C2x Proposal: WG14 N2497

Title: **Compatibility of Pointers to Arrays with Qualifiers**
Authors: **Martin Uecker, University Medical Center Göttingen**
Aaron Ballman, GrammaTech
Jens Gustedt, University of Strasbourg
Date: **2020-02-29**

Introduction

Pointer conversion rules regarding qualifiers as specified in **6.5.16.1 Simple assignment** and referenced at various places allow to convert a pointer to a non-qualified type to a pointer to a qualified type. These rule does not work correctly for pointer to arrays, because the qualifiers is always moved to the element type of the array and then the conversion rules formally do not apply. This issue was discussed in detail in **N1923**.

Example:

```
void matrix_fun(int N, const float x[N][N]);

int N = 100;
float x[N][N];

matrix_fun(N, x); // incompatible types!
```

Most compilers largely already follow the rules suggested in the following (with some minor differences among compilers) and accept this code without warning. Also C++ has long adopted modified rules which allow these conversions. The proposed wording suggested here is a minimal change that is similar. Specifically, we propose to make the following change: An array should always be considered to have identical qualifiers as its element type (with the exception of **_Atomic**). Qualifiers do only affect lvalue conversion and should therefor not have any effect on the array itself, but all the pointer conversion rules now work as intended. The C++ standard (quoted from a recent draft) has a similar (but somewhat confusing) rule in **6.8.3(6)** that states the following:

```
Cv-qualifiers applied to an array type attach to the underlying
element type, so the notation “cv T”, where T is an array type,
refers to an array whose elements are so-qualified. An array type
whose elements are cv-qualified is also considered to have the same
cv-qualifications as its elements.
```

For the **_Atomic** qualifier introduced in **6.2.5(27)** nothing is changed, i.e. an array with an element type that has the **_Atomic** qualifier is not considered to have the same qualifier.

Explicitly creating **_Atomic** qualified arrays continues to be not allowed according to **6.7.3(4)** and **6.7.2.4(3)**.

Examples:

```
typedef int aT[3];
const aT x; // still allowed, 6.7.3(10)
typedef int aT[3];
_Atomic aT x; // still forbidden, 6.7.3(4)
```

Other Consequences of the Proposal

The proposal has two other minor effects, which both seem desirable, but which are not fully backwards compatible.

1.) Converting a pointer to a qualified array to a void* is not allowed anymore because the revised rules now detect that the qualifier is lost.

```
const int foo[5];
memset(&foo, 0, sizeof foo); // not allowed anymore
```

2.) The type of an expression involving the tertiary operator choosing between expressions of type **void*** and of type pointer-to-qualified-array will now change to include the qualifier which is otherwise lost. Compilers currently differ in whether the result has the qualifier or not.

```
void* v;
const int (*i)[3];
foo = (1 ? v : i); // <-- will have type 'const void*’
```

Explicit Construction of Qualified Array Types Using Typedefs

Reflector discussion brought up the concern that having **restrict**-qualified arrays might be confusing. The newly introduced qualification of the array type does not seem to be too problematic as it does not have any direct effect itself and can be safely ignored by programmers. Nevertheless, one could consider to completely forbid the explicit construction of qualified arrays using typedefs (which is now blessed by **6.7.3(10)** and otherwise syntactically not possible) or to allow this construction only for some qualifiers (e.g. the **const** qualifier). One the other hand, one could also go in the different direction and – for the sake of consistency - simply allow this construction for **_Atomic** too.

Examples:
(comments refer to the proposed rules)

```
typedef int *aT[3];
_Atomic aT a;  // not allowed (not allowed in C++)
const aT b;    // allowed (allowed in C++)
```

```
restrict aT c; // allowed (restrict does not exist in C++)
volatile At d; // allowed (allowed in C++)
```

Pointer Conversion Rules for restrict

Another question raised was whether it makes sense that a **restrict** qualifier can be added to the pointer type referenced itself by a pointer type This is allowed for **restrict** too as it follows all the generic rules of a qualifier. The present proposal will then also allow adding the **restrict** qualifier to arrays of pointers that are referenced by a pointer type, i.e. allow to add the qualifier to all pointers in an array at the same time. Both cases do not seem very important, but could be useful to mark pointers passed by reference as **restrict** qualified or to add the **restrict** qualifier to a large number of pointer variables without incurring the overhead of copying the complete array.

Examples:

```
void foo(int * restrict *ap);
int *a;
foo(&a); // pointer to restrict qualified pointer(s)

void foo(int * restrict (*a)[3]);
int *a[3];
foo(&a); // pointer to array(s) of three restrict qualified pointers
```

Suggested Wording Changes

6.2.5(26)
Any type so far mentioned is an unqualified type. Each unqualified type has several qualified versions of its type, corresponding to the combinations of one, two, or all three of the const, volatile, and restrict qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements. **An array and its element type are always considered to be identically qualified*) Any other A** derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

***) This does not apply to the _Atomic qualifier. Note that qualifiers do not have any direct effect on the array type itself, but affect conversion rules for pointer types that reference an array type.**

6.7.6.2(3) Array Declarators

If, in the declaration "T D1", D1 has one of the forms:

D [type-qualifier-list_opt assignment-expression_opt]
D [type-qualifier-list_opt assignment-expression]
D [type-qualifier-list static assignment-expression]
D [type-qualifier-list_opt]

and the type specified for ident in the declaration "T D" is "derived-declarator-type-list T", then the type specified for ident is "derived-declarator-type-list array of T".145,***)** (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword static.)

***) The array is considered identically qualified to T according 6.2.5(26).**

6.7.3(10)
If the specification of an array type includes any type qualifiers, **both the array and** the element type ~~**is**~~ **are** so-qualified~~**, not the array type**~~. If the specification of a function type includes any type qualifiers, the behavior is undefined.139)

139) ~~**Both of these**~~ **This** can occur through the use of typedefs. **Note that is rule does not apply to the _Atomic qualifier, and that qualifiers do not have any direct effect on the array type itself, but affect conversion rules for pointer types that reference an array type.**

6.7.3(2)
Types other than pointer types whose referenced type is an object type **and array types with such pointer types as element type** shall not be restrict-qualified.

References

1.) N 1923: Compatibility of Pointers to Arrays with Qualifiers
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1923.htm
2.) Reflector discussion from SC22WG14.17437 to SC22WG14.17460)

Acknowledgement

Joseph Myers for help with the implementation.