# The `restrict` qualifier should not be a full type qualifier

*Jutta Degener*

DIN

*ABSTRACT*

In N488, Bill Homer proposes a **`restrict`** qualifier that lets programmers communicate aliasing properties of their code to the implementation. N448 suggests making **`restrict`** a type qualifier, similar to **`volatile`**, but applied to the pointer itself, not to the non-aliased data. C limits the conversions between qualified and unqualified types. In case of **`restrict`**, these limitations make no sense. A **`restrict`**-qualified type should be compatible with its not **`restrict`**-qualified counterpart.

## Introduction

A *restricted pointer* is one that, in the scope of the identifier used to refer to it, provides unaliased access to the data it points to. For example, the restricted pointer definition for **`memcpy`**

```
void *
memcpy(void * restrict s1, const void * restrict s2, size_t n)
{
        ...
}
```

promises that no user will call **`memcpy`** so that the same piece of storage will be accessed through both **`s1`** and **`s2`** within it (or through any other pointer that is not based on **`s1`** or **`s2`**, respectively).

## Differences between `volatile` and `restrict`.

In Bill Homer's proposal, **`restrict`** has the same status as **`volatile`**; both are type qualifiers. When casting pointers to qualified types, the C standard permits adding qualifiers and forbids removing them. This originates with **`const`**, and still makes a certain amount of sense with **`volatile`**; but with **`restrict`**, the model breaks down.

Fundamentally, this is because **`volatile`** makes demands where **`restrict`** gives permission to the C implementation. It is safe to demand more than one needs (hence adding **`volatile`** and **`const`** qualifiers is legal); it is dangerous to demand less (hence stripping them requires an explicit cast.)

Consider a piece of code that communicates with the outside world through two arrays of **`volatile`**- or **`restrict`**-qualified pointers, **`volatile_matrix`** and **`restrict_matrix`**.

```
char * volatile * volatile_matrix;
char * restrict * restrict_matrix;
```

What are the constraints imposed by these qualifications, and what the permissons granted?

The **`restrict`** qualifier places a (not automatically verifiable) constraint on the user of an interface: "Do not refer to the area pointed to by **`*restrict_matrix`** through a pointer that isn't based on **`*restrict_matrix`**." This gives permission to the optimizer: "As long as you don't see assignments from **`*restrict_matrix`**, you can trust that it isn't aliased."

With **`volatile`**, the directions of constraint and permission are reversed: **`volatile`** constrains the implementor: "Beware of unpredictable changes in the value **`*volatile_matrix`**," but gives permission to the user: "You can use this piece of code with **`volatile`** objects."

35

The assymmetric rules for conversion between qualified and unqualified types are geared towards the **volatile** model, not towards the **restrict** model.

To observe the conversion rules in action, let's introduce completely unqualified pointers and look at the restrictions and semantics for conversion from and to them.

```
char ** alloc_matrix();              /* allocate empty matrix */
char ** free_matrix(char ** m);      /* free() a matrix       */
```

The assignment to a pointer to **volatile**

```
volatile_matrix = alloc_matrix();
```

is safe—there's probably nothing **volatile** at ***alloc_matrix()**, but the compiler treats it as if there were, anyway. The opposite direction

```
free_matrix(volatile_matrix);      /* constraint violation  */
```

violates a constraint in 6.3.16.1 by losing a qualifier: I've asked the compiler to handle ***volatile_matrix** with special care, yet suddenly I allow it to lapse upon entry to the **free_matrix** function—sounds contradictory, and C requires an explicit cast.

With **restrict**, the opposite semantics apply. The assignment

```
restrict_matrix = alloc_matrix();
```

is unsafe. It expresses a promise I make as a programmer to the code that uses **restrict_matrix**: in **restrict_matrix**'s scope, no expressions will access ****restrict_matrix** except those visibly based on the ***restrict_matrix** pointer. The reverse form promises nothing,

```
free_matrix(restrict_matrix);      /* constraint violation  */
```

but is prohibited, since it would strip the type pointed to by **restrict_matrix** of one of its qualifiers.

**Why bother getting it right?**

In most cases, assignments to and from restricted pointers will happen on the topmost level of indirection; even more so while programmers are still getting acquainted with the new feature.

```
char * restrict str = "Hello, World!";
```

A type qualifier on the topmost level applies to the lvalue, but not to the type of the rvalue; **volatile**, **const**, and **restrict** values will be freely assignable amongst each other. Should we really need to assign an unqualified pointer to a pointer to restricted, we can always cast. So why bother? Does it matter if the semantics are a little bit off? I think yes, for two reasons.

1. In C, one cannot just 'cast a type qualifier away.' Casts are very powerful; whenever programmers have to cast, they lose almost all type safety. Normally, casts suggest that something implementation-specific is going on; in case of **restrict** they suddenly are required to express semantically inconspicuous and straightforward operations. That's a bad thing.
2. Restricted pointers are not a localized phenomenon. Code optimizes better, or can in some cases only be parallelized at all, if all pointers on all levels of indirection within a block of code are restricted, and if pointers within data structures are restricted. When objects, not just values, are concerned, an additional level of indirection arises quickly: a dynamic array of something, a function that allocates something; a function that sets a pointer for later use.

**Proposed solution**

Make a restrict-qualified pointer compatible with its not restrict-qualified counterpart. This would allow implicit conversions to proceed in both directions, adding and removing **restrict** qualifiers at will.

36