object, PropertyHelper
{
    Property Height = Prop("Height", 10);
    Property Width = Prop("Width", 10);
};
```

But this uses a fixed parameter list. What if you have to deal with multiple kinds of properties?

```
template<typename T>
struct Property
{
    Property(char const* name, T const& initial, Object* owner) :
        m_name(name), m_value(initial), m_owner(owner) {}

    ⟦ other methods elided - use your imagination ⟧

    char const* m_name;
    Object* m_owner;
    T m_value;
};
```

or arbitrary types, not just specializations of `Property`?

```
template<typename Handler>
struct Event
{
    Event(char const* name, Object* owner) :
        m_name(name), m_owner(owner) {}

    ⟦ other methods elided - use your imagination ⟧

    char const* m_name;
    Object* m_owner;
    std::vector<Handler> m_handlers;
};
```

What all of these little classes have in common is that they take a pointer to the containing class as the final parameter. Can we generalize this solution without having to create a bunch of one-off classes, one for each type we need to wrap?

Sure. The idea is to use a trick we saw a little while ago: You can simulate a function that returns different types depending on what the caller wants by returning a proxy type that holds onto the parameters and then performs the work when the destination type is revealed by the conversion operator.

```
template<typename...Args>
struct maker
{
    std::tuple<Args&&...> m_args;

    maker(Args&&... args) : m_args((Args&&)args...) {}

    template<typename T>
    operator T() {
        return std::make_from_tuple<T>(std::move(m_args));
    }
};

template<typename D>
struct OwnerHelper
{
    template<typename...Args>
    auto Owned(Args&&... args)
    { return maker<Args&&..., D*>((Args&&)args..., static_cast<D*>(this)); }
};

struct Widget : Object, OwnerHelper<Widget>
{
    Property<int> Height = Owned("Height", 10);
    Property<std::string> Name = Owned("Name", ""s);
    Event<NameChangedHandler> NameChanged = Owned("NameChanged");
};
```

There is a subtlety here: We need to cast `args` to ensure that its reference category is preserved.

Note that the use of a parameter pack means that we have to write something like `""s` or `std::string{}` to get an empty string, rather than the more convenient `{}` because template type parameters match types, and not braced things that could be used to construct a type but aren't a type themselves.

As before, we can simplify this with *deducing this*:

```
struct OwnerHelper
{
    template<typename O, typename...Args>
    auto Owned(this O&& self, Args&&... args)
    {
        return maker<Args&&..., O*>((Args&&)args..., &self);
    }
};

struct Widget : Object, OwnerHelper
{
    Property<int> Height = Owned("Height", 10);
    Property<std::string> Name = Owned("Name", ""s);
    Event<NameChangedHandler> NameChanged = Owned("NameChanged");
};
```