Instead of a C++ template parlor trick, why not just add support based on whether the header file has already been included?

devblogs.microsoft.com/oldnewthing/20221205-00

December 5, 2022

Raymond Chen

Last time, I showed off a C++ template parlor trick of using a type before it is defined. The idea is that you want to add optional support for a type provided by another header file, but you don't want to require the other header file. We accomplished this by forcing delayed instantiation of the constructor that consumes that foreign type.

But did we have to do all that? Why not just `#ifdef` the support for the foreign type based on whether the foreign header file has already been included?

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

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

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

This works, but it has a number of downsides.

For one, maybe Contoso also wants to provide courtesy support for LitWare types. If both LitWare and Contoso use the "You must include the other header file first", then you have a Catch-22 situation. Each header wants you to include the other one first. But *somebody* has to go first, and whoever goes first will therefore disable support for the other header.

Another downside is that it creates header file inclusion order dependencies, which are often land mines: Everything works fine for now, but once you perturb them, things explode.

For example, maybe you run a code tidying tool that sorts all include files alphabetically. This is not an uncommon policy, not just to keep things looking neat and organized, but also to reduce the likelihood of merge conflicts. But doing so will also change the order of inclusion, which will be an unintentional breaking change if any headers have dependencies on the order of inclusion.

Even if you are careful never to change the order of header file inclusion, you may run into a problem like this:

```
// client.cpp
#include "pch.h"

// Make sure to include contoso.h before litware.h, so
// that litware.h will activate its Contoso-related features.
#include <contoso.h>
#include <litware.h>
```

You compile the code, and the Contoso features are not active in LitWare.

But you included `contoso.h` first. It's right there. Would your eyes lie to you?

The problem is that `pch.h` also included `litware.h`, and that first inclusion is the one that counts. Since there was no `contoso.h` active at the time of the first inclusion, `litware.h` does not activate its Contoso-related features. In order to get `litware.h` to recognize Contoso, you'll have to modify `pch.h` so it includes `contoso.h` before it includes `litware.h`. Now you're changing a frequently-used header file, and the risk of unintented problems increases. (Maybe there's another client that uses a private header file that *conflicts* with `contoso.h`.)

Even if you are careful to keep all mention of either `contoso.h` or `litware.h` out of your `pch.h` (so that each component can decide for itself whether it wants `contoso.h` or not), you still have problems if some `.cpp` files choose to include `contoso.h` before `litware.h` and others do not: You run afoul of the *one definition rule*.

There are many provisions of the one definition rule, but the one relevant here is that if a class is used in multiple translation units, all definitions must be word-for-word identical. If some definitions have a Contoso helper constructor and others don't, then that's a violation of the rule, and the program is ill-formed.

The delayed-instantiation trick means that a single definition of the `LitWare::Point` class can be used regardless of when a definition of `Contoso::Point` is made, or whether it even appears at all. That way, `litware.h` doesn't care whether it is included before or after

`contoso.h`. You just have to make sure you have included both headers before you try to use a Contoso-related feature from `litware.h`. (This is easy to arrange in practice, because without `contoso.h`, you can't even access any Contoso objects.)

Bonus chatter: The wil library takes a different approach to activating features conditionally: The `wil/resource.h` header file can be included multiple times, and each inclusion turns on features that were dependent upon any newly-included headers.

```
#include <first.h>
#include <second.h>

// This inclusion activates features that are dependent
// upon first.h or second.h.
#include <wil/resource.h>

#include <third.h>

// This inclusion activates features that are dependent
// upon third.h.
#include <wil/resource.h>
```

In this way, you avoid order-of-inclusion problems because you can always include `wil/resource.h` again, and it will turn on any features that are available at the point of inclusion.

Raymond Chen

Follow