How can I create a type-dependent expression that is always false?

devblogs.microsoft.com/oldnewthing/20200311-00

March 11, 2020

Raymond Chen

Giving a C++ lambda expression more than one `operator()` was an abuse of the language.[1] But one of the side effects of exploring ways to abuse the language is that during your explorations, you may discover a useful trick.

One of the things I had to do was prevent compilation from succeeding if the lambda was called incorrectly. I had a chain of `if constexpr` tests for the valid cases, and I needed to put a `static_assert` in the `else` that said "You should never get here."

```
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(false, "Don't know what you are asking me to do.");
  }
};
```

However, this does not compile because the `static_assert` fails immediately.

The reason is that the controlling expression for the `static_assert` is not dependent upon the type of the arguments, and therefore it is evaluated when the lambda is compiled, not when the lambda is invoked (and the implicit template instantiated).[2]

In order to defer the `static_assert` to instantiation, we need to make it type-dependent.

What is a type-dependent expression that is always false?

We could always make up our own:

```
template<typename T>
inline constexpr bool always_false_v = false;

...

 static_assert(always_false_v<Op>,
               "Don't know what you are asking me to do.");
```

but it feels weird creating a whole new variable template just to generate a fixed `false` value.[3] Maybe we can live off the land.

We could take advantage of the fact that `sizeof` is never zero.[4]

```
 static_assert(!sizeof(Op),
               "Don't know what you are asking me to do.");
```

but this runs into problems if `Op` is an incomplete type or `void`. Now, the way we happen to have written our code, an incomplete type and `void` are not possible because the type corresponds to an actual parameter. But let's look for a more general solution.

If the type is indeed incomplete or `void`, then the code will fail to compile, but the error message will be confusing because the provided error text will not be used: The error occurred before the compiler could get that far.

However, *pointers to* incomplete types or `void` are valid. So we could do this:

```
 static_assert(!sizeof(Op*),
               "Don't know what you are asking me to do.");
```

A static assertion of a type-dependent expression that is always false is a handy thing to put into templates, because it defers the assertion failure to the instantiation of the template. Here, we used it in a potentially-discarded statement, so that the instantiation does not occur when the statement is discarded.

We'll find another use next time.

Bonus chatter: Billy O'Neal called out some gotchas with this approach, which I'll take up in a future entry.

[1] What some people call an abuse of the language others call a *proxy object*, such as the one produced by `std::vector<bool>` 's `[]` operator.

[2] This does raise a confusing point in the C++ standard. According to the standard, the not-used branch of an `if constexpr` is a *discarded statement*. This is the only place where the term *discarded statement* appears in the standard. And it is never defined! The closest thing to a definition is the sentence

> During the instantiation of an enclosing templated entity (Clause 17), if the condition is not value-dependent after its instantiation, the discarded substatement (if any) is not instantiated.

which describes a *discarded substatement*. And it doesn't really define what a discarded substatement is. It just names one characteristic of discarded substatements.

I think the standard intended the sentence to be something like

> A *discarded statement* is treated the same as a statement, except that during the instantiation of an enclosing templated entity (Clause 17), if the condition is not value-dependent after its instantiation, the discarded statement (if any) is not instantiated.

[3] See the proposal for `std::dependent_false` (and committee sentiment) for further discussion.

[4] Empty base optimization and `[[no_unique_address]]` also scare me, because they can lead to an object having an effective size of zero. I don't want to get caught out if a future version of the standard makes some subtle changes that lead to `sizeof(T) == 0` in some fringe cases.

Raymond Chen

Follow