Remove Deprecated Volatile Features From C++26

Proposal to remove easily misunderstood feature

Document #:	P2866R1
Date:	2023-09-15
Project:	Programming Language C++
Audience:	Library Evolution SG1 Concurrency SG22 C Interoperability
Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

C++ has deprecated a number of features related to `volatile` semantics in both the core language specification and in the library specification. This paper proposes removing those features from C++26.

2 Revision history

R1: September 2023 (midterm mailing)

- Removed revision history's redundant subsection numbering
- Note passed EWG review, but awaiting LEWG confirmation before passing to Core
- Add SG22 C Interoperability to target audience
- Provide the missing Library Analysis
- Analyze the remaining structured binding dependency on `volatile` in the library
- Wording updates
  - Applied initial Core wordsmith preview from Jens Maurer
  - Rebased onto latest working draft, [N4958]
  - Updated stable label cross-reference to C++23

R0: May 2023 (pre-Varna mailing)

Original version of this document, extracted from the C++23 proposal [P2139R2].

Key changes since that earlier paper:

- Combines core and library updates in a single paper
- C++23 undeprecated compound assignment
- Rebased wording onto N4944

3 Introduction

At the start of the C++23 cycle, [P2139R2] tried to review each deprecated feature of C++ to see which we would benefit from actively removing and which might now be better undeprecated. Consolidating all this analysis into one place was intended to ease the (L)EWG review process but in return gave the author so much feedback that the next revision of the paper was not completed.

For the C++26 cycle, a much shorter paper, [P2863R1], will track the overall analysis, but for features that the author wants to actively progress, a distinct paper will decouple progress from the larger paper so that the delays on a single feature do not hold up progress on all.

This paper takes up the deprecated operations on volatile types, D.5 [depr.volatile.type], and the associated deprecated library features.

4 Background

The `volatile` keyword is an original part of the C legacy for C++, and describes constraints on programs intended to model hardware changing values beyond the program's control. As this entered the type system of C++, certain interactions were discovered to be troublesome, and latent bugs that could be detected at the time of program translation go unreported. [P1152R4] breaks down each context where the `volatile` keyword can be used, and deprecated for C++20 those uses that are unconditionally dangerous, or serve no good purpose.

Following the C++20 deprecations, the C committee looked to adopt a similar stance on `volatile` and were given feedback that a number of vendors were strongly opposed to the deprecation of compound-assignment operators, as among other reasons, many hardware APIs and device drivers would expect to use volatile compound assignment to communicate with their devices. This subset of the deprecated functionality was undeprecated for C++23 by [P2327R1], followed by further undeprecations in [CWG2654].

5 Feature Analysis

5.1 Core language

A quick micro-analysis suggests the main concerns of the first two paragraphs are read/modify/write operations, where by the nature of volatile objects, the value being rewritten may have changed since read and modified. This kind of pattern is most likely in old (pre-C++11) code using `volatile` as a poor proxy for atomic. Since we will have well over a decade of real atomic support in the language when C++26 ships, it could be desirable to further encourage such code (when compiled in the latest dialect) to adapt to the memory model and its stronger guarantees.

The third paragraph addresses function arguments and return values. These are temporary or elided objects created entirely by the compiler, and guaranteed to not display the uncertainty of value implied by the `volatile` keyword. As such, any use is redundant and misleading, so it would be helpful to remove this facility sooner rather than later, and have one fewer oddity to teach when learning (and understanding) the language. The biggest concern would be for compatibility with C code, that may still use this feature in its headers. To mitigate, me may consider removing volatile function parameters and return values for only functions with extern “C++” linkage.

The fourth paragraph considers the `volatile` qualifier in structured bindings, and can affect only code written since C++17, that will have been deprecated as long as it was non-deprecated when C++23 is published. It would be good to remove this now, before more deprecated code is written.

After 6 years of deprecation warnings, and the potential to diagnose hard-to-reproduce latent bugs for users to fix, the recommendation is to remove support for these deprecated use cases from C++26. It would also be possible to review each of the 4 noted usages separately, and remove only the features with lowest risk from removal, notably paragraphs 3 and 4.

5.2 Library

There are three distinct feature sets deprecated as part of the deprecating volatile work for C++20. Both `std::tuple` and `std::variant` have an API to query how many elements or alternates a type contains, and another to query what the type of a given element or alternate is. These APIs support volatile-qualified `tuple` and `variant` types, yet there has never been a corresponding `get` API to retrieve the value of that type, making these interfaces largely redundant.

