Type qualifiers in C

Abstract

Type qualifiers are part of the C language since 1989 (`const` and `volatile`), 1999 (`restrict`), respectively 2011 (`_Atomic`). They are used to qualify types, modifying the properties of variables in certain ways. Since qualifiers are one of the lesser-understood features of the language, this article aims at experienced C programmers, and explains the reasoning behind these qualifiers.

Introduction

Since 'Standard C' (C89), all variables are considered "unqualified" if none of the available qualifiers is used in the definition of the variable. Additionally, since all four qualifiers are completely independent[1] from one another, for each unqualified simple type, we may have fifteen ($2^4 - 1$) forms of qualified types[2].

Beware that qualifiers change the properties of their variables only for the scope and context in which they are used. A variable declared '`const`' is a constant only as far as the current scope and context is concerned. If we widen the scope (and regard, for instance, the callers of the function) or the context (for instance, other threads or tasks, interrupt service routines, or different autonomous systems), the variable may very well be not constant at all. For '`volatile`', '`restrict`' and '`_Atomic`' similar arguments exist.

Consider calling `'memcpy'`, prototyped

```
void *memcpy(void * restrict dest, const void * restrict src, size_t len);
```

This function may very well be called with a source pointer pointing to a regular (non-read-only) part of your memory. What you can safely assume is this: The implementation will not write into the area pointed to by the `src` pointer, as long as you observe the description ("overlap causes undefined behavior"), and your pointers are (for the purpose of this context) the only pointers to the areas involved (i.e. `dest` and `src`, because of the `restrict` keyword).

Only lvalues

The standard states that "The properties associated with qualified types are meaningful only for expressions that are lvalues." Informally, this means that whenever expressions using qualified typed objects are used as "on the right-hand side of an assignment operator[3]", they will be used just like unqualified objects. This will be of no practical consequence except for `volatile`. We'll address it later.

Const

The qualifier '`const`' is most often used in modern programs, and probably best understood. 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 "Placement"). It also allows certain kinds of optimizations, based on the premise that the variable's value cannot change. Please note, however, that "const-ness" may be cast away explicitly.

Since '`const`' variables cannot change their value (at least not within the scope and context considered) during runtime, they must be initialized at their point of definition.

Example:

```
const int i = 5;
```

[1] The C11 standard differentiates between the three C99 qualifiers and '`_Atomic`': A "qualified variable" is not atomic; if a variable is both atomic and otherwise-qualified, the standard talks about a "qualified, atomic variable".
[2] Since '`restrict`' may only be used for pointers, for most variables only seven differently-qualified types exist. And be aware: Not all combinations are useful!
[3] If you are a 'language-lawyer', this informal document is not for you. Yes, I know there are several issues around lvalue / rvalue contexts, but I do not intend to clarify such issues here. If you need the formal information, use the standard itself.

An alternate form is also acceptable, since the order of type specifiers and qualifiers does not matter:

```
int const i = 5;
```

Order becomes important when composite types with pointers are used:

```
int * const cp = &i; /* const pointer to int */
const int * ptci;    /* pointer to const int */
int const * ptci;    /* pointer to const int */
```

The pointer `cp` is itself const, i.e. the pointer cannot be modified; the integer variable it points to can.
The pointer `ptci` can be modified, however, the variable it points to cannot.
Using `typedef` complicates the placement issue even more:

```
typedef int * ip_t;
const ip_t cp1 = &i; /* const pointer to int */
ip_t const cp2 = &i; /* const pointer to int!! */
```

Casting away 'const-ness' is possible, but considered dangerous. Modifying a const-qualified variable in that way is not only dangerous, but may even lead to run-time errors, if the values are placed in read-only storage:

```
const int * ptci;
int *pti, i;
const int ci;

ptci = pti = &i;
ptci = &ci;
*ptci = 5;          /* Compiler error */
pti = &ci;          /* Compiler error */
pti = ptci;         /* Compiler error */
pti = (int *)&ci; /* OK, but dangerous */
*pti = 5;           /* OK, dangerous and potential runtime error */
```

PC-Lint and similar tools, as well as some compilers, will warn you about such dangerous situations, if you will let them.

