those values. ~~Comparing values having distinct enumerated types, however, is deprecated and will typically elicit a warning.~~[1]

Introducing the C++11 scoped enumerations

With the advent of modern C++, we now have a new, alternative enumeration construct, **enum class**, that simultaneously addresses strong type safety and lexical scoping, two distinct and often desirable properties:

```
enum class Name { e_Enumerator0 /* = value0 */, e_EnumeratorN /* = valueN */ };
    // enum class enumerators are both type-safe and scoped
```

Another major distinction is that the default underlying type for a C-style **enum** is **implementation defined**, whereas, for an **enum class**, it is always an **int**. See ***enum class*** *and underlying type* on page 337 and *Potential Pitfalls — External use of opaque ~~enumerators~~* on page 350.

Unlike unscoped enumerations, **enum class** does not leak its enumerators into the enclosing scope and can therefore help avoid collisions with other enumerations having like-named enumerators ~~defined~~ in the same scope:

```
enum       VehicleUnscoped  { e_CAR, e_TRAIN, e_PLANE };
struct     VehicleScopedExplicitly { enum Enum { e_CAR, e_TRAIN, e_PLANE }; };
enum class VehicleScopedImplicitly { e_CAR, e_BOAT, e_PLANE };
```

Just like an unscoped **enum** type, an object of a scoped enumeration type is passed as a parameter to a function using the enumeration name itself:

```
void f1(VehicleUnscoped value);          // unscoped enumeration passed by value
void f2(VehicleScopedImplicitly value);  // scoped enumeration passed by value
```

If we use the approach for adding scope to enumerators that is described in *Drawbacks relating to weakly typed, C++03 enumerators* on page 333, the name of the enclosing **struct**

[1] As of C++20, attempting to compare two values of distinct classically enumerated types is a compile-time error. Note that explicitly converting at least one of them to an integral type — for example, using built-in unary plus — both makes our intentions clear and avoids warnings.

```
void test()
{
  if (e_A0 < 0)       { /*...*/ }  // OK, comparison with integral type
  if (1.0 != e_B1)    { /*...*/ }  // OK, comparison with arithmetic type
  if (A() <= e_A2)    { /*...*/ }  // OK, comparison with same enumerated type
  if (e_A0 == e_B0)   { /*...*/ }  // warning, deprecated (error as of C++20)
  if ( e_A0 == +e_B0) { /*...*/ }  // OK, unary + converts to integral type
  if (+e_A0 ==  e_B0) { /*...*/ }  // OK,   "         "     "     "      "
  if (+e_A0 == +e_B0) { /*...*/ }  // OK,   "         "     "     "      "
}
```