The other use of `volatile` in the standard library is as part of the atomic APIs, and several overloads for `volatile` non-lock-free `atd::atomic` that should be constrained to not exist, per the primary library specification, remain deprecated in Annex D.

5.2.1 Deployment experience

By testing the following program with all of the latest compilers and Standard Library implementations available through Godbolt Compiler Explorer, we discovered that none of the existing library implementations are warning on use of the deprecated `tuple` and `variant` APIs. Deeper analysis may be needed to confirm whether this is a library issue, or whether usage is something the compiler finds difficult to warn about use of the `[[deprecated]]` attribute.

```
#include <tuple>
#include <type_traits>
#include <variant>

using TypeT = std::tuple<int, char, float> volatile;
using TypeV = std::variant<int, char, float> volatile;

static_assert(std::is_same_v<std::tuple_element<0, TypeT>::type, int volatile>);
static_assert(std::is_same_v<std::tuple_element_t<0, TypeT>, int volatile>);
```

```
static_assert(std::is_same_v<std::variant_alternative<0, TypeV>::type, int volatile>);
static_assert(std::is_same_v<std::variant_alternative_t<0, TypeV>, int volatile>);
```

6 C++23 Feedback

6.1 Initial EWG review

The following feedback was provided when this core language feature was originally discussed in the EWG telecon on May 30, 2020.

This clause is effectively four different sub-features, that were reviewed and polled independently. The author offered to pull this whole section out into another paper if there were concerns about processing a complex topic in this simplified omnibus paper (which has effectively happened in this paper), but there was relatively little contention throughout the discussion, so it will remain here for now.

Some concerns were raised that by removing some of these features, we would be creating inconsistencies between the treatment of `const` and `volatile` in the language. Others suggested that this was a good thing, and that one of the early concerns Bjarne expressed about the design and evolution of C++ is that there was too much consistency in the treatment of these two qualifiers that do different things in practice.

It was noted several times that volatile qualifiers on locally scoped variables, such as function arguments, rarely means what naive users expect them to mean, and can be freely ignored by an optimizing compiler. By removing support for some of those declarations, we make it harder to write misleading (but otherwise correct) code.

6.2 Subsequent feedback

Following feedback from WG14 and their progress for C23, reading the deprecated result of compound-assignment to a volatile lvalue for the bitwise operators was undeprecated for C compatibility in C++23 by [P2327R1]. Subsequently, responding to NB comment US 16-045, the reading the result of the remaining compound-assignment operators was undeprecated by [CWG2654], reintroducing a potential C incompatibility in favor of consistency and a simpler language.

7 Proposed Changes for C++26

This paper proposes removing from C++26 all the deprecated features regarding the use of `volatile`.

7.1 Core language

Remove the following language interactions:

- — increment and decrement operators on volatile lvalues
- — `volatile` qualifier on non-reference function parameters
- — `volatile` qualifier on non-reference function return types
- — volatile-qualified structured bindings

In addition, built-in assignment operator functions for volatile lvalues should be declared to return `void`. C++23 deprecates calling assignment operators with volatile lvalues, unless they are a discarded value expression, or an unevaluated operand. We can enforce this by simply removing the return value from the function signature. However, this is a bigger change than strictly necessary, as it further removes the non-deprecated use case as an unevaluated operand. This is the recommended choice as it means that code written to detect valid return types using SFINAE constraints will report only valid code; otherwise, we would risk breaking metaprograms.

7.2 Library

7.2.1 `tuple` API

Remove deprecated `tuple` traits of volatile-qualified types. I tried and failed to demonstrate the need to support a customization point of structured bindings of volatile-qualified types. Structured bindings of volatile-qualified `std::tuple` objects already fail to compile due to a lack of `get` support, and my test cases of tying to set up a user-customization for their own types compiled without the `volatile` specializations.

7.2.2 `variant` API

Remove deprecated `variant` interface.

7.2.3 Non-lock-free atomics

Remove deprecated `volatile` members of `atomic<T>` when `atomic<T>::is_lock_free` is `false`.

7.2.4 Change `volatile atomic` interface to match non-atomic types

`atomic<`*`integal-type`*`>` and `atomic<`*`pointer-type`*`>` should remove volatile-qualified increment and decrement operators.

All non-deleted volatile-qualified `atomic<T>` assignment-operators should change their return value to `void`, although this may be an ABI-breaking change.

