Doc Number: X3J16/93-0135
WG21/N0342
Date: September 27, 1993
Project: Programming Language C++
Ref Docs: 93-0062/N0269
Reply to: Samuel C. Kendall
Sun Microsystems Laboratories, Inc.
Sam.Kendall@East.Sun.COM

Type Combinations

Samuel C. Kendall

1. Introduction

Before we can specify what one can do with C++ types, we should agree on what C++ types exist. A well-understood example is that function parameters of array or function type are adjusted to pointer type, both in ISO C and in our WP (working paper), §8.2.5. So we don't have to specify the behavior of functions taking plain arrays as arguments -- there are no such function types. More recently, we decided in Munich to adjust function types with `const` or `volatile` parameters to eliminate those cv-qualifiers.

There are several similar open issues in C++. I attempt to enumerate them all and to suggest a solution in each case. The most prominent issue is cv-qualified return types; see section 3 below.

This paper attempts to define the status quo, not to suggest extensions. In fact, in several places where implementations differ or the WP is unclear I suggest disallowing the offending construct. Doing this has several advantages:

- We don't have to argue about what the construct means.
- Implementations that allow the construct can continue to support their current behavior (even if it differs from that of other implementations) as an extension.
- If in the future we want to un-disallow that construct, we can do so without breaking conforming code.

This paper considers the same issue discussed in my 92-0053/N0130 under the title *canonicalizing types*. I now prefer to call the issue *type combinations*.[1]

Table 1 shows the possible type combinations and what the status of each is. The open cases are indicated by boldface note numbers.

1. The difference is this: if we say that the type "function taking (`const int`) and returning `void`" is canonicalized to "function taking (`int`) and returning `void`", we are left with two questions. First, when does the canonicalization occur? And second, every time we refer to a type, do we mean the uncanonicalized or the canonicalized type? I prefer to suggest that there is no such type as "function taking (`const int`) and returning `void`"; any attempt to form such a type simply yields "function taking (`int`) and returning `void`" instead. To make this distinction is awkward in English; for example, the "type is adjusted" wording of the WP doesn't do it.

Table 1: Type Combination Rules

Type:	CV-Qualifier or Type Modifier: `const`	`volatile`	array `[N]`	array `[]`	function return or cast	function parameter	pointer to	reference to	pointer to member
`void`	✓	✓	✗	✗	✓	9	✓	✗	**7**
fundamental	✓	✓	✓	✓	✓	✓	✓	✓	✓
enum	✓	✓	✓	✓	✓	✓	✓	✓	✓
class	✓	✓	✓	✓	✓	✓	✓	✓	✓
`const`	**10**	✓	✓	✓	**3**	4	✓	✓	✓
`volatile`	✓	**10**	✓	✓	**3**	4	✓	✓	✓
`const volatile`	**10**	**10**	✓	✓	**3**	4	✓	✓	✓
array `[N]`	1	1	✓	✓	✗	8	✓	✓	✓
array `[]`	1	1	✓	✗	✗	8	✓	✓	✓
function	2	2	✗	✗	✗	5	✓	✓	✓
pointer	✓	✓	✓	✓	✓	✓	✓	✓	✓
reference	**10**	**11**	✗	✗	✓	✓	✗	✗	**6**
pointer to member	✓	✓	✓	✓	✓	✓	✓	✓	✓

Legend

✓ Well-formed; the resulting type is the row, qualified or modified by the column.

✗ Not well-formed.

Notes

1 *Proposed*: the cv-qualifier percolates down to the first non-array type modifier. See section 2.

2 Only allowed for nonstatic member functions. I propose minor editorial changes in section 8.

3 *Proposed*: ✓. See section 3. This is the important case.

4 Strip cv-qualifier from the function type; however, the declared parameter retains the cv-qualifier.

5 Adjusted to pointer to function.

6 *Proposed*: not well-formed. Pointers to members of reference type is (implicitly) allowed by the WP, but interpretations of the meaning differ between compilers. See section 4.

7 *Proposed*: not well-formed. “Pointer to member of class C of type (possibly cv-qualified) `void`” is implicitly allowed, but its meaning is not discussed anywhere in the WP. See section 5.

8 "Array of T" is adjusted to "Pointer to T".

9 A parameter of type `void` must be the only parameter and cannot be named. A paremeter of type cv-qualified `void` is not well-formed.

