Trying to create a factory that remembers the parameters to pass to another method

devblogs.microsoft.com/oldnewthing/20220422-00

April 22, 2022

Raymond Chen

Continuing from using maker functions to work around limitations of class template argument deduction, let's suppose you want to create a generic factory. This sounds like an arbitrary academic exercise, but this stemmed from a real problem. This has practical use, say, if you have a common object and need to generate various factories that produce preconfigured objects.

```
// Imaginary code that doesn't even compile.

template<typename T, typename...Args>
struct generic_factory
{
    template<typename... Actuals>
    generic_factory(Actuals&&... args) : /* something */ { }

    auto make()
    {
        return std::make_unique<T>(args...);
    }
};

template<typename T, typename... Args>
auto make_generic_factory(Args&&... args)
{
    return generic_factory<T, std::decay_t<Args>...>
        (std::forward<Args>(args)...);
}
```

The idea is that you create the generic factory by telling it what type you want to create (`T`) and the arguments to pass to the constructor (`args...`). You can then call the `make` method on the generic factory, and out comes a new `T` object, constructed with exactly those parameters.

The question is how to convert this sketch into a real class.

A handy place to store a bunch of values is a tuple, and getting the values out of a tuple to pass them as parameters can be done with `std::apply`: While we're at it, we'll move the `std::decay` into the `generic_factory`, to make things a little less awkward.

```
// Compiles but doesn't work.

template<typename T, typename...Args>
struct generic_factory
{
    using Tuple = std::tuple<std::decay_t<Args>...>;
    Tuple captured;

    generic_factory(Args&&... args) :
        captured(std::forward<Args>(args)...) {}

    auto make()
    {
        return std::apply(std::make_unique<T>, captured);
    }
};

template<typename T, typename...Args>
auto make_generic_factory(Args&&... args)
{
    return generic_factory<T, Args...>(std::forward<Args>(args)...);
}
```

Now, it looks like we're reinventing `std::bind`, and yes, that's very similar what we're doing. We don't have the special rules about `std::ref` and `std::cref` that `std::bind` has, nor do our parameters decay, nor do we support placeholders. But one annoying thing about `std::bind` is that the return type *has no name*, so it's hard to store it in a variable for later use. (I guess what you usually do is immediately put it inside a `std::function`, but that comes with its own issues.)

And it turns out that `std::bind` has the same problem that our `make` does:

```
    // with std::bind
    auto make = std::bind(std::make_unique<T>, args...);
    make();

    // with std::apply
    return std::apply(std::make_unique<T>, captured);
```

Both of these fail with some horrible error message:

```
// std::bind
Failed to specialize function template 'unknown-type std::_Binder<std::_Unforced,
std::unique_ptr<T, std::default_delete<T>> (__cdecl &)(void),T>::operator ()(_Unbound
&&...) noexcept() const'

// std::apply
'std::invoke': no matching overloaded function found
see reference to function template instantiation 'decltype(auto)
std::_Apply_impl<std::unique_ptr<T, std::default_delete<T>> (__cdecl &)
(void),std::tuple<Args...>&,0>(std::unique_ptr<T,std::default_delete<T>> (__cdecl &)
(void), std::tuple<int> &, std::integer_sequence<size_t, 0>)' being compiled
```

I find it interesting that calling `std::bind` does compile. However, it produces an object that cannot be used for anything: Trying to invoke the bound call generates the compiler error.

These calls fail because the first parameter to `std::bind` and `std::apply` is a *callable*. No overloading or template type inference is happening here: In order for those to occur, the expression needs to be cast to a specific type, or there need to be parameters to force a resolution to occur. But `std::bind` and `std::apply` accept their first parameters as an arbitrary type, so there is no coersion to any particular parameter list.

I mean, you and I know that the parameter is given by the remaining arguments, but that's only because we understand the semantics of what `std::bind` and `std::apply` are going to do, namely, combine the first parameter with the other parameters to create a function call. But the compiler doesn't know that. It just sees a bunch of parameters and doesn't know that they're going to be combined at some point in the future.

This means that when we write `std::make_unique<T>`, we are specifying the version of `make_unique` that takes no parameters. The parameters to `make_unique` correspond to the second and subsequent template type paramters, for which we passed none.

