```
class Car { /*...*/ };

struct Transmission
{                                       // explicitly scoped
    enum Enum { e_MANUAL, e_AUTOMATIC };  // classic enum
};                                      // (BAD IDEA)

int buildCar(Car* result, int numDoors, Transmission::Enum transmission)
{
    int status = Transmission::e_MANUAL;    // Bug, accidental misuse

    for (int i = 0; i < transmission; ++i)  // Bug, accidental misuse
    {
         attachDoor(i);
    }

    return status;
}
```

As shown in the example above, it is never correct for a value of type `Transmission::Enum` to be assigned to, compared with, or otherwise modified like an integer; hence, *any* such use would necessarily be considered a mistake and, ideally, flagged by the compiler as an error. The stronger typing provided by **`enum class`** achieves this goal:

```
class Car { /*...*/ };

enum class Transmission { e_MANUAL, e_AUTOMATIC };  // modern enum class (GOOD IDEA)

int buildCar(Car* result, int numDoors, Transmission transmission)
{
    int status = Transmission::e_MANUAL;    // Error, incompatible types

    for (int i = 0; i < transmission; ++i)  // Error, incompatible types
    {
         attachDoor(i);
    }

    return status;
}
```

By deliberately choosing the **`enum class`** in the example above over the *classic* **`enum`**, we automate the detection of many common kinds of accidental misuse. Secondarily, we slightly simplify the interface of the function **`signature`** by removing the extra `::Enum` boilerplate qualifications required of an explicitly scoped, less-type-safe, classic **`enum`**, but see *Potential Pitfalls — Strong typing of an* ***`enum class`*** *can be counterproductive* on page 344.

In the event that the numeric value of a strongly typed enumerator is needed (e.g., for serialization), it can be extracted explicitly via a **`static_cast`**: