Proposal for C23
WG14 n2743

Title:	Volatile C++ Compatibility
Author, affiliation:	Robert C. Seacord, NCC Group
Date:	2021-5-18
Proposal category:	Feature
Target audience:	Implementers
Abstract:	Maintain C++ compatibility by deprecating certain volatile accesses
Prior art:	C++

Volatile C++ Compatibility

Reply-to: Robert C. Seacord (rcseacord@gmail.com)
Document No: **n2743**
Reference Document: P1152R4 (http://wg21.link/P1152R4)
Date: 2021-5-18

P1152R4 deprecates some uses of volatile and was adopted by WG21 and incorporated into C++20. This paper proposes maintaining compatibility with C++ by declaring equivalent features in the C language to be obsolescent features.

Change Log

2021-5-18:

- Initial version

1. PROBLEM DESCRIPTION

1.1 Simple Assignment

Section 6.5.16 Assignment operators, paragraph 3 of the C Standards states:

> An assignment expression has the value of the left operand after the assignment, [121)] but is not an lvalue.
>
> [121)] The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

This means that the following C code contains implementation-defined behavior:

```
int a, c;
volatile int b;
a = (b = c);
```

To eliminate this implementation-defined behavior, the user could write:

```
b = c
a = b
```

indicating that `b` must be read, or

```
b = c
a = c
```

indicating that `b` must not be read.

1.2. volatile parameters and returns

Marking parameters as volatile makes sense to denote external modification through signals or `setjmp` / `longjmp`. In that sense it's similar to `const`-qualified parameters: it has clear semantics within the function's implementation. However, it leaks function implementation information to the caller. It also has no semantics when it comes to calling convention because it is explicitly ignored (and must therefore have the same semantics as a non-`volatile` declaration). It's much simpler to have the desirable behavior by copying a non-`volatile` parameter to an automatic stack variable

marked **volatile**. A compiler could, if stack passing is required by the ABI, make no copy at all in this case.

volatile return values are pure nonsense. Is register return disallowed? What does it mean for return value optimization? A caller is better off declaring a **volatile** automatic stack variable and assigning the function return to it, and the caller will be none the wiser.

1.2. Compound Assignment

Supplying an lvalue of volatile-qualified type as an operand to compound assignment can mislead an experienced programmer into believing that the operation itself becomes compounded and only touches the memory once. Read-modify-write operations imply touching the volatile object more than once per byte because that's fundamentally how hardware works. These RMW operations are therefore misleading and should be spelled out as separate read ; modify ; write operations.

```
UART1->UCSR0B |= (1<<UCSZ01); // compound operation (obsolescent)

UART1->UCSR0B = UART1->UCSR0B | (1<<UCSZ01); // revised
```

1.2. pre- / post-increment & decrement

Pre- and post-increment (**++**) and decrement (**--**) of an lvalue of volatile-qualified type is fundamentally a read-modify-write operation that accesses the lvalue more than once per byte.

1.2. **volatile** atomic

volatile can tear, provides no ordering guarantees (with respect to non-**volatile** memory operations, and when it comes to CPU reordering), touches bytes exactly once, and inhibits optimizations. **atomic** cannot tear, has a full memory model, can require a loop to succeed, and can be optimized. **volatile** atomic should offer the union of these properties, but currently fails to do so:

- A non-lock-free atomic can be volatile, in which case it can tear when the issued instructions are considered.
- Read-modify-write operations are implemented as either loops which retry, locked instructions (which still perform a load and a store), as transactional memory operations, or as memory controller operations. Only the last of these can truly be said to touch each byte exactly once, and these hardware implementations are far from the norm.

Compiling the following code using x86-64 Clang 12.0.0 with **-Os**:

```
volatile _Atomic int small;
volatile _Atomic struct {
    int arr[16];
} big, BIG;

void inc() {
    small += 42;
}

void cpy() {
    big = BIG;
}
```

Produces the following instructions for the **inc** function:

```
inc:                                    # @inc
        lock            add     dword ptr [rip + small], 42
```

And the following instructions for the **cpy** function:

```
cpy:                                    # @cpy
        push    rbx
        sub     rsp, 64
        mov     rbx, rsp
        mov     edi, 64
        mov     esi, offset BIG
        mov     rdx, rbx
        mov     ecx, 5
        call    __atomic_load
        mov     edi, 64
        mov     esi, offset big
        mov     rdx, rbx
        mov     ecx, 5
        call    __atomic_store
        add     rsp, 64
        pop     rbx
        ret
```

Volatile atomic operations need to be performed as a single-copy atomic because shared memory across processes will not share the lock, resulting in a regular unprotected **memcpy**.

volatile _Atomic makes no sense at that size for hardware, nor does it work for signal handling, because a recursive lock can deadlock. **setjmp** / **longjmp** doesn't require **_Atomic**, just **volatile**. Consequently, there are no valid use cases for non-lock-free volatile **_Atomic**.

Functions that accept **volatile** atomic types are only guaranteed to succeed when the operations on the type are always lock free (that is, the atomic lock-free macros expand to an integer constant expression with value 2).

2. SUGGESTED CHANGES

The suggested changes in this section are against N2596 working draft — December 11, 2020.

6.5.16.1 Simple assignment

Add the following paragraph after paragraph 2:

Simple assignments where the left operand is an lvalue of volatile-qualified type is an obsolescent feature unless the expression is not evaluated or is a void expression (6.3.2.2).

6.5.16.2 Compound assignment

Change paragraph 4 as follows:

A compound assignment of the form **E1** op= **E2** is equivalent to the simple assignment expression **E1** = **E1** op (**E2**), except that the lvalue **E1** is evaluated only once. Accessing E1 through the use of an lvalue of volatile-qualified type is an obsolescent feature. ~~and w~~With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. If **E1** has an atomic type, compound assignment is a read-modify-write operation with **memory_order_seq_cst** memory order semantics.

6.7.6.3 Function declarators

Add the following paragraph after paragraph 12:

A return type or parameter declared to have volatile-qualified type is an obsolescent feature.

6.5.2.4 Postfix increment and decrement operators

Add the following paragraph after paragraph 3:

An operand that is an lvalue of a volatile-qualified type is an obsolescent feature.

6.5.3.1 Prefix increment and decrement operators

Add the following paragraph after paragraph 3:

An operand that is an lvalue of a volatile-qualified type is an obsolescent feature.

6.11 Future language directions

Add the following subsection:

6.11.8 Volatile Access

Simple assignments where the left operand is an lvalue of volatile-qualified type is an obsolescent feature unless the expression value is discarded or not evaluated.

For compound assignment of the form E1 op= E2, accessing E1 through an lvalue of volatile-qualified type is an obsolescent feature.

A parameter with volatile-qualified type is an obsolescent feature.

A volatile-qualified return type is an obsolescent feature.

An operand to the postfix increment and decrement operators that is an lvalue of a volatile-qualified type is an obsolescent feature.

An operand to the prefix increment and decrement operators that is an lvalue of a volatile-qualified type is an obsolescent feature.

7.17.1 Introduction

Add the following after paragraph 6:

An argument *A* that is not always lock-free and is an lvalue of a volatile-qualified type is an obsolescent feature.

7.31.10 Atomics <stdatomic.h>

Add the following after paragraph 2:

An argument of atomic type that is not always lock-free and is an lvalue of a volatile-qualified type is an obsolescent feature.

4.0 Acknowledgements

I would like to recognize the following people for their help with this work: JF Bastien and Aaron Ballman.

5.0 References

[P1152R0] JF Bastien. Deprecating volatile. 1 October 2018. URL: https://wg21.link/p1152r0

[P1152R1] JF Bastien. Deprecating volatile. 20 January 2019. URL: https://wg21.link/p1152r1

[P1152R2] JF Bastien. Deprecating volatile. 9 March 2019. URL: https://wg21.link/p1152r2P2327R0

[P1152R4] JF Bastien. Deprecating volatile. 19 July 2019. URL: http://wg21.link/P1152R4

[P1831R1] JF Bastien. Deprecating volatile: library. `12 February 2020. URL: http://wg21.link/P1831R1

[P2327R0] De-deprecating volatile compound operations http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2327r0.pdf