Class `V` above is trivially copyable in every version of C++, but the `std::is_trivially_copyable` trait might not reflect that on some older versions of C++14 compilers, depending on which interpretation of that Standard is in effect; see *Relevant standard type traits are unreliable* on page 527.

One might question what happens with trivially copyable types that have **`const`**-qualified or reference data members and whether bitwise copying (e.g., using `std::memcpy`) such objects has undefined behavior as these bitwise-copy operations inevitably overwrite a reference or **`const`** object. Performing a bitwise copy of such objects and then subsequently using that data in any way was modified in C++20, when a twofold change was made: (1) If a *non***`const`** object — referred to by a reference, pointer, or name (call it a *ref*) — is destroyed and a new object of the same type is subsequently constructed at the same location, the usability of the original *ref*, as of C++20, is not impacted by whether the object contains a **`const`** or reference subobject, whereas the existence of such a subobject, prior to C++20, would have rendered the *ref* unusable; and (2) `std::memcpy` and `std::memmove` implicitly create a new object in the destination location, making the previously introduced rule applicable to cases of bitwise copy using either `std::memcpy` or `std::memmove`.[52] Note that user-defined bitwise copy (e.g., using **`unsigned char`**s directly) still has UB as it does not begin the lifetime of the target object. Also note that a valid bitwise copy using `std::memcpy` or `std::memmove` implicitly creates a new object — i.e., has copy-construction semantics — even if the only *non*deleted trivial function conferring trivially copyable status is one of the assignment operators.

As an example of the kind of UB that might occur via optimization in C++11/14, consider that a compiler may cache the result of reading a **`const`** data member in a register, as the value may not change within the lifetime of the object in a well-defined program. Any attempt to replace that object via `std::memcpy` might be respected, but the stale value in the register will not necessarily be invalidated, so subsequent reads of that data member might produce the old value. A C++20 compiler, conversely, must now also allow for the possibly that such an object might be overwritten by `std::memcpy` or placement `::`**`operator new`** and inhibit this particular optimization in such cases.[53] For compilers implementing Standards prior to C++20, we can reduce risks in generic code by checking that a type is both trivially copyable *and* either copy assignable or move assignable (e.g., using `std::is_assignable`) before attempting to use `std::memcpy`, so as to avoid UB associated with **`const`**- and reference-qualified *non***`static`** data members. Note that the additional check for assignability will reject trivially copyable types having no publicly invocable assignment operators, even absent any **`const`** or reference *non***`static`** data members. Such rejection might better reflect the semantic intent of the class author anyway; see *Potential Pitfalls — Using `memcpy` on objects having **`const`** or reference subobjects* on page 489.

[52] See **smith20**.

[53] See CWG issue 1776; **finland13**.