# Understanding “volatile” qualifier in C | Set 1 (Introduction)

In spite of tons of literature on C language, “**volatile**” keyword is somehow not understood well (even by experienced C programmers). We think that the main reason for this is due to not having real-world use-case of a ‘*volatile*‘ variable in typical C programs that are written in high level language. Basically, unless you’re doing some low level hardware programming in C, you probably won’t use a variable while is qualified as “**volatile**“. By low level programming, we mean a piece of C code which is dealing with peripheral devices, IO ports (mainly memory mapped IO ports), Interrupt Service Routines (ISRs) which interact with Hardware. That’s why it’s not so straight forward to have a sample working C program which can easily show-case the exact effect of “*volatile*” keyword.

In fact, in this article, if we could explain the meaning and purpose of ‘*volatile*‘, it would serve as basic groundwork for further study and use of ‘volatile’ in C. To understand ‘volatile’, we first need to have some background about what a compiler does to a C program. At high level, we know that a compiler converts C code to Machine code so that the executable can be run without having actual source code. Similar to other technologies, compiler technology had also evolved a lot. While translating Source code to Machine code, compilers typically try to optimize the output so that lesser Machine code needs to be executed finally. One such optimization is removing unnecessary Machine code for accessing variable which is not changing from Compiler’s perspective. Suppose we have the following code:

```
uint32 status = 0;

while (status == 0)
{
  /*Let us assume that status isn't being changed
  in this while loop or may be in our whole program*/

  /*So long as status (which could be reflecting
  status of some IO port) is ZERO, do something*/
}
```

Run on IDE

An optimizing Compiler would see that *status* isn’t being changed by *while* loop. So there’s no need to access *status* variable again and again after each iteration of loop. So the Compiler would convert this loop to a infinite loop i.e. *while (1)* so that the Machine code to read *status* isn’t needed. Please note that compiler isn’t aware of that *status* is a special variable which can be changed from outside the current program at any point of time e.g. some IO operation happened on a peripheral device for which device IO port was memory mapped to this variable. So in reality, we want complier to access *status* variable after every loop iteration even though it isn’t modified by our program which is being compiled by Compiler.

One can argue that we can turn-off all the compiler optimizations for such programs so that we don't run into this situation. This is not an option due to multiple reasons such as
A) Each compiler implementation is different so it's not a portable solution
B) Just because of one variable, we don't want to turn of all the other optimizations which compiler does at other portions of our program.
C) By turning off all the optimizations, our low level program couldn't work as expected e.g. too much increase in size or delayed execution.

That's where "**volatile**" comes in picture. Basically, we need to instruct Compiler that *status* is special variable so that no such optimization are allowed on this variable. With this, we would define our variable as follows:

```
volatile uint32 status = 0;
```

Run on IDE

For simplicity of explanation purpose, we chose the above example. But in general, **volatile** is used with pointers such as follows:

```
volatile uint32 * statusPtr = 0xF1230000
```

Run on IDE

Here, *statusPtr* is pointing to a memory location (e.g. for some IO port) at which the content can change at any point of time from some peripheral device. Please note that our program might not have any control or knowledge about when that memory would change. So we would make it "**volatile**" so that compiler doesn't perform optimization for the *volatile* variable which is pointed by *statusPtr*.

In the context of our discussion about "**volatile**", we quote C language standard i.e. ISO/IEC 9899 C11 – clause 6.7.3

*"An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects."*

*"A volatile declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions."*

Basically, C standard says that "**volatile**" variables can change from outside the program and that's why compilers aren't supposed to optimize their access. Now, you can guess that having too many '**volatile**' variables in your program would also result in lesser & lesser compiler optimization. We hope it gives you enough background about meaning and purpose of "volatile".

From this article, we would like you to take-away the concept of "**volatile variable –> don't do complier optimization for that variable**"!

The following article explains volatile through more examples.
Understanding "volatile" qualifier in C | Set 2 (Examples)

If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

C/C++ Puzzles

### Related Posts:

- __attribute__((constructor)) and __attribute__((destructor)) syntaxes in C
- Printing source code of a C program itself
- isalpha() and isdigit() functions in C with example
- Nested printf (printf inside printf) in C
- Assigning an integer to float and comparison in C/C++
- Counts of distinct consecutive sub-string of length two using C++ STL
- auto_ptr, unique_ptr, shared_ptr and weak_ptr
- Interesting Facts in C Programming