enum class and underlying type

Since C++11, both scoped and unscoped enumerations permit explicit specification of their integral underlying type (see Section 2.1.“Underlying Type ’11” on page 829):

```
enum Ec : char { e_X, e_Y, e_Z };
    // Underlying type is char.

static_assert(1 == sizeof(Ec),     "");
static_assert(1 == sizeof Ec::e_X, "");

enum class Es : short { e_X, e_Y, e_Z };
    // Underlying type is short int.

static_assert(sizeof(short) == sizeof(Es),     "");
static_assert(sizeof(short) == sizeof Es::e_X, "");
```

Unlike a classic **enum**, which has an implementation-defined default underlying type, the default underlying type for an **enum class** is always **int**:

```
enum class Ei { e_X, e_Y, e_Z };
    // When not specified, the underlying type of an enum class is int.

static_assert(sizeof(int) == sizeof(Ei),     "");
static_assert(sizeof(int) == sizeof Ei::e_X, "");
```

Note that, because the default underlying type of an **enum class** is specified by the Standard, eliding the enumerators of an **enum class** in a local redeclaration is *always* possible; see *Potential Pitfalls — External use of opaque ~~enumerators~~* on page 350 and Section 2.1. “Opaque **enum**s” on page 660.

Use Cases

Avoiding unintended implicit conversions to arithmetic types

Suppose that we want to represent the result of selecting one of a fixed number of alternatives from a drop-down menu as a simple unordered set of uniquely valued named integers. For example, this might be the case when configuring a product, such as a vehicle, for purchase:

```
struct Transmission
{
    enum Enum { e_MANUAL, e_AUTOMATIC };  // classic, C++03 scoped enum
};
```

Although automatic promotion of a classic enumerator to **int** works well when typical use of the enumerator involves knowing its cardinal value, such promotions are less than ideal when cardinal values have no role in intended usage: