Using the `= delete` syntax on declarations that are private results in error messages concerning privacy, not the use of deleted functions. Care must be exercised to make *both* changes when converting code from the old style to the new syntax.

Preventing a particular implicit conversion

Certain functions — especially those that take a `char` as an argument — are prone to inadvertent misuse. As a truly classic example, consider the C library function `memset`, which might be used to write the character `*` five times in a row, starting at a specified memory address, `buf`:

```
#include <cstdio>   // puts
#include <cstring>  // memset

void f()
{
    char buf[] = "Hello World!";
    memset(buf, 5, '*');  // undefined behavior: buffer overflow
    puts(buf);            // expected output: "***** World!"
}
```

Sadly, inadvertently reversing the order of the last two arguments is a commonly recurring error, and the C language provides no help. As shown above, `memset` writes the nonprinting character `'\x5'` 42 (i.e., the integer value of ASCII `'*'`) times, way past the end of `buf`. In C++, we can target such observed misuse using an extra deleted overload:

```
namespace my {
void* memset(void* str, int ch, std::size_t n);  // Standard Library equivalent
void* memset(void* str, int n, char) = delete;   // defense against misuse
}
```

Pernicious user errors can now be reported during compilation:

```
void f2()
{
    char buf[] = "Hello World!";
    my::memset(buf, 5, '*');              // Error, call to deleted function
    my::memset(buf, '*', (std::size_t)5);  // OK
}
```

Preventing all implicit conversions

The `ByteStream::send` member function on the next page is designed to work with 8-bit unsigned integers only. Providing a deleted overload accepting an `int` forces a caller to ensure that the argument is always of the appropriate type: