The type-dependent type or value that is independent of the type

devblogs.microsoft.com/oldnewthing/20200413-00

April 13, 2020

Raymond Chen

Some time ago, we saw how to create a type-dependent expression that is always false. I noted that it feels weird creating a whole new type just to create a fixed `false` value.

But maybe we can make it more useful by generalizing it.

```
template<typename T, typename...>
using unconditional_t = T;

template<typename T, T v, typename...>
inline constexpr T unconditional_v = v;
```

The `unconditional_t` alias template always represents the type `T`, and the `unconditional_v` variable template always represents the value `v`.

```
template<typename Whatever>
void f()
{
  // X is always int
  using X = unconditional_t<int, Whatever>;

  // v is always 42
  auto v = unconditional_v<int, 42, Whatever>;
}
```

Even though the resulting type or value is always the same, it is nevertheless a dependent type, and therefore the evaluation does not occur until template instantiation.

We can use this to solve our "cannot `static_assert(false)` in a discarded statement" problem:

```
auto lambda = [total](auto op, auto value) mutable
{
  using Op = decltype(op);
  if constexpr (std::is_same_v<Op, add_tax_t>) {
   total += total * value; // value is the tax rate
   return total;
  } else if constexpr (std::is_same_v<Op, apply_discount_t>) {
   total -= std::max(value, total); // value is the discount
   return total;
  } else {
   static_assert(unconditional_v<Op, bool, false>,
                 "Don't know what you are asking me to do.");
  }
};
```

The `unconditional_t` generalizes the alias template `std::void_t<...>`: Whereas `std::void_t<...>` always evaluates to `void`, the `unconditional_t` lets you pick the type that it resolves to.

```
template<typename... Types>
using void_t = unconditional_t<void, Types...>;
```

The `unconditional_t` also generalizes the template class `std::type_identity<T>`: Whereas `std::type_identity<T>` takes only one template type parameter, the `unconditional_t` lets you pass extra parameters, which are evaluated (for SFINAE) but otherwise ignored.

```
template<typename T>
using type_identity = unconditional_t<T>;
```

Raymond Chen

Follow