**ANSI Doc No:** X3J16/92-0098
**ISO Doc No:** WG21/N0175
**Date:** September 16, 1992
**Project:** Programming Language C++
**Ref Doc:**
**Reply To:** Eric J. Krohn
krohn@bae.bellcore.com
+1 908 699-2708

# Qualified Class Names

## 1. Abstract

There is valid C and C++ code which provokes C++ to allow a qualified class name in an elaborated class specifier. This paper presents three different ways of expressing the qualified class names in elaborated class specifiers. The paper recommends allowing the syntax `struct A::B::C`. One open issue is whether to allow qualified class names in class declarations as well. The Appendices detail changes to the C++ grammar for each of the proposals.

## 2. Introduction

The syntax for *elaborated-class-specifier* allows only an identifier after the *class-key*. However, there exist circumstances where a qualified name must be used to specify the class. This paper examines some of these circumstances and proposes changes to the working paper.

## 3. The Problems

Consider the following valid C and C++ code.

```
struct A {
        struct B {
                struct C {
                        int i;
                        } C;
                } B;
        enum E {
                e = 1
                } E;
        };
```

Because of the hiding of class names and enumeration names in a scope by object names in that same scope (sec. 3.2p2), mention of `A::B` refers to the `B` object in `struct A`. There is no way to refer to the type `struct B` from outside the enclosing class. The same is true of the type `enum E`.

Some compilers currently recognize `struct A::B` as the inner struct, even though the syntax is not valid according to the working paper.

## 4. Proposals

1. **Do nothing.**
   Pros: Leave the syntax unchanged. Less work for the committee.
   Cons: This leaves a valid C structure declaration with no way to name the type of the inner struct.

2. **Allow struct A::B::C.**
   Pros: This change would allow references to the inner struct's type. This change allows a "natural" order: we have the class key followed by a name, where the name may be qualified.
   Cons: Some people argue that since the `struct` at the far left applies to the `C` on the far right it is "too far away". One can imagine a deeply nested struct where this becomes painfully obvious:
   ```
   struct A::B::C::D D_object;
   ```
   Yet this statement makes clear that some sort of struct object is being declared.

3. **Allow struct A::struct B::struct C.**
   Pros: This change would allow references to the inner struct's type. Some people argue that the `struct` is close to the name, `C`, that it modifies.
   Cons: This syntax is ugly, though one person argued that C compatibility "hacks" ought to be ugly. This change now introduces "optional" class keys in new places: One might refer to the inner `struct C` above as any of

```
A::B::struct C
A::struct B::struct C
struct A::struct B::struct C
struct A::B::struct C
```

   This follows from the rule that an identifier followed immediately by `::` is recognized as a class name, even when hidden by an object of the same name.

4. **Allow only A::B::struct C.**
   Pros: This proposal differs from the previous by allowing the *class-key* or `enum` only before the rightmost name. As such this proposal is somewhat simpler than the previous one and it allows only one naming variant for `A::B::struct C` above.
   Cons: This syntax is still ugly.

5. **Variants**

In addition to the above possible changes to *elaborated-type-specifiers*, we also have the possibility of incorporating related changes.

It is not clear whether a qualified name should be permitted in a class declaration, such as

```
struct A {
      struct B;
      struct B *B;
            // Choose at most one of the following:
      struct A::B             { int i; }; // Proposal 2
      struct A::struct B      { int i; }; // Proposal 3
      A::struct B             { int i; }; // Proposal 4
      };
```

If so, then we need grammar rules to allow one of these. If not, then we introduce an inconsistency in what may follow a *class-key*.

Proposal 3 (in the previous section) shows examples where `struct A::` is synonymous with `A::` in elaborated class specifiers and elaborated enum specifiers, but not elsewhere. We could write

```
struct A::struct B::struct C  a_C_object;
```

but not

```
B_ptr->struct B::C = some_value;
```

We can either leave the incongruity or make `struct A::` synonymous with `A::` in every place.

A. **Allow only qualified class and enum elaborations.**
   Pros: This may be what we want semantically, so we may choose to have the syntax follow the semantics.
   Cons: This introduces another exception to the rules.

B. **Allow qualified class and enum declarations also.**
   Pros: Provides uniformity between class elaborations and class declarations. Likewise for enums.
   Cons: No demonstrated need (yet).

C. **Allow** *class-key class-name* `::` as a synonym for *class-name* `::`.
   This variant only applies to Proposal 3 above.
   Pros: Uniformity: if we allow it one place, why not allow it everywhere?
   Cons: This variant provides yet another way of naming things. It allows the following code.

```
struct R {
        // ...
        };
struct A {
        struct R mf ();
        };

struct R struct A::mf ()
{
        // ...
}
```

Note too that

```
struct A::B
```

refers to the B object within A under this variant.

