**Qualifiers in C**

* Constant
* Volatile

**Constant**

The addition of a 'const' qualifier indicates that the (relevant part of the) program may not modify the variable. Such variables may even be placed in read-only storage (cf. section ""). It also allows certain kinds of optimizations, based on the premise that the variable’s value cannot change. Since 'const' variables cannot change their value during runtime (at least not within the scope and context considered), they must be initialized at their point of definition.

const int i = 5;

int const i = 5; //this is also applicable

Order becomes important when composite types with pointers are used i.e

**const int \*pi**

* Can be pointed at multiple variables.
* Cannot be used to assign a new value to a variable via indirection.

**int \*const ip**

* Can only be pointed to a single variable (done at initialization)
* Can be used to assign a new value to a variable via indirection.

**const int \* const pip**  
(*Yes, it is possible to declare a variable of this type too.*)

* Can only be pointed to a single variable (done at initialization)
* Cannot be used to assign a new value via indirection

***Example source code***

int main(void) {

int x = 1;

int y = 2;

const int \*pi;

pi = &x;

pi = &y; // Can be pointed to another variable

\*pi = 23; // <----- NOT ALLOWED

Printf(“\n\*pi = %d\n”,\*pi); // <--- Value can be accessed

int \* const ip = &x;

\*ip = 7; // <----- Can assign a value via indirection

ip = &y; // <----- NOT ALLOWED

Printf(“\n\*pi = %d\n”,\*pi); // <---- Value can be accessed

const int \* const pip = &y;

\*pip = 42; // <------ NOT ALLOWED

pip = &x; // <------ NOT ALLOWED

Printf(“\n\*pip = %d\n”,\*pip);

return 0;

}

const int \* p means "p points to an int that is const"  
int \* const p means "p is a const pointer to an int"

**Volatile**

The qualifier 'volatile' is normally avoided, understood only marginally, and quite often forgotten. It indicates to the compiler, that a variable may be modified outside the scope of the program. Such situations may occur for example in multitasking/-threading systems, when writing drivers with interrupt service routines, or in embedded systems, where the peripheral registers may also be modified by hardware alone. The following fragment is a classical example of an endless loop:

int ready = 0;

while (!ready);

An aggressively optimizing compiler may very well create a simple endless loop If we now add 'volatile', indicating that the variable may be changed out of context, the compiler is not allowed to eliminate the variable entirely:

volatile int ready = 0;

while (!ready);

The basic principle is simple: Every time when a variable is used in more than one context, qualify it

with 'volatile':

* Whenever you use a common variable in more than one task or thread;
* Whenever you use a variable both in a task and one or more interrupt service routines;
* Whenever a variable corresponds to processor-internal registers configured as input (consider the processor or external hardware to be an extra context).

**More explanation**

The **volatile** qualifier maintains consistency of memory access to data objects. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed. The **volatile** qualifier declares a data object that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock). The compiler is thereby notified not to apply certain optimizations to code referring to the object.

Take the example of a status register...we are reading the value of a status register in a variable for every interval of 1 ms..so in this case what the compiler will do it will read from the register after certain iterations(cache memory) to improve the performance or speed but through hardware the value may change so it is required to read from the memory always before loading..to achieve this we have to declare that variable as volatile.

*One more clear example used in delay*

The volatile type qualifier tells the compiler not to optimize a particular variable, regardless of the optimization settings for the compiler. The compiler will generate code to reread the value from memory, every time the variable is accessed (instead of reading the value from a [CPU[http://images.intellitxt.com/ast/adTypes/mag-glass_10x10.gif](http://forums.devshed.com/)](http://forums.devshed.com/) register, where possible). Normally, you'd specify the volatile type if the value of the variable will be changed by another program, or an interrupt, or something else outside your program.   
  
Another situation would be, if you wanted to code a delay/sleep function. Some libraries for embedded systems have no delay function, so you're forced to code your own, something like this:

#define SCALE 1000

void delay(int seconds) {

int n, max;

max = seconds \* SCALE;

for (n = 0; n <= max; n++)

;

}

With optimizations turned on, the compiler may detect that you're not doing anything useful in the for loop. Therefore, it may compile the code, as though you'd written:

#define SCALE 1000

void delay(int seconds) {

int n, max;

n = seconds \* SCALE;

}

This is clearly the wrong thing, since the code does not pause at all! To prevent this from happening, you would write something like this:

#define SCALE 1000

void delay(int seconds) {

volatile int n, max;

max = seconds \* SCALE;

for (n = 0; n <= max; n++)

;

}

The volatile declaration will prevent the compiler from optimizing the code and it will compile the code, exactly the way you intended it to work. Hope this helps!

**Combining qualifiers**

const volatile unsigned int \*p = 0x10;

This means I cannot change the value (within my current scope and context), and the value may be changed out of context.

const volatile unsigned int \* const ptcvi = 0xFFFFFFCAUL;

The initialization value is hexadecimal, unsigned long, and taken from an imaginary embedded processor. The pointer is const, so I cannot change the pointer. And the value it points to is "const volatile unsigned int": I cannot change the value (within my current scope and context), and the value may be changed out of context.

The const modifier means that this code cannot change the value of the variable, but that does not mean that the value cannot be changed by means outside this code.