together with a consistent name for the enumeration, such as `Enum`, has to be used to indicate an enumerated type:

```
void f3(VehicleScopedExplicitly::Enum value);
    // classically scoped enum passed by value
```

Qualifying the enumerators of a scoped enumeration is the same, irrespective of whether the scoping is explicit or implicit:

```
void g()
{
    f3(VehicleScopedExplicitly::e_PLANE);
        // call f3 with an explicitly scoped enumerator

    f2(VehicleScopedImplicitly::e_PLANE);
        // call f2 with an implicitly scoped enumerator
}
```

Apart from implicit scoping, the modern, C++11 scoped enumeration deliberately does *not* support implicit conversion, in any context, to its underlying type:

```
void h()
{
    int i1 = VehicleScopedExplicitly::e_PLANE;
       // OK, scoped C++03 enum (implicit conversion)

    int i2 = VehicleScopedImplicitly::e_PLANE;
        // Error, no implicit conversion to underlying type

    if (VehicleScopedExplicitly::e_PLANE > 3) {} // OK
    if (VehicleScopedImplicitly::e_PLANE > 3) {} // Error, implicit conversion
}
```

Enumerators of an **enum class** do, however, admit equality and ordinal comparisons within their own type:

```
enum class E { e_A, e_B, e_C };  // By default, enumerators increase from 0.

static_assert(E::e_A < E::e_C, "");  // OK, comparison between same-type values
static_assert(0 == E::e_A, "");      // Error, no implicit conversion from E
static_assert(0 == static_cast<int>(E::e_A), "");  // OK, explicit conversion

void f(E v)
{
    if (v > E::e_A) { /*...*/ }  // OK, comparing values of same type, E
}
```

~~Note that incrementing an enumeration variable from one strongly typed enumerator's value to the next requires an explicit cast; see *Potential Pitfalls — Strong typing of an* **enum class** *can be counterproductive* on page 344.~~