The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.2.3 Tags

type contents defined once

A specific type shall have its content defined at most once. 1454

Commentary

declaration only one if no linkage

content 1462 list defines

The general requirement that an identifier with no linkage not be declared more than once does not apply to tags. An identifier denoting the same tag can be declared zero or more times if no content is defined. Among these declarations can be one that defines the content. What constitutes *content* is specified elsewhere.

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard does not classify the identifiers that occur after the **enum**, **struct**, or **union** keywords as tags. There is no tag namespace. The identifiers exist in the same namespace as object and typedef identifiers. This namespace does not support multiple definitions of the same name in the same scope (3.3p4). It is this C++ requirement that enforces the C one given above.

tag name same struct, union or enum

Where two declarations that use the same tag declare the same type, they shall both use the same choice of **struct**, **union**, or **enum**. 1455

Commentary

This sentence was added by the response to DR #251. The following code violates this constraint:

```
1  struct T {
2             int mem;
3            };
4  union T x; /* Constraint violation. */
```

C90

The C90 Standard did not explicitly specify this constraint. While the behavior was therefore undefined, it is unlikely that the behavior of any existing code will change when processed by a C99 translator (and no difference is flagged here).

C++

7.1.5.3p3 *The* `class-key` *or* **enum** *keyword present in the* `elaborated-type-specifier` *shall agree in kind with the declaration to which the name in the* `elaborated-type-specifier` *refers.*

Common Implementations

Early translators allowed the **struct** and **union** keywords to be intermixed (i.e., the above example was considered to be valid).

A type specifier of the form 1456

```
enum identifier
```

without an enumerator list shall only appear after the type it specifies is complete.

Commentary

Incomplete types are needed to support the declaration of mutually recursive structure and union types. It is not possible to create a mutually recursive enumerated type and a declaration making use of self-referencing recursion is an edge case that does not appear to be of practical use.

C90

This C99 requirement was not specified in C90, which did not containing any wording that ruled out the declaration of an incomplete enumerated type (and confirmed by the response to DR #118). Adding this constraint brings the behavior of enumeration types in line with that for structure and union types. sizeof constraints
Source code containing declarations of incomplete enumerator types will cause C99 translators to issue a diagnostic, where a C90 translator was not required to issue one.

```
enum E1 { ec_1 = sizeof (enum E1) };   /* Constraint violation in C99. */
enum E2 { ec_2 = sizeof (enum E2 *) }; /* Constraint violation in C99. */
```

C++

[Note: if the elaborated-type-specifier *designates an enumeration, the* identifier *must refer to an already declared* enum-name*.* 3.3.1p5

If the elaborated-type-specifier *refers to an* enum-name *and this lookup does not find a previously declared* enum-name*, the* elaborated-type-specifier *is ill-formed.* 3.4.4p2

```
enum incomplete_tag *x; /* constraint violation */
                        // undefined behavior

enum also_incomplete; /* constraint violation */
                      // ill-formed
```

Semantics

1457 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. tag declarations same scope

Commentary

What constitutes a declaration of structure, union, or enumerated type? The answer depends on whether a prior declaration, using the same identifier as a tag, is visible. If such an identifier is visible at the point of another declaration, no new type is declared (and there may also be a constraint violation). 1472 struct-or-union identifier visible 1454 type contents defined once

```
struct T_1 {   /* A declaration (1) of tag T_1. */
             int mem_1;
            };

void f(void)
{
struct T_1 *p; /* Prior declaration (1) is visible, not a declaration of T_1. */
struct T_1 {   /* A declaration (2) of tag T_1. */
             float mem_2;
            };
struct T_1 *q; /* Prior declaration (2) is visible and used. */
struct U_1 *r; /* No prior declaration of U_1 visible, a declaration of U_1. */
struct U_1 {   /* Defines the content of prior declaration of U_1. */
             void *mem_3;
```

```
15              };
16
17    sizeof(p->mem_1);
18    sizeof(q->mem_2);
19    sizeof(r->mem_3);
20    }
```

struct-or-union identifier visible 1472

The wording that covers tags denoting the same type, but declared in different scopes occurs elsewhere.

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard specifies this behavior for class types (9.1p2). While this behavior is not specified for enumerated types, it is not possible to have multiple declarations of such types.

Coding Guidelines

identifier declared in one file ??

If the guideline recommendation specifying a single point of declaration is followed, the only situation where a tag, denoting the same type, is declared more than once is when its type refers to another type in some mutually recursive way.