10 *Proposed*: the redundant cv-qualifier is ignored. See section 6.

11 *Proposed*: `volatile` references are not well-formed. See section 7.1

Discussion

This rules in this table are applied in all situations where types are formed by combination of types, cv-qualifiers, andtype modifiers. However, these are not what I'll call syntax rules; some syntax rules are more restrictive. For example,

```
const const int i;
```

is not well-formed due to a syntax rule, but

```
typedef const int CI;
const CI i;
```

is allowed by the syntax, and is governed by the proposal in note 10 above. There is at least one separate question of what is allowed in ayntax rule (see §8.2.2), but this question is beyond the scope of this paper.

The question of whether expressions can be of reference type, or whether reference-typed expressions "decay" to lvalues of non-reference type, is beyond the scope of this paper. In any case reference types *are* types; see §3.6.2.

This table is not quite rigorous in the area of cv-qualifiers. Here are several examples. Note 2 applies to cv-qualified function types, as well as to unqualified function types as the table suggests. The prohibition on reference to `void` actually applies to cv-qualified `void` as well. Pointer to function type is well-formed, but not pointer to cv-qualified function type; pointer to member of class C of type cv-qulified function *is* well-formed.

2. CV-Qualified Array Type

Steve Adamczyk explored this issue so well that I'll just reprint his letter here (with his permission):

Date: Sun, 1 Aug 93 18:01:25 EDT
Original-From: jsa@edg.com (J. Stephen Adamczyk)
Subject: Const array types

To: C++ core language mailing list
Message c++std-core-2539

Tom Plum's work on rewriting 5.2.4 has reminded me of an issue I wanted to bring up:

What does it mean to apply a type qualifier to an array type in C++?

```
typedef int A[5];
const A x; // What type does x have?
```

In ANSI/ISO C, the answer is that any attempt to add a type qualifier to an array type adds the qualifier to the element type instead of the array type ("If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type."). That means the type of "x" in the above is

array[5] of const int

and not

const array[5] of int

How should this work in C++? 7.1.6p1 of the ARM/WP tells us "Each element of a const array is const". Is this saying the same thing as the C rule, or does it mean each element is const and the array itself is also const, i.e., for the above

const array[5] of const int

?

One reason why we might want this to be different in C++ is shown by the following example:

```
typedef float Mat[3];
Mat b;
const Mat &a = b;
```

Is this valid? It is if this case falls under the rules for initializing a reference to const T. Is “const Mat” a const T? Using the C rules, it would not be, because being a “const T” in C means having a top type that is const-qualified, and

array[3] of const float

is not such a type.

And yet the above seems like a reasonable thing to want to do, especially if something like the above were to come up in a template.

If we want to make the above legal, we can do so in several ways:

(1) We can make the rule for adding qualified types to arrays different in C++ than in C. Consider again

```
typedef int A[5];
const A x;
```

The type of x could be

(a) const array[5] of const int

(b) const array[5] of int

(c) array[5] of const int (This is what C does.)

Either (a) or (b) would solve the reference initialization problem, but (a) works better because its type decays properly to a pointer to const int (of course, we could change the type decay rules too).

(2) We could say that in general in C++ anywhere a “const T” is required an array of const elements is also valid. This is in effect changing the definition of const-qualified to include the array case, i.e., const-qualified does not always mean “having a const on the top type”. This seems like such a bad idea we should go no further with it.

(3) We could change the rule for initialization of references to say that the rules for initializing a const T also apply to initializing an array with const elements. The change would have to be written in a way that would cover multi-dimensional arrays, i.e., array of array of ... of const element.

Of the three alternatives, I think (3) works best, combined with a better statement that const qualifiers on arrays work the same way as in C. (1)(a) is workable, but seems like a larger and more dangerous change.

Whichever way we change this, we need to broaden the rule in 7.1.6 to deal similarly with volatile.

[Incidentally, I’ve recommended to Tom that when his rewrite of 5.2.4 is done, the other half of the sentence in 7.1.6p1, the part that reads “..., and each nonfunction, nonstatic member of a const class object is const (9.3.1)”, should be removed because it will be taken care of -- as it should -- by 5.2.4]

Steve Adamczyk

I agree with most everything Steve says. However, I like his hated option (2). I think it can be defined by a sentence as simple as “a const type is a type with a top-level const cv-qualifier, or (recursively) an array of const types.” So I recommend his options (1)(c) and (2) together.