7.3 Concerns raised by Core/Library Interaction

One corner case retains a core language dependency on `tuple_size` and `tuple_element` through structured bindings. Let us build up an example to demonstrate the specific corner, that this paper proposes removing (without deprecation) from the Core specification, allowing LWG to remove the partial specializations of `tuple_size` and `tuple_element` for volatile-qualified types.

7.3.1 Tailored structured binding

First, demonstrate the feature that has been deprecated for the last two editions of the standard. Below, we create a type in namespace test, `struct Binding`, that is a simple aggregate-like class that defines all the customization points necessary to use that type in a structured binding.

```
#include <tuple>
#include <type_traits>
#include <utility>

namespace test {
struct Binding {
   int   data{};
   char  code{};
   float value{};

   Binding() = default;
   Binding(Binding const&) = default;
};

template<unsigned N>
auto get(Binding& obj) {
    if constexpr (0 == N) {
        return obj.data;
    }
```

```
    if constexpr (1 == N) {
        return obj.code;
    }

    if constexpr (2 == N) {
        return obj.value;
    }

    std::unreachable();
}

template<unsigned N>
auto get(Binding const & obj) {
    if constexpr (0 == N) {
        return obj.data;
    }

    if constexpr (1 == N) {
        return obj.code;
    }

    if constexpr (2 == N) {
        return obj.value;
    }

    std::unreachable();
}

}

namespace std {

template <>
struct tuple_size<test::Binding> : std::integral_constant<unsigned, 3> {};

template <>
struct tuple_element<0, test::Binding> {
   using type = int;
};

template <>
struct tuple_element<1, test::Binding> {
   using type = char;
};

template <>
struct tuple_element<2, test::Binding> {
   using type = double;
};

}
```

```
int main() {
   test::Binding x = {};
   auto [a,b,c] = x;
   return a;
}
```

Here, explicitly specialize the templates `std::tuple_size` and `std::tuple_element` for our class type, and add `get` overloads in its own namespace that are found via ADL. This is a basic demonstration of supporting our own type in a structured binding.

7.3.2 Deprecated `volatile` structured binding

Then we update the `main` program as follows:

```
int main() {
   test::Binding x = {};
   auto volatile [a,b,c] = x;
   return a;
}
```

This program gives warnings that this use of `volatile` is deprecated, and is the usage this paper proposes removing.

7.3.3 Non-deprecated structured binding to a volatile-qualified lvalue

Next, we bind from a volatile-qualified lvalue instead:

```
int main() {
   test::Binding volatile x = {};
   auto [a,b,c] = x;
   return a;
}
```

This fails to compile as the structure binding wants to make a copy of `x`, but there is no constructor that can take a volatile-qualified argument, so we update `Binding` as follows:

```
struct Binding {
   int   data{};
   char  code{};
   float value{};

   Binding() = default;
   Binding(Binding const&) = default;
   Binding(Binding const volatile &) {}  // construct with default intiializers
};
```

By overloading with the `const volatile &` copy constructor, the program with the volatile-qualified `x` now compiles:

```
int main() {
   test::Binding volatile x = {};
   auto [a,b,c] = x;    // This will compile now
   return a;
}
```

Note that this use of `volatile` is not deprecated, so should remain supported. However, we may be wondering if this uses `tuple_size` on a volatile-qualified type? So let us test that!

Define explicit specializations of `tuple_size` for all cv-qualified variations of `test::Binding`, so that only the unqualified version provides the integral constant base characteristics required by the structured binding protocol:

```
template <>
struct tuple_size<test::Binding> : std::integral_constant<unsigned, 3> {};

template <>
struct tuple_size<test::Binding const> {};              // canary

template <>
struct tuple_size<test::Binding volatile> {};           // canary

template <>
struct tuple_size<test::Binding const volatile> {};     // canary
```

If structured binding attempted to find `tuple_size` of a volatile-qualified object, it should fail to compile; however, our program continues to compile just fine, indicating that structured bindings are querying the non-volatile-qualified copy of `x` used for the by-value binding. Hence, this valid use of `volatile` is not impacted by the proposal to remove `volatile` support from `tuple_size` (and `tuple_element`).

