Writing a remove_all_pointers type trait, part 1

devblogs.microsoft.com/oldnewthing/20240627-00

June 27, 2024

Raymond Chen

There is a `std::remove_pointer` type trait helper. If you give it a pointer type `T*`, its `type` member type is `T`. Otherwise, the `type` member type is just the template type unchanged.

But what if you want to remove *all* pointers? For example, `remove_all_pointers<int*const*volatile*>::type` should be `int`.

You can define this as a recursive operation. In pseudo-code:

```
template<typename T>
auto remove_all_pointers
{
    if (std::is_pointer_v<T>) {
        return remove_all_pointers<
            std::remove_pointer_t<T>
        >;
    } else {
        return T;
    }
}
```

One way to express conditional evaluation in template metaprogramming is to use `std::conditional<a, b, c>::type`, which is `b` if `a` is `true` and is `c` if `a` is `false`.

Therefore, your first attempt might be to write it as a one-liner built out of `std::conditional`.

```
template<typename T>
using remove_all_pointers_t =
    std::conditional_t<
        std::is_pointer_v<T>,
        remove_all_pointers_t<
            std::remove_pointer_t<T>&gt,;
        T>;
```

Okay, this doesn't work because of the recursive reference to `remove_all_pointers_t` before it has completed its declaration. We can sidestep this by using a `struct`.

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

This compiles, but you get an error when you try to use it:

```
using test = remove_all_pointers<int*const*volatile*>::type;

// gcc
In instantiation of 'struct remove_all_pointers<int>':
    recursively required from 'struct remove_all_pointers<int* const* volatile>'
    required from 'struct remove_all_pointers<int* const* volatile*>'
    required from here
error: invalid use of incomplete type 'struct remove_all_pointers<int>'
    |   using type = std::conditional_t<
    |         ^~~~
note: definition of 'struct remove_all_pointers<int>' is not complete until the
closing brace
    | struct remove_all_pointers
    |        ^~~~~~~~~~~~~~~~~~

// clang
error: no type named 'type' in 'remove_all_pointers<int>'
    |         typename remove_all_pointers<
    |         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    |             std::remove_pointer_t<T>>::type,
    |             ~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~
note: in instantiation of template class 'remove_all_pointers<int>' requested here
    |         typename remove_all_pointers<
    |                  ^
note: in instantiation of template class 'remove_all_pointers<int *const>' requested
here
note: in instantiation of template class 'remove_all_pointers<int *const *volatile>'
requested here
note: in instantiation of template class 'remove_all_pointers<int *const *volatile
*>' requested here
   | using test = remove_all_pointers<int*const*volatile*>::type;
   |              ^

// msvc
error C2146: syntax error: missing '>' before identifier 'type'
note: the template instantiation context (the oldest one first) is
note: see reference to class template instantiation 'remove_all_pointers<int *const
*volatile *>' being compiled
note: see reference to class template instantiation 'remove_all_pointers<int *const
*volatile >' being compiled
note: see reference to class template instantiation 'remove_all_pointers<int *const
>' being compiled
note: see reference to class template instantiation 'remove_all_pointers<int>' being
compiled
```

Okay, maybe we were too ambitious.

All the error messages show that the template was able to recurse and strip away pointers, but then it ran into a problem when it reached the base case. Let's look at that base case:

```
struct remove_all_pointers<int>
{
  using type = std::conditional_t<
        std::is_pointer_v<int>,
        remove_all_pointers<
            std::remove_pointer_t<int>>::type,
        int>;
};
```

After substituting `std::remove_pointer_t<int> = int`, we get

```
struct remove_all_pointers<int>
{
  using type = std::conditional_t<
        std::is_pointer_v<int>,
        remove_all_pointers<int>::type,
        int>;
};
```

Now we see the problem. The definition of `remove_all_pointers<int>::type` is dependent on itself.

The catch here is that `std::conditional` is not a short-circuiting operator. How can it be? It's a template!

In order to instantiate a template, the compiler first evaluates the template parameters, and then it looks at the template expansion that results. The compiler doesn't "look ahead" and say, "Oh, I can tell that the template expansion never uses its second parameter, so I will skip the evaluation of the second parameter."[1]

One way to solve this problem is to move the expansion of the two parameters to a partial specialization. That way, only the pointer cases invoke the template recursively.

```
template<typename T,
    bool = std::is_pointer_v<T>>
struct remove_all_pointers;

template<typename T>
struct remove_all_pointers<T, false>
{
    using type = T;
};

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

We add a hidden second template parameter which defaults to `std::is_pointer_v<T>`. We then partially specialize the template on that second template parameter: If it's `false` (`T` is not a pointer), then the type is `T` itself, which provides our base case (no longer accidentally referring to itself). If it's `true` (`T` is a pointer), then the type is calculated recursively after stripping away one layer of indirection.

```
template<typename T>
using remove_all_pointers_t =
    typename remove_all_pointers<T>::type;

static_assert(std::is_same_v<
    remove_all_pointers_t<int*const*volatile*>,
    int>);
```

As a small tuning step, we can fold the base case into the initial definition, so that only the recursive case is a partial specialization.

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

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

Are we done?

No, not yet.

We'll continue next time.

[1] Indeed, the "I evaluate all the parameters even if they aren't used" behavior is one of the things that SFINAE relies on!