## 6. Recommendations

Proposal 1 (No Change) leaves C++ with Yet Another C Incompatibility.
Not recommended.

Proposal 2 (struct A::B::C) + Variant A fixes the C incompatibility. It already has been implemented by compilers. Its syntax is somewhat pleasing. Appendix I shows the grammar changes needed for this proposal.
Recommended.

Proposal 2 (struct A::B::C) + Variant B fixes the C incompatibility. Its syntax is somewhat pleasing. Appendix I shows the grammar changes needed for this proposal.
Recommended.

Proposal 3 (struct A::struct B::struct C) + Variant A fixes the C incompatibility. It permits too many ways of naming nested types. It is visually ugly. Appendix II shows the grammar changes needed for this proposal.
Not recommended.

Proposal 3 (struct A::struct B::struct C) + Variant B fixes the C incompatibility. It permits too many ways of naming nested types. It is visually ugly. Appendix II shows the grammar changes needed for this proposal.
Not recommended.

Proposal 3 (struct A::struct B::struct C) + Variant C fixes the C incompatibility. It permits too many ways of naming nested types. It makes `struct A::` completely synonymous with `A::`. It is visually ugly. Appendix II shows the grammar changes needed for this proposal.
Not recommended.

Proposal 4 (A::B::struct C) + Variant A fixes the C incompatibility. It is visually ugly. Appendix III shows the grammar changes needed for this proposal.
Not recommended.

Proposal 4 (A::B::struct C) + Variant B fixes the C incompatibility. It is visually ugly. Appendix III shows the grammar changes needed for this proposal.
Not recommended.

Proposal 2/A or 2/B cause the fewest conflicts and offer a reasonable naming scheme. Proposal 2/B introduces no new conflicts in the grammar but it would require either prose to disallow qualified names in class and enum declarations or prose to ascribe meaning to these declarations.

## 7. Notes on Appendices

Though not shown in the Appendices in all cases, this analysis had the *class-head* rule folded into the *class-specifier* rule to lessen the number of conflicts.

### Appendix I — Grammar Changes for Proposal 2 (struct A::B::C) + Variant A

Proposal 2 (struct A::B) requires the following grammar changes.

*elaborated-type-specifier:*
*class-key identifier*
*class-key class-name*
enum *enum-name*

becomes

*elaborated-type-specifier:*
*class-key nested-class-specifier*
enum *nested-enum-specifier*

*nested-enum-specifier:*
*enum-name*
*class-name* : : *nested-enum-specifier*

Note that the distinction in the original rules between *identifier* and *class-name* is lost in the latter rules. These rules add one shift/reduce conflict because qualified names are not allowed in class declaration. In effect, we disallow a parse which has no valid semantics. By Pennello's classification in X3J16/91-0007, this is a category 3 situation (unwanted constraint-wise b).

### Grammar Changes for Proposal 2 (struct A::B::C) + Variant B

To allow qualified names in class and enum declarations, the rules

*class-head:*
*class-key identifier*$_{opt}$ *base-clause*$_{opt}$
*class-key class-name base-clause*$_{opt}$
*enum-specifier:*
enum *identifier*$_{opt}$ { *enumerator-list*$_{opt}$ }

should be replaced by

*class-head:*
*class-key nested-class-specifier*$_{opt}$ *base-clause*$_{opt}$
*enum-specifier:*
enum *nested-enum-specifier*$_{opt}$ { *enumerator-list*$_{opt}$ }

These rules add no new shift/reduce or reduce/reduce conflicts. In this case, we either allow a parse with no valid semantics or we must ascribe semantics to the parse (Variant B).

## Appendix II — Grammar Changes for Proposal 3 (struct A::struct B::struct C) + Variant A

Proposal 3 (struct A::struct B::struct C) requires the following grammar changes.

*elaborated-type-specifier:*
*class-key identifier*
*class-key class-name*
enum *enum-name*

becomes

*elaborated-type-specifier:*
*elaborated-class-specifier*
*elaborated-enum-specifier*

*elaborated-class-specifier:*
*class-key class-name*
*class-key identifier*
*class-key*$_{opt}$ *class-name* : : *elaborated-class-specifier*
*class-key identifier* : : *elaborated-class-specifier*

*elaborated-enum-specifier:*
enum *enum-name*
enum *identifier*
*class-key*$_{opt}$ *class-name* : : *elaborated-enum-specifier*
*class-key identifier* : : *elaborated-enum-specifier*

These changes add 8 shift/reduce conflicts due to the ambiguity between *class-key class-name* and *class-key class-name*: :. Choosing the shift over the reduce discards undesired parses (Pennello's category 3: unwanted constraint-wise b).