7.3.4 Binding by reference to a volatile-qualified lvalue

Now, let us try to make a structured binding by-reference to a volatile lvalue. Note that according to the core language wording this is well-defined behavior that is not deprecated in C++23.

```
int main() {
   test::Binding volatile x = {};
   auto & [a,b,c] = x;
   return a;
}
```

Here we find that the structured binding relies upon the deprecated library value `tuple_size<volatile test::Binding>::value`, suggesting that our proposal would break this code. However, retaining the volatile-qualified support for `tuple_size` is not yet enough for the above code to compile, even in the original C++17 specification that preceded the deprecations. The remaining issue is that our `get` overloads do not accept references to volatile-qualified types. Hence, to complete our implementation and as required for C++17, not affected by any changes proposed by this paper, we must add the ADL-discoverable volatile-qualified overloads:

```
template<unsigned N>
auto get(Binding volatile & obj) {
    if constexpr (0 == N) {
        return obj.data;
    }

    if constexpr (1 == N) {
        return obj.code;
    }

    if constexpr (2 == N) {
        return obj.value;
    }

    std::unreachable();
}

template<unsigned N>
```

```
auto get(Binding const volatile & obj) {
    if constexpr (0 == N) {
        return obj.data;
    }

    if constexpr (1 == N) {
        return obj.code;
    }

    if constexpr (2 == N) {
        return obj.value;
    }

    std::unreachable();
}
```

Adding these two overloads is sufficient to create a structured binding by-reference to our volatile-qualified lvalue. Note that the `get` overloads in namespace `std` for native arrays, `std::array`, `std::pair`, and `std::tuple` do not support volatile-qualified objects, and never have. Hence, support for reference bindings to volatile lvalues has only ever been supported for user-provided types that supply the necessary ADL-discoverable overloads of `get`.

If we adopt this proposal to remove support for volatile-qualified types in the `tuple` metafunctions, then users will have to add their own specializations for `tuple_size` and `tuple_element` for their own type in addition to their existing `get` overloads.

```
template <>
struct tuple_size<test::Binding> : std::integral_constant<unsigned, 3> {};

template <>
struct tuple_size<test::Binding const>
     : tuple_size<test::Binding>::type {};

template <>
struct tuple_size<test::Binding volatile>
     : tuple_size<test::Binding>::type {};

template <>
struct tuple_size<test::Binding const volatile>
     : tuple_size<test::Binding>::type {};
```

Note that it is not clear that users are currently permitted to specialize `tuple_size` in this way, nor is it clear that they are not allowed. However, if such specializations are not allowed, the original example and all like it are also not allowed, so there would be no breakage of well-defined code under this proposal.

7.3.5 Proposed resolution

A proposal that we pre-emptively reject is disqualifying structured binding to volatile-qualified lvalues, without a period of deprecation. It may be that such a deprecation was also intended by the original paper, [P1152R4], but if so, that has not been explicitly drafted.

The only breakage that occurs by removing the `tuple_size` and `tuple_element` specializations is binding by-reference to a volatile-qualified lvalue, and that already requires a user to provide partial/explicit specializations of the primary template for their type, and provide a larger set of `get` overloads in their namespace (or as template-member functions of the class) than provided in namespace `std` for standard types, supporting volatile-qualified objects.

Assuming a user has already done all of the above so that their program fails to compile with C++26, the specification is already clear how they can fix their program: the compiler is going to look for those specializations of `tuple_size` and `tuple_element` for their type, and being a user-provided type they can provide those specializations themselves.

Hence, the recommendation of this paper is to remove the deprecated `tuple` API, and maybe add a note to the structured bindings clause to suggesting how the user may support this edge case, in addition to any Annex C wording.

8 Proposed wording changes

Make the following changes to the C++ Working Draft. All wording is relative to [N4958], the latest draft at the time of writing.

8.1 Update Core Wording

First, where we want to restrict operations to modifiable lvalues that no longer support `volatile`-qualified types, we will call out "modifiable non-volatile lvalues", which excludes all *cv*-qualifiers, so we can strike *cv*-qualification too.

Then, to eliminate special treatment of discarded value expressions for the assignment operator (but not for the updeprecated compound-assignment operators), we will change the return type to `void` when the argument is a `volatile`-qualified lvalue. We must validate that this will not break ABIs — although we believe we are safe as it should not be possible to take the address of a built-in operator, and users are responsible for their own operator overloads on their own `volatile`-qualified types. Note that this change could break code relying on the result of assigning to a `volatile`-qualified lvalue in unevaluated expressions, which was not previously deprecated.

Finally, we remove support for the `volatile` qualifier without a reference qualifier when declaring a structured binding.

8.1.1 Update core clauses

7.6.1.6 [expr.post.incr] Increment and decrement

1 The value of a postfix `++` expression is the value of its operand.

[*Note 1:* The value obtained is a copy of the original value. —*end note*]

The operand shall be a modifiable non-volatile lvalue. The type of the operand shall be an arithmetic type other than ~~*cv*~~ `bool`, or a pointer to a complete object type. ~~An operand with volatile-qualified type is deprecated; see D.5 [depr.volatile.type].~~ The value of the operand object is modified (3.1 [defns.access]) by adding `1` to it. The value computation of the `++` expression is sequenced before the modification of the operand object. With respect to an indeterminately-sequenced function call, the operation of postfix `++` is a single evaluation.

[*Note 2:* Therefore, a function call cannot intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix `++` operator. —*end note*]

The result is a prvalue. The type of the result is the ~~cv-unqualified version of the~~ type of the operand. If the operand is a bit-field that cannot represent the incremented value, the resulting value of the bit-field is implementation-defined. See also 7.6.6 [expr.add] and 7.6.19 [expr.ass].

2 The operand of postfix `--` is decremented analogously to the postfix `++` operator.

[*Note 3:* For prefix increment and decrement, see 7.6.2.3 [expr.pre.incr]. —_end note]

7.6.2.3 [expr.pre.incr] Increment and decrement

1 The operand of prefix `++` is modified (3.1 [defns.access]) by adding `1`. The operand shall be a modifiable non-volatile lvalue. The type of the operand shall be an arithmetic type other than ~~*cv*~~ `bool`, or a pointer to a completely-defined object type. ~~An operand with volatile-qualified type is deprecated; see D.5 [depr.volatile.type].~~ The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. The expression `++x` is equivalent to `x+=1`.

[*Note 1:* See the discussions of addition7.6.6 [expr.add] and assignment operators 7.6.19 [expr.ass] for information on conversions. —*end note*]

2 The operand of prefix `--` is modified (3.1 [defns.access]) by subtracting `1`. The requirements on the operand of prefix `--` and the properties of its result are otherwise the same as those of prefix `++`.

[*Note 2:* For postfix increment and decrement, see 7.6.1.6 [expr.post.incr]. —*end note*]

7.6.19 [expr.ass] Assignment and compound assignment operators

4 ...

5 An assignment whose left operand is of a volatile-qualified type is ~~deprecated (D.5 [depr.volatile.type])~~ ill-formed unless the (possibly parenthesized) assignment is a discarded-value expression ~~or an unevaluated operand (7.2.3 [expr.context])~~.

9.3.4.6 [dcl.fct] Functions

3 ...

4 The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called.

[*Note 1:* The *parameter-declaration-clause* is used to convert the arguments specified on the function call; see 7.6.1.3 [expr.call]. —*end note*]

If the *parameter-declaration-clause* is empty, the function takes no arguments. A parameter list consisting of a single unnamed parameter of non-dependent type `void` is equivalent to an empty parameter list. Except for this special case, a parameter shall not have type *cv* `void`. A parameter ~~with~~ shall not have a volatile-qualified type ~~is deprecated; see D.5 [depr.volatile.type]~~. If the *parameter-declaration-clause* terminates with an ellipsis or a function parameter pack (13.7.4 [temp.variadic]), the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where "`...`" is not part of an *abstract-declarator*, "`, ...`" is synonymous with "`...`".

[*Example 1:* The declaration

```
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a `const char*`. —*end example*]

[*Note 2:* The standard header `<cstdarg>` (17.13.2 [cstdarg.syn]) contains a mechanism for accessing arguments passed using the ellipsis (see 7.6.1.3 [expr.call] and 17.13 [support.runtime]). —*end note*]

5 The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own *parameter-declaration* (9.3 [dcl.decl]). After determining the type of each parameter, any parameter of type "array of T" or of function type T is adjusted to be "pointer to `T`". After producing the list of parameter types, any top-level ~~*cv-qualifiers*~~ const-qualifier modifying a parameter type ~~are~~ is deleted when forming the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function's *parameter-type-list*.

6 …

15 The return type shall be a non-volatile non-array object type, a reference type, or ~~*cv*~~ possibly const-qualified `void`.

[*Note 8:* An array of placeholder type is considered an array type. —*end note*]

16 A volatile-qualified return type is deprecated; see 13.7.4 [temp.variadic].

9.6 [dcl.struct.bind] Structured binding declarations

1 A structured binding declaration introduces the *identifiers* `v0`, `v1`, `v2`,… of the *identifier-list* as names of *structured bindings*. Let *cv* denote the *cv-qualifiers* in the *decl-specifier-seq* and `S` consist of the *storage-class-specifiers* of the *decl-specifier-seq* (if any). A *cv* that includes `volatile` is ~~deprecated; see D.5~~ ill-formed. First, a variable with a unique name *e* is introduced. If the *assignment-expression* in the *initializer* has array type *cv1* `A` and no *ref-qualifier* is present, *e* is defined by

attribute-specifier-seq$_{opt}$ `S` *cv* `A` *e* ;

and each element is copy-initialized or direct-initialized from the corresponding element of the *assignment-expression* as specified by the form of the *initializer*. Otherwise, *e* is defined as-if by

attribute-specifier-seq$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ *e initializer* ;

where the declaration is never interpreted as a function declaration and the parts of the declaration other than the *declarator-id* are taken from the corresponding structured binding declaration. The type of the *id-expression* *e* is called `E`.

[*Note 1:* `E` is never a reference type (7.2 [expr.prop]). —*end note*]

2 …

5 Otherwise, …

[*Example 2:*

```
struct S { mutable int x1 : 2; volatile double y1; };
S f();
const    auto [ x, y ] = f();
volatile auto [ a, b ] = f(); //error, volatile structured binding
```

The type of the *id-expression* `x` is "`int`", the type of the *id-expression* `y` is "`const volatile double`". —*end example*]

12.5 [over.built] Built-in operators

3 …

4 For every ~~pair (*T* , *vq*)~~ type *T*, where *T* is a cv-unqualified arithmetic type other than `bool` or a cv-unqualified pointer to (possibly cv-qualified) object type, there exist candidate operator functions of the form

```
~~vq~~ T& operator++(~~vq~~ T&);
T operator++(~~vq~~ T&, int);
~~vq~~ T& operator--(~~vq~~ T&);
T operator--(~~vq~~ T&, int);
```

5 …

18 For every triple (*L* , *vq*, *R*), where *L* is an arithmetic type, and *R* is a floating-point or promoted integral type, there exist candidate operator functions of the form

```
~~vq~~ L& operator=(~~vq~~ L&, R);
void operator=(volatileL&,R);
vq L& operator*=(vq L&, R);
vq L& operator/=(vq L&, R);
vq L& operator+=(vq L&, R);
vq L& operator-=(vq L&, R);
```

19 For every pair (*T* , *vq*), where *T* is any type, there exist candidate operator functions of the form

```
T*~~vq~~& operator=(T*~~vq~~&, T*);
void operator=(T* volatile &,T*);
```

20 For every pair (*T* , *vq*), where *T* is an enumeration or pointer-to-member type, there exist candidate operator functions of the form

```
~~vq~~ T & operator=(~~vq~~ T &, T);
void operator=(volatile T &,T);
```

8.1.2 Update Annex C

C.1.2 [diff.cpp23.expr] Clause 7: expressions

2 **Affected subclause:** 7.6.1.6 [expr.post.incr] and 7.6.2.3 [expr.pre.incr]

Change: Cannot increment or decrement volatile scalars

Rationale:

Effect on original feature:

3 **Affected subclause:** 7.6.19 [expr.ass]

Change: Cannot use the return value of assignment to a volatile-qualified type

Rationale:

Effect on original feature:

Clause 9: declarations [diff.cpp23.dcl]

1 **Affected subclause:** 9.3.4.6 [dcl.fct]

Change: Cannot declare volatile-qualified function parameter types and function return types

Rationale:

Effect on original feature:

2 **Affected subclause:** 9.6 [dcl.struct.bind]

Change: Cannot define a volatile-qualified structured binding

Rationale:

Effect on original feature:

C.7.4 [diff.expr] Clause 7: expressions

x **Affected subclause:** 7.6.1.6 [expr.post.incr] and 7.6.2.3 [expr.pre.incr]

Change: Cannot increment or decrement volatile scalars

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2 = x1;             // invalid C++
struct X x3;
x3 = x1;                      // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. If non-volatile semantics are required, an explicit `const_cast` can be used.

How widely used: Seldom.

y **Affected subclause:** 7.6.19 [expr.ass]

Change: Cannot use the return value of assignment to a volatile-qualified type

Rationale:

Effect on original feature:

How widely used: Seldom.

C.7.6 [diff.dcl] Clause 7: declarations

x **Affected subclause:** 9.6 [dcl.struct.bind]

Change: Cannot declare volatile-qualified function parameter types and function return types

Rationale:

Effect on original feature:

How widely used: Seldom.

8.1.3 Strike core wording from Annex D

D.5 [depr.volatile.type] Deprecated volatile types

1 Postfix `++` and `--` expressions (7.6.1.6 [expr.post.incr]) and prefix `++` and `--` expressions (7.6.2.3 [expr.pre.incr]) of volatile-qualified arithmetic and pointer types are deprecated.

[*Example 1:*

```
volatile int velociraptor;
++velociraptor; // deprecated
```

—*end example*]

2 Certain assignments where the left operand is a volatile-qualified non-class type are deprecated; see 7.6.19 [expr.ass].

[*Example 2:*

```
int neck, tail;
volatile int brachiosaur;
brachiosaur = neck;           // OK
tail = brachiosaur;           // OK
```

```
tail = brachiosaur = neck;    // deprecated
brachiosaur += neck;          // OK
```

—*end example*]

3 A function type (9.3.4.6 [dcl.fct]) with a parameter with volatile-qualified type or with a volatile-qualified return type is deprecated.

[*Example 3:*

```
volatile struct amber jurassic();
void trex(volatile short left_arm, volatile short right_arm);
void fly(volatile struct pterosaur* pteranodon);
```

—*end example*]

4 A structured binding (9.6 [dcl.struct.bind]) of a volatile-qualified type is deprecated.

[*Example 4:*

```
struct linhenykus { short forelimb; };
void park(linhenykus alvarezsauroid) {
  volatile auto [what_is_this] = alvarezsauroid;                // deprecated
  // ...
}
```

—*end example*]

8.2 Update Library Wording

8.2.1 No changes to zombie names

As all the entities being struck are overloads of identifiers that retain their original meaning, there are no new names to add to 16.4.5.3.2 [zombie.names].

8.2.2 Add Annex C Library wording

C.1.X Annex D: compatibility features [diff.cpp23.depr]

Change: Remove volatile support for volatile-qualified `tuple` and `variant` in the metafunctions `tuple_element`, `tuple_size`, `variant_alternative`, and `variant_size`.

Rationale: The library does not make extra effort to support volatile types and the support offered by just these metafunctions without support from the function `get` provided little value.

Effect on original feature: A valid C++ 2023 program using these metafunctions for volatile-qualified `tuple` or `variant` will not compile.

[*Note N*: This change does not remove support for volatile-qualified types stored in a `tuple` or `variant`, e.g., as might be returned by a structured binding to a volatile-qualified lvalue —*end note*]

Change: Remove support for operations on `volatile atomic<T>` unless `atomic<T>::is_always_lock_free` is `true`.

Rationale: ...

Effect on original feature: A valid C++ 2023 program using a such an `atomic<T>` object might not compile.

8.2.3 Strike Library wording from Annex D

D.19 [depr.tuple] Tuple

1 The header (22.4.2 [tuple.syn]) has the following additions:

```
namespace std {
  template<class T> struct tuple_size<volatile T>;
  template<class T> struct tuple_size<const volatile T>;
  template<size_t I, class T> struct tuple_element<I, volatile T>;
  template<size_t I, class T> struct tuple_element<I, const volatile T>;
}
```

```
template<class T> struct tuple_size<volatile T>;
template<class T> struct tuple_size<const volatile T>;
```

2 Let *TS* denote `tuple_size<T>` of the cv-unqualified type `T`. If the expression `TS::value` is well-formed when treated as an unevaluated operand (7.2.3 [expr.context]), then specializations of each of the two templates meet the *Cpp17TransformationTrait* requirements with a base characteristic of `integral_constant<size_t, TS::value>`. Otherwise, they have no member value.

3 Access checking is performed as if in a context unrelated to *TS* and `T`. Only the validity of the immediate context of the expression is considered.

4 In addition to being available via inclusion of the `<tuple>` (22.4.2 [tuple.syn]) header, the two templates are available when any of the headers `<array>` (24.3.2 [array.syn]), `<ranges>` (ranges.syn), or `<utility>` (22.2.1 [utility.syn]) are included.

```
template<size_t I, class T> struct tuple_element<I, volatile T>;
template<size_t I, class T> struct tuple_element<I, const volatile T>;
```

5 Let *TE* denote `tuple_element_t<I, T>` of the cv-unqualified type `T`. Then specializations of each of the two templates meet the *Cpp17TransformationTrait* requirements with a member typedef `type` that names the following type:

— for the first specialization, `add_volatile_t<TE>`, and
— for the second specialization, `add_cv_t<TE>`.

6 In addition to being available via inclusion of the `<tuple>` (22.4.2 [tuple.syn]) header, the two templates are available when any of the headers `<array>` (24.3.2 [array.syn]), `<ranges>` (ranges.syn), or `<utility>` (22.2.1 [utility.syn]) are included.

D.20 [depr.variant] Variant

1 The header (22.6.2) has the following additions:

```
namespace std {
  template<class T> struct variant_size<volatile T>;
  template<class T> struct variant_size<const volatile T>;
  template<size_t I, class T> struct variant_alternative<I, volatile T>;
  template<size_t I, class T> struct variant_alternative<I, const volatile T>;
}
```

```
template<class T> struct variant_size<volatile T>;
template<class T> struct variant_size<const volatile T>;
```

2 Let *VS* denote `variant_size<T>` of the cv-unqualified type `T`. Then specializations of each of the two templates meet the *Cpp17UnaryTypeTrait* requirements with a base characteristic of `integral_constant<size_t, VS::value>`.

```
template<size_t I, class T> struct variant_alternative<I, volatile T>;
template<size_t I, class T> struct variant_alternative<I, const volatile T>;
```

3 Let *VA* denote `variant_alternative<I, T>` of the cv-unqualified type `T`. Then specializations of each of the two templates meet the *Cpp17TransformationTrait* requirements with a member typedef `type` that names the following type:

— for the first specialization, `add_volatile_t<VA::type>`, and
— for the second specialization, `add_cv_t<VA::type>`.

D.30.2 [depr.atomics.volatile] Volatile access

1 If an atomic specialization has one of the following overloads, then that overload participates in overload resolution even if `atomic<T>::is_always_lock_free` is `false`:

```
void store(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
T operator=(T desired) volatile noexcept;
T load(memory_order order = memory_order::seq_cst) const volatile noexcept;
operator T() const volatile noexcept;
T exchange(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
                           memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
                             memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
                           memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
                             memory_order order = memory_order::seq_cst) volatile noexcept;
T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;
T operator op=(T operand) volatile noexcept;
T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) volatile noexcept;
```

D.30.3 [depr.atomics.nonmembers] Non-member functions

```
template<class T>
  void atomic_init(volatile atomic<T>* object, typename atomic<T>::value_type desired) noexcept;
template<class T>
  void atomic_init(atomic<T>* object, typename atomic<T>::value_type desired) noexcept;
```

x *Constraints:* For the `volatile` overload of this function, `atomic<T>::is_always_lock_free` is `true`.

1 *Effects:* Equivalent to: `atomic_store_explicit(object, desired, memory_order::relaxed);`

8.3 Update cross-reference for stable labels for C++23

Cross-references from ISO C++ 2023

All clause and subclause labels from ISO C++ 2023 (ISO/IEC 14882:2023, *Programming Languages — C++*) are present in this document, with the exceptions described below.

container.gen.reqmts *see*
container.requirements.general

depr.atomics.volatile *removed*
depr.res.on.required *removed*
depr.tuple *removed*
depr.variant *removed*
depr.volatile.type *removed*

9 Acknowledgements

Thanks to Michael Parks for the pandoc-based framework used to transform this document's source from Markdown.

Thanks again to Matt Godbolt for maintaining Compiler Explorer, the best public resource for C++ compiler and library archaeology, especially when researching the history of deprecation warnings!

Thanks to Jens Maurer for the initial wording review and corrections.

10 References

[CWG2654] US. 2022-11-03. Un-deprecation of compound volatile assignments.
https://wg21.link/cwg2654

[N4958] Thomas Köppe. 2023-08-14. Working Draft, Programming Languages -- C++.
https://wg21.link/n4958

[P1152R4] JF Bastien. 2019-07-22. Deprecating volatile.
https://wg21.link/p1152r4

[P2139R2] Alisdair Meredith. 2020-07-15. Reviewing Deprecated Facilities of C++20 for C++23.
https://wg21.link/p2139r2

[P2327R1] Paul M. Bendixen, Jens Maurer, Arthur O'Dwyer, Ben Saks. 2021-10-04. De-deprecating volatile compound operations.
https://wg21.link/p2327r1

[P2863R1] Alisdair Meredith. 2023-08-16. Review Annex D for C++26.
https://wg21.link/p2863r1