```
class ByteStream
{
public:
    void send(unsigned char byte) { /*...*/ }
    void send(int) = delete;

    // ...
};

void f()
{
    ByteStream stream;
    stream.send(0);   // Error, send(int) is deleted.     (1)
    stream.send('a'); // Error, send(int) is deleted.     (2)
    stream.send(0L);  // Error, ambiguous                 (3)
    stream.send(0U);  // Error, ambiguous                 (4)
    stream.send(0.0); // Error, ambiguous                 (5)
    stream.send(
        static_cast<unsigned char>(100));  // OK          (6)
}
```

Invoking `send` with an **`int`** — noted with (1) in the code above — or any integral type, other than **`unsigned char`**, that promotes to **`int`** (2) will map exclusively to the deleted **`send(int)`** overload; all other integral, (3) and (4), and floating-point types (5) are convertible to both via a **standard conversion** and hence will be ambiguous. Note that implicitly converting from **`unsigned char`** to either a **`long`** or **`unsigned`** integer involves a **standard conversion** (not just an **integral promotion**), the same as converting to a **`double`**. An explicit cast to **`unsigned char`** (6) can always be pressed into service if needed.

Hiding a structural, nonpolymorphic base class's member function

Avoiding deriving publicly from concrete classes is commonly advised because by doing so, we do not hide the underlying capabilities, which can easily be accessed (potentially breaking any invariants the derived class might want to keep) via assignment to a pointer or reference to a base class, with no casting required. Worse, inadvertently passing such a class to a function taking the base class by value will result in slicing, which can be especially problematic when the derived class holds data. A more robust approach would be to use layering or at least private inheritance.[1] Best practices notwithstanding,[2] it can be cost-effective in the short term to provide an elided "view" on a concrete class for trusted clients. Imagine a class `AudioStream` designed to play sounds and music that — in addition to providing basic "play" and "rewind" operations — sports a large, robust interface:

[1] For more on improving compositional designs at scale, see **lakos20**, section 3.5.10.5, "Realizing Multicomponent Wrappers," and section 3.7.3, "Improving Purely Compositional Designs," pp. 687–703 and 726–727, respectively.

[2] See **meyers92**, "Item 38: Never define an inherited default parameter value," pp. 132–135.