C++ template parlor tricks: Using a type before it is defined

devblogs.microsoft.com/oldnewthing/20221202-00

December 2, 2022

Raymond Chen

C++ templates are instantiated only on demand, which means that they provide you a way of talking about things before knowing what they are.

```
template<typename T>
auto get_foo(T&& t)
{
    return t.foo;
}
```

This template function takes any object and returns its `foo` member. It has no idea what type `T` it's going to be given, but by golly it's going to return the `foo` member.

Sometimes, though, you may know what type you have, but you don't have the power to say its name.

```
// contoso.h

namespace Contoso
{
    struct Point
    {
        int X;
        int Y;
    };
}

// litware.h

namespace LitWare
{
    struct Point
    {
        int X;
        int Y;

        constexpr Point() : X(0), Y(0) {}
        constexpr Point(int x, int y) : X(x), Y(y) {}

        // To ease interop with Contoso.
        constexpr Point(Contoso::Point const& cpt) : X(cpt.X), Y(cpt.Y) {}
    };
}
```

The idea here is that LitWare knows that a lot of its customers use the Contoso library, so it adds a conversion from `Contoso::Point` to for `LitWare::Point` to improve interop.

This works great, as long as the customer is actually using the Contoso library and has perform a `#include <contoso.h>`.

But the `litware.h` header may choose to be "Contoso-agnostic": It doesn't want to include `contoso.h` explicitly because the consuming application may not have the Contoso SDK installed, or maybe the consuming application declined to include the `contoso.h` header because it comes with other side effects like additional dependencies.

We can use a template here, taking advantage of the fact that dependent expressions in templates are not compiled until the template is instantiated.

```
        template<typename T>
        Point(T const& cpt) : X(cpt.X), Y(cpt.Y) {}
```

This constructor accepts any object which has fields called `X` and `Y` which can be used to initialize an `int`. We have a `Contoso::Point` in mind, but really, anything that meets the criteria will work.

Since this is defined as a template, the dependent expressions `cpt.X` and `cpt.Y` are not resolved until somebody actually tries to use that template.

```
void test(Contoso::Point& cpt)
{
    LitWare::Point lpt(cpt);
}
```

This invokes the constructor with a `Contoso::Point`, so `T` becomes `Contoso::Point`, and the `X` and `Y` members of the `LitWare::Point` are initialized with the corresponding members of the `Contoso::Point`.

Unfortunately, we also get confusing error messages when the thing that gets passed in doesn't satisfy these criteria:

```
void test(std::mutex& oops)
{
    // clang error: no member named 'X' in 'std::mutex'
    // gcc error: 'const class std::mutex' has no member named 'X'
    // msvc error: 'X' is not a member of 'std::mutex'
    LitWare::Point lpt(oops);
}
```

We were hoping for an error message like "Cannot convert `std::mutex` to `Contoso::Point`."

Another thing that doesn't work is constructing a `LitWare::Point` from an object that is *convertible to* a `Contoso::Point`:

```
struct ProtoPoint
{
    operator Contoso::Point const&() const;
};

void test(ProtoPoint& ppt)
{
    // clang error: no member named 'X' in 'ProtoPoint'
    // gcc error: 'const class ProtoPoint' has no member named 'X'
    // msvc error: 'X' is not a member of 'ProtoPoint'
    LitWare::Point lpt(ppt);

    // must convert explicitly
    LitWare::Point lpt(static_cast<Contoso::Point const&>(ppt));
}
```

We can sort of work around this with SFINAE and a forward declaration.

```
namespace Contoso
{
    struct Point;
}

namespace LitWare
{
    struct Point
    {
        ...

        template<typename T, typename = std::enable_if_t<
            std::is_convertible_v<std::decay_t<T>, Contoso::Point>>>
            constexpr Point(T const& cpt) ...
    };
}
```

but what do we write as the body of the constructor? We want to convert `T` to `Contoso::Point`, but do it only once. This means writing a helper.

```
    template<typename T, typename = std::enable_if_t<
        std::is_convertible_v<std::decay_t<T>, Contoso::Point>>>
        constexpr Point(T const& cpt)
        noexcept(noexcept(static_cast<Contoso::Point const&>(cpt))) :
            Point(convert(static_cast<Contoso::Point const&>(cpt))) {}

private:
    static constexpr Point convert(Contoso::Point const& cpt) noexcept
    {
        return { cpt.X, cpt.Y };
    }
```

Note the use of the noexcept noexcept idiom in the constructor to deal with the possibility that the conversion from `T` to `Contoso::Point` may throw.

But now we're back where we started: We need a definition for `Contoso::Point` in order to write `convert()`.

To escape this trap, we use two tricks. The first is to reintroduce the dependent type trick:

```
    template<typename T>
    static constexpr Point convert(T const& cpt) noexcept
    {
        return { cpt.X, cpt.Y };
    }
```

Now, the `convert()` function takes anything at all, but in practice, the only thing it is passed is a `Contoso::Point`. It's a C++ version of the Most Interesting Man in the World meme.

Our revised constructor is awfully wordy and still has problems.

For one thing, we might be adding a potentially-throwing constructor, whereas the original version consisted exclusively of non-throwing constructors. This may have ripple effects such as making the `LitWare::Point` ineligible for certain optimizations, or making it unusable by certain algorithms or data structures.

Another problem is that it changes the relative priority of the overload with respect to other overloads, since its specificity is lower than a simple conversion directly from `Contoso::Point`. This could result in conflicts with other constructors of `LitWare::Point`. For example, if you also want to support a `Fabrikam::Point` (which uses lowercase letters for the X- and Y-coordinates), you'll get a template redeclaration error.

```
    template<typename T, typename = std::enable_if_t<
        std::is_convertible_v<std::decay_t<T>, Contoso::Point>>>
        constexpr Point(T const& cpt)
        noexcept(noexcept(static_cast<Contoso::Point const&>(cpt))) :
            Point(convert(static_cast<Contoso::Point const&>(cpt))) {}

    // error: 'template<class T, class> Test::Test(const T&)' cannot be overloaded
with
    //        'template<class T, class> Test::Test(const T&)'
    template<typename T, typename = std::enable_if_t<
        std::is_convertible_v<std::decay_t<T>, Fabrikam::Point>>>
        constexpr Point(T const& cpt)
        noexcept(noexcept(static_cast<Fabrikam::Point const&>(cpt))) :
            Point(convert_fabrikam(static_cast<Fabrikam::Point const&>(cpt))) {}

private:
    static constexpr Point convert(Contoso::Point const& cpt) noexcept
    {
        return { cpt.X, cpt.Y };
    }

    static constexpr Point convert_fabrikam(Fabrikam::Point const& cpt) noexcept
    {
        return { cpt.x, cpt.y };
    }
```

Also, we generate a separate templated constructor for each argument type that is convertible to `Contoso::Point`, so you do have some template expansion bloat.

The error is also a little cumbersome if you pass something that isn't convertible to `Contoso::Point`:

```
std::string oops;
LitWare::Point pt(oops);

// clang
no matching constructor for initialization of 'LitWare::Point'

candidate template ignored: requirement 'std::is_convertible_<std::string,
Contoso::Point>' was not satisfied

    constexpr Point(T const& cpt) : Point(convert(static_cast<Contoso::Point const&>
(cpt))) {}

// gcc

no matching function for call to 'LitWare::Point::Point(std::string&)'

    LitWare::Point pt(oops);

candidate 'template<class T, class> constexpr LitWare::Point::Point(const T&)'

    constexpr Point(T const& cpt) : Point(convert(static_cast<Contoso::Point const&>
(cpt))) {}

template argument deduction/substitution failed.

// msvc
error: 'LitWare::Point::Point': none of the 4 overloads could convert all the
argument types

could be 'LitWare::Point::Point(LitWare::Point &&)'
or       'LitWare::Point::Point(const LitWare::Point &)'
or       'LitWare::Point::Point(int, int) noexcept'
or       'LitWare::Point::Point(void) noexcept'

while trying to match the argument list '(std::string)'
```

The Microsoft Visual C++ compiler doesn't even list the SFINAE'd away constructor, the one that the caller was trying to invoke, but couldn't because it failed the `enable_if`.

