Writing a remove_all_pointers type trait, part 2

devblogs.microsoft.com/oldnewthing/20240628-00

June 28, 2024

Raymond Chen

Last time, we wrote a `remove_all_pointers` type trait, but I noted that even though we found a solution, we weren't finished yet.

We can bring back the one-liner by using a different trick to delay the recursion: Don't ask for the `type` until we know we really are recursing.

The sketch is

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

template<typename T>
struct remove_all_pointers
{
    if (std::is_pointer_v<T>) {
        using type_holder =
            remove_all_pointers<
                std::remove_pointer_t<T>
            >;
    } else {
        using type_holder = dummy<T>;
    }
    using type =
        typename type_holder::type;
};
```

We first define a `type_holder` to be a type which has a `type` member type that holds our answer. If `T` is a pointer, then the type holder is the recursive call. Otherwise, the type holder is a dummy type whose sole purpose is to have a `type` member type that produces `T` again.

We can now pack up that `if` into a `std::conditional`.

```
template<typename T>
struct remove_all_pointers
{
    using type_holder =
        std::conditional_t<
            std::is_pointer_v<T>,
            remove_all_pointers<
                std::remove_pointer_t<T>
            >,
            dummy<T>>;
    using type =
        typename type_holder::type;
};
```

It turns out that we don't need to define a `dummy`: The C++ standard library comes with one built in! It's called `std::type_identity<T>`, available starting in C++20. (We looked at `std::type_identity<T>` a little while ago.)

```
template<typename T>
struct remove_all_pointers
{
    using type_holder =
        std::conditional_t<
            std::is_pointer_v<T>,
            remove_all_pointers<
                std::remove_pointer_t<T>
            >,
            std::type_identity<T>>;
    using type =
        typename type_holder::type;
};
```

Now we can inline the `type_holder`.

```
template<typename T>
struct remove_all_pointers
{
    using type =
        typename std::conditional_t<
            std::is_pointer_v<T>,
            remove_all_pointers<
                std::remove_pointer_t<T>
            >,
            std::type_identity<T>>::type;
};
```

Or even better, just derive from the `type_holder`!

```
template<typename T>
struct remove_all_pointers :
        std::conditional_t<
            std::is_pointer_v<T>,
            remove_all_pointers<
                std::remove_pointer_t<T>
            >,
            std::type_identity<T>>
{
};
```