How can I explicitly specialize a templated C++ constructor?

devblogs.microsoft.com/oldnewthing/20241011-00

October 11, 2024

C++ allows constructors to be templated, but there is no syntax for explicitly specializing the constructor. Here's a rather artificial example:

```
// Assume derived classes by convention have a constructor
// whose first parameter is an ObjectManager&.
struct CommonBase
{
    virtual ~CommonBase(){}
    virtual void initialize(int reason) = 0;
};

struct ObjectManager
{
    // Concrete should derive from CommonBase
    template<typename Concrete, typename...Args>
    ObjectManager(int reason, Args&&...args) :
        m_base(std::make_unique<Concrete>(
                *this, std::forward<Args>(args)...))
    {
        m_base->initialize(reason);
    }

    std::unique_ptr<CommonBase> m_base;
};
```

The idea here is that you have some type, and you want to templatize the constructor. It is legal to have a templated constructor, but there is no way to explicitly specialize a constructor.

```
struct Widget : CommonBase
{
    Widget(int param);
    ⟦ ... ⟧
};

// This is not allowed[1]
auto manager = ObjectManager::ObjectManager<Widget>(42);
```

So how do you tell the constructor, “I want you to use this type for `Concrete`?”

Your only option is type inference, so you’ll have to make it inferrable from a parameter.

Enter `std::in_place_type` and friends.

We start with `std::in_place_type_t`, which is an empty type that takes a single type as a template parameter. You can use this as a dummy parameter and deduce the template type parameter from it.

```
struct ObjectManager
{
    // Concrete should derive from CommonBase
    template<typename Concrete, typename...Args>
    ObjectManager(int reason,
        std::in_place_type_t<Concrete>,
        Args&&...args) :
        m_base(std::make_unique<Concrete>(
                *this, std::forward<Args>(args)...))
    {
        m_base->initialize(reason);
    }

    std::unique_ptr<CommonBase> m_base;
};

// Example usage:
auto manager = ObjectManager(9, std::in_place_type_t<Derived>{}, 42);
```

The `in_place_type_t` is an empty class that is default-constructible. As a convenience, the standard library also defines a premade value:

```
template<T>
inline constexpr std::in_place_type_t in_place_type{};
```

Which lets you simplify the usage to

```
auto manager = ObjectManager(9, std::in_place_type<Derived>, 42);
```

Note that there is no member type `type` inside the `std::in_place_type_t`, so you have to use deduction to pull it out. You can’t say

```
// Concrete should derive from CommonBase
template<typename Trait, typename...Args>
ObjectManager(int reason,
    Trait,
    Args&&...args) :
    m_base(std::make_unique<typename Trait::type>(
            *this, std::forward<Args>(args)...))
{
    m_base->initialize(reason);
}
```

You might be tempted to use `std::type_identity`[2] as the type holder:

```
// Concrete should derive from CommonBase
template<typename Concrete, typename...Args>
ObjectManager(int reason,
    std::type_identity<Concrete>,
    Args&&...args) :
    m_base(std::make_unique<Concrete>(
            *this, std::forward<Args>(args)...))
{
    m_base->initialize(reason);
}
```

but that is not allowed.

According to the C++ standard, `std::type_identity` is a Cpp17TransformationTrait, and [meta.rqmts] spells out the requirements of various trait types in the standard library.

Trait	Constructible?	Copyable?	Special member
Cpp17UnaryTypeTrait	Yes	Yes	`value`
Cpp17BinaryTypeTrait	Yes	Yes	`value`
Cpp17TransformationTrait	No	No	`type`

Since a Cpp17TransformationTrait is not constructible, and the language does not provide any pre-made instances, there is no legal way of gaining access to an instance of a Cpp17TransformationTrait. An implemention would be within its rights to define `type_identity` as

```
template<typename T>
struct type_identity
{
    using type = T;

    // not constructible
    type_identity() = delete;

    // not copyable
    type_identity(type_identity const&) = delete;
}
```

[1] Another place you cannot specialize a templated function is operator overloading.

```
struct ObjectMaker
{
    ObjectMaker(std::string name) : m_name(std::move(name)) {}

    template<typename Concrete>
    Concrete operator()() { return Concrete(m_name); }

    std::string m_name;
};

void sample()
{
    ObjectMaker maker("adam");

    // You can't do this
    auto thing1 = maker<Thing1>();
    auto thing2 = maker<Thing2>();
}
```

You have to use more cumbersome syntax to specialize the overloaded operator:

```
void sample()
{
    ObjectMaker maker("adam");

    // You have to write it like this
    auto thing1 = maker.operator()<Thing1>();
    auto thing2 = maker.operator()<Thing2>();
}
```

It’s cumbersome, but at least it’s possible.

But if you’re going to do that, you may as well give it a name:

```
struct ObjectMaker
{
    ObjectMaker(std::string name) : m_name(std::move(name)) {}

    template<typename Concrete>
    Concrete make() { return Concrete(m_name); }

    std::string m_name;
};

void sample()
{
    ObjectMaker maker("adam");

    auto thing1 = maker.make<Thing1>();
    auto thing2 = maker.make<Thing2>();
}
```

[2] For further reading: What's the deal with `std::type_identity`?"