---

tag incomplete until

The type is incomplete[109)] until the closing brace of the list defining the content, and complete thereafter. 1458

Commentary

The closing brace that defines its content may occur in a separate declaration. Incomplete types are one of the three kinds of types defined in C. The only other incomplete type is **void**, which can never be completed.

incomplete types void is incomplete type

C++

The C++ Standard specifies this behavior for class types (9.2p2), but is silent on the topic for enumerated types.

---

tag declarations different scope

Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. 1459

Commentary

tag declarations same scope 1457

The discussion on what constitutes a declaration is also applicable here.

```
1   struct T_1 {           /* A declaration (1) of tag T_1. */
2               int mem_1;
3              };
4
5   void f(void)
6   {
7   struct T_1 *p; /* Prior declaration (1) is visible, not a declaration of T_1. */
8   struct T_1;    /* This is always a declaration, it is a different type from (1). */
9   struct T_1 *q; /* q points at a different type than p. */
10  }
11
12  void g(struct T_1 *); /* Prior declaration visible, not a declaration of T_1. */
13  void h(struct U_1 *); /* No prior declaration visible, a declaration of U_1.  */
```

C90

tag declarations same scope 1457

The C99 Standard more clearly specifies the intended behavior, which had to be inferred in the C90 Standard.

C++

The C++ Standard specifies this behavior for class definitions (9.1p1), but does not explicitly specify this behavior for declarations in different scope.

Coding Guidelines

If the guideline recommendation dealing with the reuse of identifier names is followed there will never be two distinct types with the same name. The case of distinct tags being declared with function prototype scope does not need a guideline recommendation. Such a declaration will render the function uncallable, as no type can be declared to be compatible with its parameter type. A translator will issue a diagnostic if a call to it occurs in the source.

?? identifier reusing names

1460 Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

struct/union declaration no tag

Commentary

A declaration of a structure or union type that includes a tag may declare a distinct type, or it may refer to a previously declared distinct type.

If one of the identifiers declared is a typedef name, it will be possible to refer to the type in other contexts. If the identifier being declared is an object there is no standard defined way of referring to its type. Such types are sometimes known as *anonymous* types.

1468 footnote 110

Two types have compatible type if they are the same. Types that are distinct are not the same.

compatible type if

C90

The C90 Standard refers to a " . . . a new structure, union, or enumerated type," without specifying the distinctness of new types. The C99 Standard clarified the meaning.

C++

The C++ Standard specifies that a class definition introduces a new type, 9.1p1 (which by implication is distinct). However, it does not explicitly specify the status of the type that is created when the tag (a C term) is omitted in a declaration.

Other Languages

A small number of languages (e.g., CHILL) use structural equivalence for their type compatibility rules, rather than name equivalence. In such cases it is possible for many different type declarations to be treated as being compatible.

structural equivalence

Common Implementations

Some implementations (`gcc`) include the **`typeof`** operator. This returns the type of its operand. With the availability of such an operator no types can be said to be anonymous.

1461 A type specifier of the form

struct-or-union *identifier*$_{opt}$ **{** *struct-declaration-list* **}**

or

enum *identifier* **{** *enumerator-list* **}**

or

enum *identifier* **{** *enumerator-list* **,** **}**

declares a structure, union, or enumerated type.

Commentary

This specification provides semantics for a subset of the possible token sequences supported by the syntax of *type-specifier*. The difference between this brace delimited form and the semicolon terminated form is similar to the difference between the brace delimited and semicolon terminated form of function declarations (i.e., one specifies content and the other doesn't).

type specifier syntax
1464 struct tag;

C90

Support for the comma terminated form of enumerated type declaration is new in C99.

C++

The C++ Standard does not explicitly specify this semantics (although 9p4 comes close).

---

content list defines

The list defines the *structure content*, *union content*, or *enumeration content*. 1462

Commentary

type 1454 contents defined once identifier scope determined by declaration placement name space

This defines the terms *structure content*, *union content*, or *enumeration content*, which is the *content* referred to by the constraint requirement. The content is the members of the type declared, plus any type declarations contained within the declaration. Any identifiers declared by the list has the same scope as that of the tag, that might be defined, and may be in several name spaces.

Other Languages

Object-oriented languages allow additional members to be added to a class (structure) through the mechanism of inheritance.

---

tag declare

If an identifier is provided,[110)] the type specifier also declares the identifier to be the tag of that type. 1463

Commentary

This provides the semantic association for the identifier that appears at this point in the syntax.

C++

The term *tag* is not used in C++, which calls the equivalent construct a *class name*.