Since the compiler doesn't have enough information to infer the extra parameters, we have to specify them explicitly:

```
    // with std::bind
    auto make = std::bind(std::make_unique<T, Args...>, args...);
    make();

    // with std::apply
    return std::apply(std::make_unique<T, Args...>, captured);
```

Unfortunately, this still doesn't work. The confusing error message this time is

```
// std::bind
'operator __surrogate_func': no matching overloaded function found
Failed to specialize function template 'unknown-type std::_Binder<std::_Unforced,
std::unique_ptr<T, std::default_delete<T>> (__cdecl &)(void), T>::operator ()
(_Unbound &&...) noexcept() const'

// std::apply
'std::invoke': no matching overloaded function found
see reference to function template instantiation 'decltype(auto)
std::_Apply_impl<std::unique_ptr<T, std::default_delete<T>>(__cdecl &)(int &&),
std::tuple<int>&, 0>(std::unique_ptr<T, std::default_delete<T>> (__cdecl &)(int &&),
std::tuple<int>, std::integer_sequence<size_t, 0>)' being compiled
```

Buried in all that error message is the interesting part:

```
(__cdecl &)(int &&)
```

The specialization of `make_unique` we are calling wants an `int&&`. Why does it want an `int&&`?

The corresponding parameter is an `int&&` because the declaration of `make_unique` uses a universal reference:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args);
```

Since we explicitly passed `Args = int`, this makes the parameter list `make_unique(int&& args)`. And that’s why the function wants an `int&&`.

Okay, so we need to pass an `int&&`. But what does `std::apply` actually pass?

The `std::apply` function passes `std::get<N>(tuple)` for each parameter. Since the tuple we passed was an lvalue, `std::get<N>(tuple)` returns an lvalue reference to the tuple element.

And that’s where the error is coming from. The function wants an rvalue reference to `int`, but we’re passing an lvalue reference.

Before we try to solve this problem, we need to understand what we are trying to do.

We want to capture the parameters to pass to `make_unique`, and each time someone calls `make`, we want to call `make_unique` again with the same parameters. Therefore, we don’t want to pass rvalue references to our tuple elements: The first call to `make_unique` would be able to steal the resources from our captured parameters, leaving nothing for the the second and subsequent calls.

Similarly, we don't want to pass a straight lvalue reference, because that allows the `T` constructor to mutate the parameter, which would mess up the captured parameters for the second and subsequent calls.

What we really want to pass is a `const` lvalue reference. And we can make this easier to enforce by making our `captured` member variable also `const`.

```
template<typename T, typename...Args>
struct generic_factory
{
    using Tuple = std::tuple<std::decay_t<Args>...>;
    Tuple const captured;

    generic_factory(Args&&... args) :
        captured(std::forward<Args>(args)...) {}

    auto make()
    {
        return std::apply(std::make_unique<T,
            std::decay_t<Args> const&...>, captured);
    }
};

template<typename T, typename...Args>
auto make_generic_factory(Args&&... args)
{
    return generic_factory<T, Args...>(std::forward<Args>(args)...);
}
```

A note of caution here: The parameters passed to `make_generic_factory` are captured as-is. If constructing a `T` from them requires a parameter conversion, the conversion is applied at the time the `T` is constructed, not at the time the parameters are captured.

Here's an example:

```
struct widget
{
    widget(std::string const& name) : m_name(name) { }
    std::string m_name;
};

auto factory = make_generic_factory<widget>("bob");
```

The parameter captured by `make_generic_factory` is the string literal, not a `std::string`. Each time you call `make`, the string literal is converted to a `std::string`, which is then passed to the `widget` constructor, and then when the constructor returns, the temporary `std::string` is destructed. If you want to construct the `std::string` only once, you'll have to capture it as a `std::string`:

```
auto factory = make_generic_factory<widget>("bob"s);
```

This can get scary for some conversion constructors:

```
auto make_widget_factory(int id)
{
    char name[80];
    snprintf(name, 80, "item #%d", id);
    return make_generic_factory<widget>(name);
}
```

In this case, the captured parameter is the raw pointer to the stack buffer, which immediately goes out of scope. When you call `make()`, that raw pointer is then passed to `make_unique`, which will try to convert it to a `std::string`, but it's too late. The raw pointer is dangling.

Bonus chatter: A lambda would also do the trick. The `make` method becomes the `operator()`:

```
template<typename T, typename...Args>
auto make_generic_factory(Args&&... args)
{
    return [captured = std::make_tuple(args...)]() {
        return std::apply(std::make_unique<T,
            std::decay_t<Args> const&...>, captured);
    };
}
```

C++20 adds the ability to capture a parameter pack into a lambda without having to hide it inside a tuple:

```
template<typename T, typename...Args>
auto make_generic_factory(Args&&... args)
{
    return [...args = std::forward<Args>(args)]() {
        return std::make_unique<T>(args...);
    };
}
```

Raymond Chen

Follow