Placement

Placement of variables in actual memory is hardly standardized, because of the many requirements of specific compilers, processor architectures and requirements. But especially for `const`-qualified variables, it is a very interesting topic, and needs some discussion.
First of all, placement is compiler-specific. This means, that a compiler may specify how a programmer or system architect may direct the linker and loader as to where to place which variables or categories of variables. This may be done using extra configuration files, or using `#pragma`'s, or some other way. Refer to your compiler manual, especially when writing code for embedded systems.
If you revert to the compiler defaults, the compiler/linker/loader[4] may put non-`volatile`, `const`-qualified variables (not such combinations like 'pointer-to-const', since here the variable is a 'pointer' and non-const!) into read-only storage[5]. If the compiler has no other indication, and can oversee the full scope of the variable (for instance, a `static const int const_int = 5;` at the global level in some C source file), it may even optimize in such a way, that the variable effectively disappears (replacing each occurrence with an immediate value), though not all compilers provide this kind of optimization (yet?).
If the compiler retains the variable as such (i.e. the variable is still present in the object-file), qualified with the property '`const`', the linker combines all corresponding references throughout all modules into one, complaining if the qualifications do not match, and the loader gets to decide, where the variable is placed in memory (dynamic linkers are even more complex, and disregarded here). If a memory area with read-only storage is available, `const`-qualified variables may end up there, at the discretion of the loader.
For details, consult your compiler manuals.

[4] In most compiler tool chains, the loader is an integrated part of the linker. For several embedded systems, however, the linker only produces relocatable code segments, to be effectively placed in memory by the loader.
[5] ISO/IEC 9899:2011 Footnote 132, page 121: "The implementation may place a const object that is not volatile in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used."

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 its current visibility. 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 (Microsoft Visual Studio 6.0, Release build with full optimization):

```
$L837:

; 5    :  int ready = 0;
; 6    :  while (!ready);

  00000   eb fe          jmp    SHORT $L837
```

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);
```

This snippet becomes (using the option "favor small code"):

```
; 5    :  volatile int ready = 0;

  00004   33 c0          xor    eax, eax
  00006   89 45 fc       mov    DWORD PTR _ready$[ebp], eax
$L845:

; 6    :  while (!ready);

  00009   39 45 fc       cmp    DWORD PTR _ready$[ebp], eax
  0000c   74 fb          je     SHORT $L845
```

As you can see, even with aggressive, full optimization, the code still checks the variable every time through the loop.
Most compilers do not optimize this aggressively by default, but it is good to know that it is possible.
The ordering issues as discussed in the section for the qualifier '`const`' also apply for '`volatile`'; if in doubt, refer back to page 2.

When do you need to use 'volatile'?

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 in 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).

Does it hurt to use 'volatile' unnecessarily?

Well, yes and no. The functionality of your code will still be correct. However, the timing and memory footprint of your application will change: Your program will run slower, because of the extra read operations, and your program will be larger, since the compiler is not allowed to optimize as thoroughly, although that would have been possible.

Why don't we declare all variables 'volatile'?

Well, we partially do: On DEBUG-builds, all optimization is usually disabled. This is not quite the same, since the read operations needed extra are not necessarily inserted, but for most practical purposes,

no optimizations involving the (missing) '`volatile`' qualification are executed. This can be considered at least partially equivalent.
We don't, however, deliver DEBUG-builds to the customer: They usually are too big and too slow (among a few other properties), just like if we declare all variables to be '`volatile`'.
But, whenever in doubt, it is better to use 'volatile' unnecessarily, than to forget it when really necessary. Debugging a "missing `volatile`" declaration is a nightmare; a superfluous `volatile` doesn't normally require any debugging at all.

Pitfalls

The use of `volatile` may deceive the unwary:

- It has no influence whatsoever on atomic transactions. Variables declared with `volatile long` will still need to use two or more memory accesses to read one value if the width of a `long` variable is larger than the regular data bus width.
  This means that multitasking systems will still need external guards to protect such variables from unwanted concurrent access.
- Unrelated variable/structure accesses may be reordered by the compiler, even if one of the variables/structures is declared `volatile`:

```
volatile bool bBufferFree;
uint8_t uTxBuffer[BUFFER_SIZE];

void Clear_Buffer(void)
{
  sint16_t iLoop;
  for (iLoop = 0; iLoop < BUFFER_SIZE; iLoop++)
  {
    uTxBuffer[iLoop] = 0;
  }
  bBufferFree = true;
  return;
}
```

  Since `bBufferFree` and `uTxBuffer` have no mutual interaction within the function, a compiler is free to reorder the execution, and set `bBufferFree` to `true` before setting the `uTxBuffer` to `0`. This is probably not what the author intended, and luckily, most compilers will not reorder the execution in such a way. However, it is more than thinkable that an aggressively parallelizing compiler might legally do so.

Restrict

The qualifier '`restrict`' is special in several ways:

- The '`restrict`' qualifier may only be used for pointers to an object type;
- The optimizations enabled by using the '`restrict`' qualification have been present in most compilers, however, only on a global basis (compiler options); the extra gain to be expected is not quite as large, therefore, the necessity of using this qualifier can be considered minimal;
- The qualifier's intended use is to promote optimization. Leaving out the keyword shall not change the behavior of a program[6].

The meaning of a restrict-qualified pointer is that there's no other method in use to access the object the pointer points at except the so-qualified pointer itself. This property indicates that aliasing is not an issue for a so-qualified pointer, making optimizations possible that would otherwise be prohibited (unless enabled by other means like compiler options, `#pragma` indicators, or other means).
Hint one: If in doubt, refrain from using '`restrict`' in your code. Your code doesn't need it. If your libraries use it, be sure to read the constraints of the various functions and abide by them.

[6] ISO/IEC 9899:2011 6.7.3 ad 8, page 122: "deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior)."

Hint two: If your compiler doesn't implement '`restrict`' (yet), and doesn't even reserve the keyword, make sure to define a high-level macro, in order to prevent programmers using the name for a variable or similar:

```
#define restrict ~=~=~ /* Reserved word - usage will cause syntax error */
```

_Atomic

The new C11 qualifier _Atomic (not to be confused with the type specifier '`_Atomic(`*type*`)`') opens a whole new world of types: A new header file stdatomic.h, a whole set of macros called `ATOMIC_…`, several types including `memory_order` and `atomic_flag` as well as a truckload of functions and macros. The basic idea is to create a standardized way for C programs to work as multithreaded, pre-emptive applications and interrupt service routines, needing atomic access to variables independent from the implemented memory model, bus width, instruction set, caching limitations, etc.
The breadth of the topic at hand cannot be served right by a short section in this document. For now, I have to refer to the standard, since I'm not aware of any books really covering this subject (as of January 2012). And I'm sure that currently, not many compilers will have added full support already.
One issue is still interesting here: As with a few other additions (complex arithmetic, multithreading, and variable-length arrays), a C11-conforming implementation is not required to implement the '`_Atomic`' qualifier or any of the stdatomic.h additions; it only has to pre-define the macro `__STDC_NO_ATOMICS__` to 1, to indicate that it offers neither the header file nor the keyword.
I assume this is what many compiler vendors will have implemented quite quickly now ;-)

Combining qualifiers

In the introduction the existence of fifteen (seven, cf. footnote 2 on page 1) different qualified types for each (simple) unqualified one has been stated. Using four qualifiers (three plus '`_Atomic`'), this means that qualifiers can be combined.
In C89, each qualifier may only be used once, in C99 or later, multiple occurrences of each single qualifier are explicitly allowed and silently ignored.
But now, take '`volatile`' and '`const`': What does it mean to have a variable qualified with both:

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

OK, 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.
So, imagine a free running counter, counting upwards from 0 to 65535 (hexadecimal 0xFFFF or 16 bit), and rolling over again to 0. If this counter is automatically started by the hardware, or started by the (assembly-coded) startup-routine (outside the scope of the C program), is never stopped, and only used for relative time measurements, we have exactly this situation: The counter is read-only, so I want the compiler to supervise all programmers that they do not try to write the counter register.
At the same time, the value is constantly changed, so if I want to use the value, the compiler better make sure to re-read the value in every single case.
You can also imagine a battery backed-up clock chip, running autonomously, with values for the current date and time memory-mapped into the processors virtual memory space.
OK, such situations will not occur every day, and for many programmers they will never occur. But it is not unimaginable. And now go out and ask the most experienced C programmer you know, whether it is possible, allowed and/or useful. You'll be amazed about the answers you'll get (or maybe not).

Literature

C A Reference Manual – (5th edition[7]); Samuel P. Harbison III, Guy L. Steele Jr; Prentice hall, 2002; ISBN 0-13-089592X
ISO/IEC 9899:2011 "Information technology — Programming languages — C"

Comments, remarks and criticism are welcome.

Johan Bezem
j.bezem@computer.org
http://www.bezem.de

[7] This edition only covers up to the C99 standard. It is unknown whether a new edition for C11 is in the works.