## Grammar Changes for Proposal 3 (struct A::struct B::struct C) + Variant B

To allow qualified names in class and enum declarations, the rules

*elaborated-type-specifier:*
*class-key identifier*
*class-key class-name*
enum *enum-name*
*class-specifier:*
*class-head* { *member-specification*$_{opt}$ }
*class-head:*
*class-key identifier*$_{opt}$ *base-clause*$_{opt}$
*class-key class-name base-clause*$_{opt}$

should be replaced by

*elaborated-type-specifier:*
*elaborated-class-specifier*
*elaborated-enum-specifier*

*elaborated-class-specifier:*
*class-key class-name*
*class-key identifier*
*class-key*$_{opt}$ *class-name* : : *elaborated-class-specifier*
*class-key identifier* : : *elaborated-class-specifier*

*class-specifier:*
*elaborated-class-specifier base-clause*$_{opt}$ { *member-specification*$_{opt}$ }
*class-key base-clause*$_{opt}$ { *member-specification*$_{opt}$ }

*elaborated-enum-specifier:*
enum *enum-name*
enum *identifier*
*class-key*$_{opt}$ *class-name* : : *elaborated-enum-specifier*
*class-key identifier* : : *elaborated-enum-specifier*

*enum-specifier:*
*elaborated-enum-specifier* { *enumerator-list*$_{opt}$ }
enum { *enumerator-list*$_{opt}$ }

These changes add 4 shift/reduce conflicts due to the ambiguity between *class-key class-name* and *class-key class-name* : :. Choosing the shift over the reduce discards undesired parses (Pennello's category 3: unwanted constraint-wise b).

**Grammar Changes for Proposal 3 (struct A::struct B::struct C) + Variant C**

To allow *class-key class-name* : : as a synonym for *class-name* : :, use the rules above plus

*qualified-id:*
*nested-class-specifier* : : *id-expression*
*qualified-type-specifier:*
*typedef-name*
*class-name* : : *qualified-type-specifier*
*nested-class-specifier:*
*class-name*
*class-name* : : *nested-class-specifier*

should be replaced by

*qualified-id:*
*class-name* : : *id-expression*
*class-key class-name* : : *id-expression*
*qualified-type-specifier:*
*typedef-name*
*class-name* : : *qualified-type-specifier*
*class-key class-name* : : *qualified-type-specifier*
*nested-class-specifier:*
*class-name*
*class-key class-name*
*class-name* : : *nested-class-specifier*
*class-key class-name* : : *nested-class-specifier*

These changes add 12 shift/reduce conflicts and 112 reduce/reduce conflicts. The bulk of the reduce/reduce conflicts arise from an ambiguity between *elaborated-class-specifier* and *nested-class-specifier*. Choosing to reduce by either rule discards valid sentences (Pennello's category 4: lost sentences).

A better approach might be to use the technique of Jim Roskind's YACCable C++ grammar (release 5), where the scope prefixes are gathered into a small set of rules which are then used everywhere that scopes are needed. This technique effectively limits the conflicts but it alters the existing grammar extensively and is too lengthy to present here.

## Appendix III — Grammar Changes for Proposal 4 (A::B::struct C) + Variant A

Proposal 4 (A::B::struct C) with Variant A requires the following grammar changes.

*elaborated-type-specifier:*
*class-key identifier*
*class-key class-name*
enum *enum-name*

becomes

*elaborated-type-specifier:*
*elaborated-class-specifier*
*elaborated-enum-specifier*

*elaborated-class-specifier:*
*class-key class-name*
*class-key identifier*
*class-name* : : *elaborated-class-specifier*

*elaborated-enum-specifier:*
enum *enum-name*
enum *identifier*
*class-name* : : *elaborated-enum-specifier*

These rules add 4 shift/reduce conflicts. Choosing the shift over the reduce discards undesired parses (Pennello's category 3: unwanted constraint-wise b).

## Grammar Changes for Proposal 4 (A::B::struct C) + Variant B

Proposal 4 (A::B::struct C) with Variant B requires the following grammar changes.

*elaborated-type-specifier:*
*class-key identifier*
*class-key class-name*
enum *enum-name*

becomes

*elaborated-type-specifier:*
*elaborated-class-specifier*
*elaborated-enum-specifier*

*elaborated-class-specifier:*
*class-key class-name*
*class-key identifier*
*class-name* : : *elaborated-class-specifier*
*class-specifier:*
*elaborated-class-specifier base-clause*$_{opt}$ { *member-specification*$_{opt}$ }
*class-key base-clause*$_{opt}$ { *member-specification*$_{opt}$ }

*elaborated-enum-specifier:*
enum *enum-name*
enum *identifier*
*class-name* : : *elaborated-enum-specifier*

*enum-specifier:*
*elaborated-enum-specifier* { *enumerator-list*$_{opt}$ }
enum { *enumerator-list*$_{opt}$ }

These rules add 2 shift/reduce conflicts. Choosing the shift over the reduce discards undesired parses (Pennello's category 3: unwanted constraint-wise b).