Table 1463.1: Occurrence of types declared with tag names (as a percentage of all occurrences of each keyword). Based on the visible form of the `.c` and `.h` files.

	`.c` files	`.h` files
`union` identifier	65.5	75.8
`struct` identifier	99.0	88.4
`enum` identifier	86.6	53.6

---

struct tag;

A declaration of the form 1464

```
struct-or-union identifier ;
```

specifies a structure or union type and declares the identifier as a tag of that type.[111]

Commentary

This form of declaration either declares, or redeclares, the identifier, as a tag, in the current scope. The following are some of the uses for this form of declaration:

EXAMPLE 1474 mutually referential structures

- To support mutually referring declarations when there is the possibility that a declaration of one of the tags is already visible.
- To provide a mechanism for information hiding. Developers can declare a tag in an interface without specifying the details of a types implementation,
- In automatically generated code, where the generator does not yet have sufficient information to fully define the content of the type, but still needs to refer to it.

1465 109) An incomplete type may only by used when the size of an object of that type is not needed.

footnote 109
size needed

Commentary

When is the size of an object not needed? Who, or what needs the size and when do they need it?

The implementation needs the size of objects to allocate storage for them. When does storage need to be allocated for an object? In theory, not until the object is encountered during program execution (and in practice for a few languages). However, delaying storage allocation until program execution incurs a high-performance penalty. Knowing the size during translation enables much more efficient machine code to be generated. Also, knowing the size when the type is first encountered (if the size has to be known by the implementation) can simplify the job of writing a translator (many existing translators operated in a single pass).

object
reserve storage

implementation
single pass

The size of an object having an incomplete array type is not needed to access an element of that array.

incomplete array
indexing

The Committee responses to defect reports (e.g., DR #017) asking where the size of an object is needed do not provide a list of places. Now the wording has been moved to a footnote, perhaps this discussion will subside.

C90

It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.

The above sentence appears in the main body of the standard, not a footnote.

The C99 wording is more general in that it includes all incomplete types. This is not a difference in behavior because these types are already allowed to occur in the same context as an incomplete structure/union type.

incomplete types

C++

The C++ Standard contains no such rule, but enumerates the cases:

[Note: the rules for declarations and expressions describe in which contexts incomplete types are prohibited.] 3.9p8

Other Languages

Knowing the size of objects is an issue in all computer languages. When the size needs to be known is sometimes decided by high-level issues of language design (some languages require their translators to effectively perform more than one pass over the source code), other times it is decided by implementation techniques.

Common Implementations

Most C translators perform a single pass over the source code, from the point of view of syntactic and semantic processing. An optimizer may perform multiple passes over the internal representation of statements in a function, deciding how best to generate machine code for them.

implementation
single pass

1466 It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.)

size not needed
examples

Commentary

A typedef name is simply a synonym for the type declared. All pointers to structure types and all pointers to union types have the same alignment requirements. No information on their content is required. The size may not be needed when a function returning a structure or union is declared, but it is needed when such a function is defined.

typedef
is synonym

alignment
pointer to structures

alignment
pointer to unions

Other Languages

Many languages support some form of pointer to generic type, where little information (including size) is known about the pointed-to type. Support for type declarations where the size is unknown, in other contexts, varies between languages. In Java the number of elements in an array type is specified during program execution.

Common Implementations

This is one area where vendors are often silent on how their language extensions operate. For instance, the gcc **typeof** operator returns the type of its operand. However, the associated documentation says nothing about the case of the operand type being incomplete and having a tag that is identical to another definition occurring within the scope that the **typeof** occurred. One interpretation (unsupported by any specification from the vendor) of the following:

```
1  extern struct fred f;
2
3  int main (void)
4  {
5  typeof (f) *x;
6  struct fred { int x; } s;
7  typeof (f) *y;
8  y=&s;                  /* Types not compatible? */
9  }
10
11 struct fred {
12              int mem;
13             };
```

is that both x and y are being declared as being pointers to the type of f, that is an incomplete type, and that the declaration of the tag fred, in a nested scope, has no effect on the declaration of y.

Example

```
1  typedef struct foo T;
2
3  struct foo *ptr;
4  struct foo f(void);
5
6  void g(void *p)
7  {
8  (struct foo *)p;
9  }
```

The specification has to be complete before such a function is called or defined. 1467

Commentary

In these contexts the commonly used methods for mapping source code to machine code need to know the number of bytes in a types object representation.

C90

The specification shall be complete before such a function is called or defined.

The form of wording has been changed from appearing to be a requirement (which would not be normative in a footnote) to being commentary.