And finally, another point against this pattern is simply that it's a hot mess.[1]

Fortunately, there's a simpler solution.

Just declare the constructor as taking a `Contoso::Point`, but add a dummy template parameter to force delayed instantiation.

```
namespace LitWare
{
    struct Point
    {
        int X;
        int Y;

        constexpr Point() : X(0), Y(0) {}
        constexpr Point(int x, int y) : X(x), Y(y) {}

        // To ease interop with Contoso.
        template<bool dummy = true>
        constexpr Point(Contoso::Point const& cpt) noexcept
            : Point(convert<dummy>(cpt)) {}

    private:
        template<bool dummy, typename T>
        static constexpr Point convert(T&& cpt) noexcept
        { return { cpt.X, cpt.Y }; }
    };
}
```

With this version, there is only one templated constructor that will be instantiated in practice, namely the one where `dummy` is `true`. The conversion to `Contoso::Point` happens at the call site rather than in the constructor, and the constructor remains non-throwing. The explicitly-typed parameter helps the overload sit at the right priority in the overload resolution process, and you can repeat this pattern for other types without running into template redefinition errors.

This trick works with functions, too.

```
template<typename T>
void Widget_ToggleHelper(T& widget)
{
    if (widget.IsOn()) {
        widget.TurnOff();
    } else {
        widget.TurnOn();
    }
}

template<bool = true>
void Widget_Toggle(Widget& widget)
{
    Widget_ToggleHelper(widget);
}
```

Since lambdas with `auto` parameters are templates, you can move the helper function inline and turn it into a lambda, which makes it impossible for a consumer to call the helper directly.

```
template<bool = true>
void Widget_Toggle(Widget& widget)
{
    [](auto& widget)
    {
        if (widget.IsOn()) {
            widget.TurnOff();
        } else {
            widget.TurnOn();
        }
    }(widget);
}
```

Bonus chatter: This trick of using a dummy parameter to delay expansion also comes in handy if you need to break a circular reference between two classes:

```
struct TweedleDee
{
    TweedleDee(TweedleDum const& dum)
    : value(dum.value) {}

    int value;
};

struct TweedleDum
{
    TweedleDum(TweedleDee const& dee)
    : value(dee.value) {}

    int value;
};
```

Each of the classes is constructible from the other, but the constructor bodies each require that the other party be a complete type, putting them in a Catch-22 situation. You can break the cycle by introducing a dummy template parameter and a forward declaration:

```
struct TweedleDum;

struct TweedleDee
{
    template<bool dummy = true>
    TweedleDee(TweedleDum const& dum)
    : value(get_value_of<dummy>(dum)) {}

    template<typename T, bool>
    int get_value_of(T&& dum) { return dum.value; }

    int value;
};

struct TweedleDum
{
    TweedleDum(TweedleDee const& dee)
    : value(dee.value) {}

    int value;
};
```

or with the lambda trick:

```
struct TweedleDee
{
    template<bool dummy = true>
    TweedleDee(TweedleDum const& dum)
    : value([](auto&& dum)
            { return dum.value; }(dum)) {}

    int value;
};
```

This trick is less useful because you can also solve the problem without needing any tricks: Forward-declare the constructor.

```
struct TweedleDum;

struct TweedleDee
{
    TweedleDee(TweedleDum const& dum);

    int value;
};

struct TweedleDum
{
    TweedleDum(TweedleDee const& dee)
    : value(dee.value) {}

    int value;
};

inline TweedleDee::TweedleDee(TweedleDum const& dum)
    : value(dum.value) {}
```

[1] Another problem is the untyped braced list.

```
LitWare::Point({ 1, 2 });
```

We'll look at this problem some more later. But this is not really much of an issue in the `LitWare::Point` case, because you could just use the two-parameter constructor.

```
LitWare::Point(1, 2);
```

Raymond Chen

Follow