I think an acceptable alternative is his (1)(c) and (3) together.

3. CV-Qualified Function Return and Cast Type

The general question is, for a function f declared

```
cv T f(argtypes);
```

(where *cv* is `const` or `volatile` or both), is `f` of type `cv T f(argtypes)` or `T f(argtypes)` ? In other words, do we strip *cv* or not?

We can ask a similar question for casts to type `(cv T)`.[1]

I claim that the answer may be different depending on whether `T` is a class or a built-in type; but it should be the same for function return types and cast types, because they both yield non-lvalues.

Thanks to Bjarne Stroustrup, Kent Budge, Sean Corfield, Bill Gibbons, Andrew Koenig, Steve Adamczyk, and Max Skaller for their feedback on this section.

3.1 CV-Qualified Class Type

The WP -- other than §3.7/2, which was pulled in from the ISO C standard fairly recently, and which I argue in N0341 = 93-0134 is severely flawed -- makes no mention of stripping cv-qualifiers. Even §3.7/2 does not mention stripping cv-qualifiers from return types (only from lvalues when their value is used). The ARM does not mention stripping at all. Cfront, Microsoft C++, and (I suspect) most other implementations preserve cv-qualifiers on class return types. Changing this would break existing code such as the following:

```
struct A {
      void f();
      void f() const;
      // ...
};
const A get_const_A();
main() {
      get_const_A().f(); // calls f() const
      return 0;
}
```

For C compatibility we should strip cv-qualifiers, but cases where it matters are likely to be extremely rare. Here is an example:

```
struct A { ... };
struct A f();
const struct A f();
```

Thus C compatibility is a very weak reason to strip.

Some on the reflectors and in private email have indicated dismay at the ambiguity or inconsistency of the current type system or object model. Some suggest that the type system must be radically revised. Others disagree.

Table 2: Summary of 3.1

“Voter”	Strip cv-qualifiers from class return types?
ARM and WP	No
Cfront 3.0, Microsoft C++	No
C compatibility	Yes (very weak)
Existing code	No
Core reflector	Mixed

Conclusion: the status quo is that cv-qualifiers *are* preserved on class return types. Likewise they should be preserved in cast types.

1. We are really talking about explicit type conversions with both functional and cast notation. It’s just shorter to say “cast”.

3.2 CV-Qualified Built-in Type

What is the return type of a function declared `const int f();` ? The answer is either `const int` (*don't* strip cv-qualifiers) or `int` (*do* strip cv-qualifiers). We've just concluded this issue for *class* return types, but for *built-in* types the issue is more open: there is a (somewhat) stronger C compatibility argument for stripping cv-qualifiers; and there is a weaker existing code argument for keeping cv-qualifiers.

Let's look at several factors: the ARM; the WP; C compatibility; a couple of examples of the differences; and existing implementations.

The ARM and the WP "vote" exactly as in the section above.

For C compatibility it would be best to strip cv-qualifiers. This is because in C the following is legal:

```
const int f();
int f();
```

If we don't strip cv-qualifiers, this will not be legal C++.[1] However, I doubt many C programs declare two functions in this fashion. C compatibility is a weak reason to prefer stripping.

Let's look at several examples.

```
// Example 1
const int f();
const int& r = f(); // ok
```

For this simple example, it doesn't matter if we strip or not; the initialization is well-formed in either case. This is true of *all* initializations where no overloading is involved.

But here are two examples where you can tell the difference:

```
// Example 2
const T f();
void g(T&);
void g(const T&);
g(f());
```

Suppose `T` is a built-in type. If the type of `f()` is `const T`, we call `g(const T&)`. If, on the other hand, the type of `f()` is plain `T`, then overloading resolution selects `g(T&)`, which is an error because you cannot initialize a non-const reference with a non-lvalue.

As Andy Koenig has pointed out, a change in argument matching rules that may be proposed would cause `g(const T&)` to be selected in either case.

The second example is more obscure:

```
// Example 3
const volatile T f();
void g(const volatile T&);
void g(const T&);
g(f());
```

Again suppose `T` is a built-in type. If the type of `f()` is `const volatile T`, we call `g(const volatile T&)`. If, on the other hand, the type of `f()` is plain `T`, then depending on your reading of §13.2, either we call `g(const T&)` or the choice is ambiguous.

