What’s the deal with std::type_identity?

devblogs.microsoft.com/oldnewthing/20240607-00

June 7, 2024

Raymond Chen

C++20 has this new thing called `std::type_identity`. Its definition is basically

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

In other words, `type_identity<T>::type = T`.

This sounds profoundly useless. Why take the type `T`, wrap it in another type, and then unwrap it? Why not just use `T` directly?

The primary purpose of `type_identity<T>` is to allow you to use a type without making it participate in type deduction.

The classic example is a function like, say, `add`:

```
template<typename T>
T add(T a, T b)
{
    return a + b;
}
```

This says “Add two things of the same type and return the result.”

So what happens if you pass two things of different type?

```
auto sum = add(0.5, 1); // error: cannot deduce T
```

Maybe you want the policy to be “The type of the first parameter determines what `T` is, and everybody else has to play along.” You can use `type_identity` to specify that the second parameter is not deducible.

```
template<typename T>
T add(T a, std::type_identity_t<T> b)
{
    return a + b;
}
```

This time, it works:

```
auto sum = add(0.5, 1); // T is "double"
```

The compiler deduces `T = double` when it matches `0.5` to the `a` parameter. When it gets around to the second parameter, it is told, “Just use the `T` you deduced somewhere else.” So the parameter `1` is treated as a `double`, which involves a numeric conversion.

A more complex case for wanting one parameter’s type to be dependent upon another parameter’s type is this example:

```
void enqueue(std::function<void(void)> const& work);

template<typename...Args>
void enqueue(std::function<void(Args...)> const& work,
    Args...args)
{
    enqueue([=] { work(args...); });
}
```

The idea here is that you call `enqueue` with a callable and some optional arguments, and the function enqueues the function call to be run somewhere else (maybe on a worker thread). If you pass arguments, then the arguments are passed to your callable.

So you decide to give this function a whirl, but it doesn’t compile:

```
enqueue([](int v) { std::cout << v; }, 42);

// gcc
error: no matching function for call to 'enqueue(<lambda(int)>, int)'
   |     enqueue([](int){}, 42);
   |     ~~~~~~~^~~~~~~~~~~~~~~
note: candidate: 'template<class ... Args> void enqueue(const std::function<void(Args
...)>&, Args ...)'
    | void enqueue(
    |      ^~~~~~~
note:   template argument deduction/substitution failed:
note:   '<lambda(int)>' is not derived from 'const std::function<void(Args ...)>'
   |     enqueue([](int){}, 42);
   |     ~~~~~~~^~~~~~~~~~~~~~~

// clang
error: no matching function for call to 'enqueue'
   |     enqueue([](int){}, 42);
   |     ^~~~~~~
note: candidate template ignored: could not match 'std::function<void (Args...)>'
against '(lambda)'
    7 | void enqueue(
      |      ^

// msvc
error C2672: 'enqueue': no matching overloaded function found
note: 'void enqueue(const std::function<void(Args...)> &,Args...)': could not deduce
template argument for 'const std::function<void(Args...)> &' from '<lambda>'
```

The problem is that template type deduction tries to match types, and it doesn't consider conversions. Conversions are handled by a later step (overload resolution). The deduction fails because the lambda is not a `std::function`; the fact that the lambda can be *converted to* a `std::function` is irrelevant.

We can fix this by using `type_identity` to remove the `std::function` from participating in template type deduction.

```
template<typename...Args>
void enqueue(
    std::type_identity_t<
        std::function<void(Args...)>
    > const& work,
    Args...args)
{
    enqueue([=] { work(args...); });
}
```

Now the deduction of `Args` is controlled only by the second and subsequent parameters. Once that's settled, the compiler then passes the selected function to overload resolution, which says, "Oh, I can convert this lambda to match the type of the first parameter."

Bonus chatter: You don't have to use `type_identity` to wrap the entire type. You can scope it down to something smaller, as long as the type you want to exempt from deduction is inside the template type parameter:

```
template<typename...Args>
void enqueue(
    std::function<void(
        std::type_identity_t<Args>...
    )> const& work,
    Args...args)
{
    enqueue([=] { work(args...); });
}
```

Bonus bonus chatter: In practice, you would probably receive the variadic arguments by universal reference, which means that the `std::function` will probably receive decayed parameters rather than references. The insertion of `std::decay_t` not only decays the `Args`, but it also prevents them from participating in deduction.

```
template<typename...Args>
void enqueue(
    std::function<void(
        std::decay_t<Args>...
    )> const& work,
    Args&&...args)
{
    enqueue([work, ...args = std::forward<Args>(args)]
            { work(std::move(args)...); });
}
```

Bonus bonus bonus chatter: The proposal for `type_identity` lists some other uses: To force programmers to specify the template type explicitly rather than allowing it to be deduced.

```
template<typename T>
void must_specialize(std::type_identity_t<T> t);

must_specialize(42); // not allowed
must_specialize<int>(42); // must state "int" explicitly
```

It can also be used to suppress class template argument deduction (CTAD), and it can be handy as a building block for template metaprogramming. Read the proposal for details.