```
template <typename T>
typename std::enable_if<std::is_copy_assignable<T>::value>::type
optimizedCopy2(T* dest, const T* first, const T* last)
    // Copy elements from range [first, last) to the range starting at dest.
    // This function does not participate in overload resolution unless
    // std::is_copy_assignable<T>::value is true.
{
    // ...  (body including static_assert unchanged from optimizedCopy above)
}
```

In the example above, the return type of `optimizedCopy2` will be **`void`** if `std::is_copy_assignable<T>::value` is **`true`** and ill formed otherwise. Unlike our previous use of a **`static_assert`** in the body of the original `optimizedCopy`, an ill-formed specialization of a function template does not necessarily result in a compile-time error but, instead, eliminates that specialization from the overload set, thereby allowing another, viable overload, if any, to be selected instead. In particular, note that this way, the **`static_assert`** in the body of an ill-formed specialization never fires and is therefore an entirely redundant defensive check. An additional advantage of using `std::enable_if` in this example is that the copy assignable constraint is expressed, in addition to any English documentation, directly in the programmatic interface. On the other hand, for a function template such as `optimizedCopy2`, having just a **`static_assert`** might produce a more comprehensible error message than, e.g., "Error - no matching function for `optimizedCopy2`."[31]

There are vanishingly few cases in practice where `std::is_trivially_copy_constructible` or `std::is_trivially_copy_assignable` would be appropriate in the implementation of a function template as there's nothing special in the core language that comports with them. (Satisfying the requirements of one or both of these interface traits is neither necessary nor sufficient for the argument type to be trivially copyable.) What's more, a type that is trivially copyable might not satisfy either of these interface traits, e.g., due to the trait's additional requirements, such as public accessibility, invocability, etc. Conversely, `std::is_trivially_copyable` identifies precisely the superset of trivial types that, for example, may be safely copied via `std::memcpy` because this core trait takes into account all *five* (including the destructor) of the relevant special member functions associated with being trivially copyable, and yet it would be rare to see `std::is_trivially_copyable` used properly in the *interface* of a well-specified function template unless that template provides a low-level service. Hence, core traits properly belong to the implementation of generic functions, whereas interface traits almost always reside in the interface (but see *Potential Pitfalls — Ineligible use of `std::memcpy`* on page 497).

When it comes to class templates, there are other kinds of appropriate uses of `std::is_trivially_copy_assignable` and its ilk. Suppose, for example, that we want to guarantee that a class template that wraps some other type (e.g., `Wrap<T>`) is *non*assignable, copy assignable, or **trivially copy assignable** corresponding to the existence and triviality of the copy-assignment operation defined for its template type argument, `T`. The

[31]In C++20, a **`requires`** clause (part of the concepts feature) provides an easier-to read alternative to `std::enable_if` and also produces a more comprehensible error message when the requirement is not met.