My conclusion from the examples: for consistency between built-in types and class types -- particularly in templates -- we should not strip.

But are these examples important? I think example 2 is somewhat important -- overloading on `T&` vs. `const T&` is something people do -- but example 3 is not.

1. Of course the same problem could occur for class types, but in C functions returning structures or unions are much rarer than functions returning built-in types.

I think the language slightly less difficult to teach if we treat built-in and class types as consistently as possible.; this argues that we should not strip.

FInally, I surveyed the core reflector. Most respondents favored not stripping, some strongly.

Table 3: Summary of 3.2

“Voter”	Strip cv-qualifiers from *built-in* function return types?
ARM and WP	No
Cfront 3.0, Microsoft C++	No
C compatibility	Yes (weak)
Examples	No (weak)
Consistency between class and built-in types	No
Survey	No (mostly)

My recommendation is therefore *don’t strip.* I don’t feel strongly about it.

Whatever we decide, cast types (types in explicit type conversions) should be stripped, or not, like function return types.

3.3 Conclusion

Don’t strip cv-qualifiers from function return types.

4. Pointer to Nonstatic Member of Reference Type

Pointers to nonstatic members of reference type is (implicitly) allowed by the WP, but interpretations of the meaning differ between cfront and Microsoft C++. Here is a brief example:

```
struct A { A(); int& r; };
&A::r; // well-formed? what is its type?
```

Bjarne Stroustrup believes it should be disallowed to allow possible optimizations. I think it should be disallowed to allow the differing implementations to be backward-compatible. A few like it. For more information see the following reflector messages: c++std-core-1217, 1219, 1223, 1227, 1228, 1230, 1232.

5. Pointer to Nonstatic Member of Void Type

“Pointer to member of class C of type (possibly cv-qualified) `void`” is implicitly allowed, but its meaning is not discussed anywhere in the WP. No implicit conversions from `T C::*` to `void C::*` are mentioned in the WP, although cfront implements them.

I suggest -- without having thought about it much -- that such types, and the accompanying implicit conversions, be disallowed. Pointers to data members are not that important.

6. Redundant CV-Qualifiers

Consider this example:

```
typedef const int CI;
const CI x;
```

The choices are that `x` is of type `const int`, or that `const`-qualifying `CI` is not well-formed. Bjarne Stroustrup favored the former resolution in message c++std-core-2341, because "otherwise writing templates will get unnecessarily complicated." After some discussion everyone agreed. So do I.

There was also agreement that

```
const const int y;
```

should not be well-formed.

Conclusion: redundant cv-qualifiers are ignored as long as they do not occur physically within the same declaration.

7. Volatile Reference Types

Consider the following:

```
struct A {
      A();
      ~A();
      int& r;
};
volatile A va;
```

Now, what is the type of the expression `va.r` ? Does it have any special semantics? This affects Tom Plum's §5.2.4 rewrite.

Also, is the following allowed:

```
int& volatile vr;
```

? Can it be constructed with typedefs? A similar question is posed in §8.2.2.

In order to avoid dealing with these unimportant but possibly tricky constructs, I suggest that we ban `volatile` references or `volatile` instances of classes containing references.

8. CV-Qualifiers on Function Types

This section proposes a small editorial change.

§8.2.5/3 defines when cv-qualifiers can go on function types. The last two sentences of that paragraph do not allow typedef'ing of cv-qualified function types; strangely, they are the only types with that restriction. Also, they imply that a cv-qualified function type cannot be created by cv-quallifying a typedef'd function type, eg, `const func_t`. Probably no one wants to do these things, but banning them is an unnecessary irregularity.

Here is a suggested replacement to eliminate these irregularities:

> *cv-qualifier-seq*, if present, indicates a cv-qualified function type. A cv-qualified function type can only be used as a typedef type, as the type of a nonstatic member function, or as part of a pointer to member function type; see §9.3.1.

A similar change should be made to §8.3/5.

9. Miscellaneous Editorial Proposal

§8.2.2 says

> The type `void&` is not permitted.

This sentence should be something like

> The types “reference to possibly-cv-qualified `void`” are not permitted.