value, whose value is **false**. ~~Although this second implementation is more likely to produce the desired result (i.e., a controlled compilation failure only when serialize is invoked with unsuitable arguments), sufficiently sophisticated compilers looking at just the current translation unit would still be able to know that no valid instantiation of serialize exists and would therefore be well within their rights to refuse to compile this still technically ill formed program.~~

~~Equivalent workarounds achieving the same result without a helper class are possible.~~

```
template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<false>)  // (2c)
{
    static_assert(0 == sizeof(T), "T must be serializable.");  // OK
         // not too obviously ill formed: compile-time error when instantiated
}
```

Using this sort of obfuscation is not guaranteed to be either portable or future-proof.

Misuse of static assertions to restrict overload sets

Even if we are careful to *fool* the compiler into thinking that a specialization is wrong *only* if instantiated, we still cannot use this approach to remove a candidate from an overload set because translation will terminate if the static assertion is triggered. Consider this flawed attempt at writing a process function that will behave differently depending on the size of the given argument:

```
template <typename T>
void process(const T& x)  // (1) first definition of process function
{
    static_assert(sizeof(T) <= 32, "Overload for small types");  // BAD IDEA
    // ... (process small types)
}

template <typename T>
void process(const T& x)  // (2) compile-time error: redefinition of function
{
    static_assert(sizeof(T) > 32, "Overload for big types");     // BAD IDEA
    // ... (process big types)
}
```

While the intention of the developer might have been to statically dispatch to one of the two mutually exclusive overloads, the ill-fated implementation above will not compile because the signatures of the two overloads are identical, leading to a redefinition error. The semantics of **static_assert** are not suitable for the purposes of **compile-time dispatch**, and SFINAE-based approaches might be used instead.

To achieve the goal of removing ~~up front~~ a specialization from consideration, we ~~will need to~~ employ SFINAE. To do that, we must instead find a way to get the failing compile-time expression to be part of the function's **declaration**: