volatile type qualifier

Each individual type in the C type system has several *qualified* versions of that type, corresponding to one, two, or all three of the const, *volatile*, and, for pointers to object types, restrict qualifiers. This page describes the effects of the *volatile* qualifier.

Every access (both read and write) made through an lvalue expression of volatile-qualified type is considered an observable side effect for the purpose of optimization and is evaluated strictly according to the rules of the abstract machine (that is, all writes are completed at some time before the next sequence point). This means that within a single thread of execution, a volatile access cannot be optimized out or reordered relative to another visible side effect that is separated by a sequence point from the volatile access.

A cast of a non-volatile value to a volatile type has no effect. To access a non-volatile object using volatile semantics, its address must be cast to a pointer-to-volatile and then the access must be made through that pointer.

Any attempt to read or write to an object whose type is volatile-qualified through a non-volatile lvalue results in undefined behavior:

```
volatile int n = 1; // object of volatile-qualified type
int* p = (int*)&n;
int val = *p; // undefined behavior
```

A member of a volatile-qualified structure or union type acquires the qualification of the type it belongs to (both when accessed using the . operator or the -> operator):

```
struct s { int i; const int ci; } s;
// the type of s.i is int, the type of s.ci is const int
volatile struct s vs;
// the types of vs.i and vs.ci are volatile int and const volatile int
```

If an array type is declared with the volatile type qualifier (through the use of typedef), the array type is not volatile-qualified, but its element type is.	(until C23)
An array type and its element type are always considered to be identically volatile-qualified.	(since C23)

If a function type is declared with the volatile type qualified (through the use of typedef), the behavior is undefined.

```
typedef int A[2][3];
volatile A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of volatile int
int* pi = a[0]; // Error: a[0] has type volatile int*
void *unqual_ptr = a; // OK until C23; error since C23
// Notes: clang applies the rule in C++/C23 even in C89-C17 modes
```

In a function declaration, the keyword volatile may appear inside the square brackets that are used to declare an array type of a function parameter. It qualifies the pointer type to which the array type is transformed.

The following two declarations declare the same function:

```
void f(double x[volatile], const double y[volatile]);
void f(double * volatile x, const double * volatile y);
```

(since C99)

A pointer to a non-volatile type can be implicitly converted to a pointer to the volatile-qualified version of the same or compatible type. The reverse conversion can be performed with a cast expression.

```
int* p = 0;
volatile int* vp = p; // OK: adds qualifiers (int to volatile int)
p = vp; // Error: discards qualifiers (volatile int to int)
p = (int*)vp; // OK: cast
```

Note that pointer to pointer to T is not convertible to pointer to pointer to volatile T; for two types to be compatible, their qualifications must be identical:

```
char *p = 0;
volatile char **vpp = &p; // Error: char* and volatile char* are not compatible types
char * volatile *pvp = &p; // OK, adds qualifiers (char* to char*volatile)
```

Uses of volatile

1) static volatile objects model memory-mapped I/O ports, and static const volatile objects model memory-mapped input ports, such as a real-time clock:

```
volatile short *ttyport = (volatile short*)TTYPORT_ADDR;
for(int i = 0; i < N; ++i)
    *ttyport = a[i]; // *ttyport is an lvalue of type volatile short
```

2) static volatile objects of type sig_atomic_t are used for communication with signal handlers.
3) volatile variables that are local to a function that contains an invocation of the setjmp macro are the only local variables guaranteed to retain their values after longjmp returns.
4) In addition, volatile variables can be used to disable certain forms of optimization, e.g. to disable dead store elimination or constant folding for microbenchmarks.

Note that volatile variables are not suitable for communication between threads; they do not offer atomicity, synchronization, or memory ordering. A read from a volatile variable that is modified by another thread without synchronization or concurrent modification from two unsynchronized threads is undefined behavior due to a data race.

Keywords

volatile

Example

demonstrates the use of volatile to disable optimizations

Run this code

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    clock_t t = clock();
    double d = 0.0;
    for (int n = 0; n < 10000; ++n)
        for (int m = 0; m < 10000; ++m)
            d += d * n * m; // reads from and writes to a non-volatile
    printf("Modified a non-volatile variable 100m times. "
           "Time used: %.2f seconds\n",
           (double)(clock() - t)/CLOCKS_PER_SEC);

    t = clock();
    volatile double vd = 0.0;
    for (int n = 0; n < 10000; ++n)
        for (int m = 0; m < 10000; ++m) {
            double prod = vd * n * m; // reads from a volatile
            vd += prod; // reads from and writes to a volatile
        }
    printf("Modified a volatile variable 100m times. "
           "Time used: %.2f seconds\n",
           (double)(clock() - t)/CLOCKS_PER_SEC);
}
```

Possible output:

```
Modified a non-volatile variable 100m times. Time used: 0.00 seconds
Modified a volatile variable 100m times. Time used: 0.79 seconds
```

References

- C17 standard (ISO/IEC 9899:2018):
  - 6.7.3 Type qualifiers (p: 87-90)
- C11 standard (ISO/IEC 9899:2011):
  - 6.7.3 Type qualifiers (p: 121-123)

- C99 standard (ISO/IEC 9899:1999):
    - 6.7.3 Type qualifiers (p: 108-110)
- C89/C90 standard (ISO/IEC 9899:1990):
    - 6.5.3 Type qualifiers

See also

C++ documentation for **cv (const and volatile) type qualifiers**	