1468 110) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part.

footnote 110

Commentary

The requirements for referring to objects declared using such types are discussed elsewhere. Such types are distinct and are said to be *anonymous*. They cannot be referred to elsewhere in the translation unit. Although their associated objects can be accessed. In:

compatible separate translation units
1460 struct/union declaration no tag

```
struct {
         int m1;
        } x, y;
struct {
         int m1;
        } z;
```

x and y are compatible with each other. They both have the same anonymous type, but the object z has a different anonymous type. Note that the types of the objects x, y, and z would be considered to be compatible if they occurred in different translation units.

compatible separate translation units

C90

This observation was is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

Other Languages

Some languages include a **`typeof`** operator, which returns the type of its operand.

Common Implementations

Some implementations include a **`typeof`** operator. This returns the type of its operand. The availability of such an operator means that no types never need be truely anonymous.

1469 Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

Commentary

Use of a typedef name does not alter the requirements on the type not being an incomplete type in some contexts.

1465 size needed

```
struct S;
typedef struct S t_1;
extern t_1 f(void);

struct S {
           int mem;
         };
typedef struct S t_2;
t_2 glob;

t_1 f(void)
{
return glob;
}
```

C90

This observation is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

Example

In the following both a and b have the same type. The typedef name T_S provides a method of referring to the anonymous structure type.

```
typedef struct {
                int m2;
                } T_S;
T_S a;
T_S b;
```

---

footnote 111

111) A similar construction with **enum** does not exist. 1470

Commentary

The need for mutual recursion between different enumerated types is almost unheard of. One possible use of such a construct might be to support the hiding of enumeration values. For instance, an object of such an enumeration type might be passed as a parameter which only ever appeared as an argument to function calls. However, C99 considers the following usage to contain a number of constraint violations.

```
enum SOME_STATUS_SET; /* First constraint violation. */

extern enum SOME_STATUS_SET get_status(void);
extern void use_status(enum SOME_STATUS_SET *);
```

C++

7.1.5.3p1 *If an elaborated-type-specifier is the sole constituent of a declaration, the declaration is ill-formed unless . . .*

The C++ Standard does not list `enum identifier ;` among the list of exceptions and a conforming C++ translator is required to issue a diagnostic for any instances of this usage.

struct-or-union identifier not visible 1471

The C++ Standard agrees with this footnote for its second reference in the C90 Standard.

---

struct-or-union identifier not visible

If a type specifier of the form 1471

struct-or-union identifier

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[111]

Commentary

size not needed examples 1466

The forms of *struct-or-union identifier*, excluded by this wording, are the identifier being followed by a semicolon or a left brace. The remaining possible occurrences of this form are described elsewhere and include:

```
struct secret_info *point_but_not_look;
```

C++

The C++ Standard does not explicitly discuss this kind of construction/occurrence, although 3.9p6 and 3.9p7 discuss this form of incomplete type.

Coding Guidelines

When no other declaration is visible at the point this type specifier occurs, should this usage be permitted? Perhaps it was intended that a tag be visible at the point in the source where this type specifier occurs. However, not having a prior declaration visible is either harmless (intended or otherwise), or will cause a diagnostic to be issued by a translator.

A pointer to an incomplete structure or union type is a more strongly typed form of generic pointer than a pointer to **void**. Whether this use of pointer to incomplete types, for information hiding purposes, is worthwhile can only be decided by the developer.

1472 If a type specifier of the form

struct-or-union identifier visible

struct-or-union identifier

or

enum *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

Commentary

The forms of *struct-or-union identifier*, excluded by this wording, are the identifier being followed by a semicolon or a left brace. This is the one form that is not a declaration.

Coding Guidelines

Technically the only reason for using tags is to define mutually recursive structure or union types. However, in practice this is the most common form used to declare objects having structure, union, or enumerated types. In general these coding guidelines recommend that developers continue to following common existing practices. Given that most existing code contains declarations that use this form of type specifier, is there a worthwhile benefit in recommending use of an alternative form? The following are some of the issues involved in the obvious alternative form, the use of typedef names:

1454 type contents defined once

culture of C

- While the use of a typedef name may appear to reduce future maintenance costs (e.g., if the underlying type changes from a structure to an array type, a single edit to the definition of a typedef name is sufficient to change to any associated object declarations). In practice the bulk of the costs associated with such a change are created by the need to modify the operators used to access the object (i.e., from a member selection operator to a subscript operator). Also experience suggests that this kind of change in type is not common.
- Changes in an objects structure type may occur as a program evolves. For instance, the object `x` may have structure type `t_1` because it needs to represent information denoted by a few of the members of that type. At a later time the type `t_1` may be subdivided into several structure types, with the members referenced by `x` being declared in the type `t_1_3`. Developers then have the choice of changing the declaration of `x` to be `t_1_3`, or leaving it alone. However, the prior use of a typedef name, rather than a tag, is unlikely to result in any cost savings, when changing the declaration of `x` (i.e., developers are likely to have declared `x` to have type `t_1`, rather than a synonym of that type, so the declaration of `x` will either have to be edited).
- What are the cognitive costs and benefits associated with the presence, or absence of a keyword in the source of a declaration? There is a cost to readers in having to process an extra token (i.e., the keyword) in the visible source, or any benefits, to readers of the visible source. However, the visual presence of this keyword may reduce the cognitive effort needed to deduce the kind of declaration being made. There does not appear to be a significant cost/benefit difference between any of these cognitive issues.

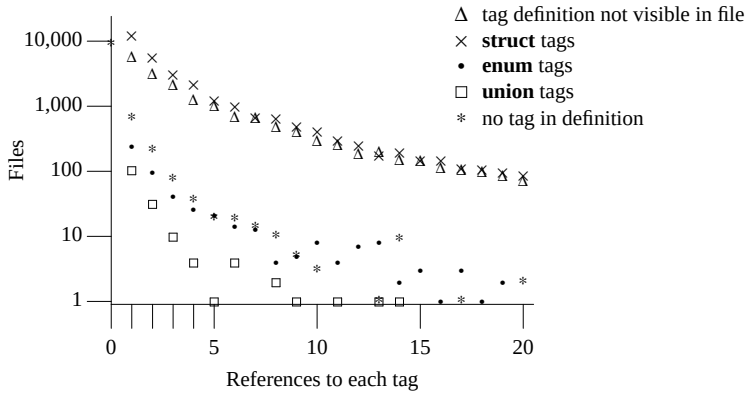

Figure 1472.1: Number of files containing a given number of references to each tag previously defined in the visible source of that file (times, bullet, square; the definition itself is not included in the count), tags with no definition visible in the `.c` file (triangle; i.e., it is defined in a header) and anonymous structure/union/enumeration definitions (star). Based on the visible form of the `.c` files.

EXAMPLE 1 This mechanism allows declaration of a self-referential structure. 1473

```
struct tnode {
        int count;
        struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be an object of the given type and **sp** to be a pointer to an object of the given type. With these declarations, the expression **sp->left** refers to the left **struct tnode** pointer of the object to which **sp** points; the expression **s.right->count** designates the **count** member of the right **struct tnode** pointed to from **s**. The following alternative formulation uses the **typedef** mechanism:

```
typedef struct tnode TNODE;
struct tnode {
        int count;
        TNODE *left, *right;
};
TNODE s, *sp;
```

Commentary

Both of these formulations of commonly seen in source code.

Other Languages

Creating a self-referential type in Pascal requires the definition of two type names.

```
1  type
2     TNODE_PTR = ^TNODE; (* special rules covers this forward reference case *)
3     TNODE = record
4             count : integer;
5             left,
6             right : TNODE_PTR
7             end;
```

1474 EXAMPLE 2 To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

EXAMPLE mutually referential structures

```
struct s1 { struct s2 *s2p; /* ... */ }; // D1
struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

specify a pair of structures that contain pointers to each other. Note, however, that if **s2** were already declared as a tag in an enclosing scope, the declaration **D1** would refer to *it*, not to the tag **s2** declared in **D2**. To eliminate this context sensitivity, the declaration

```
struct s2;
```

may be inserted ahead of **D1**. This declares a new tag **s2** in the inner scope; the declaration **D2** then completes the specification of the new type.

Commentary

```
1  struct s2 { int mem1; };
2
3  void f(void)
4  {
5  struct s2;
6  struct s1 { struct s2 *s2p; /* ... */ };
7  struct s2 { struct s1 *s1p; /* ... */ };
8  }
```

without the declaration of the tag `s2` in the body of `f`, the declaration at file scope would be visible and the member `s2p` would refer to it, rather than the subsequence definition in the same scope,

C++

This form of declaration would not have the desired affect in C++ because the braces form a scope. The declaration of `s2` would need to be completed within that scope, unless there was a prior visible declaration it could refer to.

1475 **Forward references:** declarators (6.7.5), array declarators (6.7.5.2), type definitions (6.7.7).

References

1. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.