Doc Number: X3J16/94-0071
WG21/N0458
Date: March 25, 1994
Project: Programming Language C++
Ref Docs: 94-0027/N0414, 93-0135/N0342
Reply to: Samuel C. Kendall
Sun Microsystems Laboratories, Inc.
Sam.Kendall@East.Sun.COM

Type Combinations, Revision 2

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), [dcl.fct]. So we don't have to specify the behavior of functions taking plain arrays as arguments —there are no such function types.

There is one remaining open issue (that I know of) in this area.

This document is an updated version of 93-0135/N0342. This document reflects the decisions made in San Jose and San Diego; closed issues have only brief descriptions in the table notes. I have not checked the new WP for compliance with all of the decisions.

This paper attempts to define the status quo. Sections 2 is the only open issue in this paper.

Table 1 shows the possible type combinations and what the rules for each is. There are no open cases left, just a related issue.

Table Legend

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

✕ Not well-formed.

Table Notes

1 The cv-qualifier percolates down to the first non-array type modifier. San Jose decision. But see section 2 for a related issue.

2 Only allowed for nonstatic member functions; see section 8.

3 CV-Qualifiers are kept on function return types (and in cast types). San Jose decision.

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

5 Adjusted to pointer to function. Both the parameter and its slot in the enclosing function type have type "pointer to T".

6 Pointers to nonstatic member of reference type is not well-formed. San Jose decision.

7 "Pointer to member of class C of type *cv* `void`" is not well-formed. San Jose decision.

8 Type "array of T" is adjusted to "pointer to T". Both the parameter and its slot in the function type have type "pointer to T".

Table 1: Type Combination Rules

Type	CV-Qualifier or Type Modifier								
	`const`	`vola-tile`	array `[`*N*`]`	array `[]`	func-tion return or cast	func-tion param-eter	pointer to	refer-ence 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	11	11	✕	✕	✓	✓	✕	✕	✕ 6
pointer to member	✓	✓	✓	✓	✓	✓	✓	✓	✓

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

10 The redundant cv-qualifier is ill-formed if in the same declaration (eg, `const const int`), ignored if introduced through the use of typedef or a template type parameter. San Jose decision.

11 CV-Qualified references are ill-formed if written directly; if introduced through the use of a typedef or template type parameter, the cv-qualifiers are ignored. San Diego decision.

Discussion

This table is not quite rigorous in the area of cv-qualifiers. Here are several examples. Table 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-qualified function *is* well-formed.

2. CV-Qualified Array Type

We decided in San Jose that cv-qualifiers percolate down to the first non-array type. What we formally decided in San Jose ended there.

But in order to have types like `const A`, where `A` is an array type, behave sensibly (went the original discussion in the core WG), a type such as “array [5] of const int” must somehow be considered a const type, even though its const qualifier is hidden below the array type modifier. We decided to do this by defining a “const type” as a type with a top-level const qualifier or (recursively) as an array of const types.

We did not discuss this issue in detail. Unfortunately it’s more complicated than I thought. I think the issue comes up rarely in practice, but unfortunately it becomes weird pretty quickly. I will explore some strange consequences of trying to make “array of const” types behave like top-level const types; then I will propose a solution.

Here are some examples:

```
typedef int A[5];
A x;
template <class T> void f(const T&);

const A& r = x;                                  // #1
f(x);                                            // #2
f(r);                                            // #3
```

Table 2: Types in the Example

Line	type being initialized	initializer
#1	reference to array[5] of const int	lvalue “array[5] of int”
#2	reference to const T	lvalue “array[5] of int”
#3	reference to const T	lvalue “array[5] of const int”

There are two issues, a conversion issue and a template type-matching issue.

We see the conversion issue with #1 and #2: are there conversions

“array of T”--> “array of const T”

This is a possible variants of the trivial conversion

“T” --> “const T”

(though I’m not sure that T --> const T is a trivial conversion anymore according to the new chapter 13).

This new conversion is not the only one needed. We also need the equivalent conversion for pointers to arrays; and we also need their generalizations to “arrays of arrays of ...”, and the cross-product of those generalizations with the recently generalized “pointer to pointer to ...” --> “pointer to const pointer to const ...” conversions. “Volatile” must also go in.

The template type-matching issue comes up with #2 and #3: does “const T” in a template formal parameter match “array of const” in an actual argument. Currently the answer is no.

I see three courses to take in resolving these issues:

- (1) Put in the conversions I mentioned (so that #1 and #2 are well-formed), and solve the template type matching issue (so that #3 is well-formed) somehow.
- (2) Come up with another way to formalize the interactions of const and array; for alternative approaches, see the letter from Steve Adamczyk in revision 1 of this paper.
- (3) Don’t put the conversions in (so that #1 and #2 are ill-formed), and ignore the template type matching issue (so that #3 is ill-formed).

I propose course (3); it is the simplest. This course restricts what you can do with arrays. But array types are already second-class in C and C++. Course (3) has the advantage of requiring no change to the WP, as far as I can tell.

3. CV-Qualified Function Return and Cast Type (decided)

4. Pointer to Nonstatic Member of Reference Type (decided)

5. Pointer to Nonstatic Member of Void Type (decided)

6. Redundant CV-Qualifiers (decided)

7. Const and Volatile Reference Types (decided)

8. CV-Qualifiers on Function Types

This section proposes no change.

[dcl.fct]/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; they are the only types with that restriction. Eg:

```
typedef void func1_t() const;        // ill-formed
struct A {
      typedef void func2_t() const; // ill-formed
};
```

Also, they imply that a cv-qualified function type cannot be created by cv-qualifying a typedef'd function type; eg:

```
typedef void func_t();
struct B {
      const func_t f;                // ill-formed
};
```

In revision 1 of this document I proposed allowing these examples in order to eliminate unnecessary warts in the type system. Objections from John Armstrong (core-2960) and Steven Adamczyk (convinced me otherwise.