C23 Proposal - WG14 N3032
Title: **Improved Rules for Tag Compatibility (updates N3003)**
Author: **Martin Uecker**
Date: **2022-07-20**

Introduction

N2366 proposed new rules for tag compatibility to make the language more consistent and to simplify generic programming using tagged types and was favorably received. Previous discussions can be found in N2377 (London minutes), N3004 (May 2022 minutes), and reflector messages 17171 ff., 21374 ff.

Tagged types with the same tag and content are already compatible across translation units but never inside the same translation unit. In the following example, ‘p’ and ‘r’ and also ‘q’ and ‘r’ have compatible types, while ‘p’ and ‘q’ do not have compatible types.

Example 1:

```
// TU 2
struct foo { int a; } p;

void bar(void)
{
  struct foo { int a; } q;
}


// TU 3
struct foo { int a; } r;
```

Here, it is proposed to 1) make tagged types with the same tag and content compatible inside the same translation unit and to 2) allow redeclaration of the same tagged type. This would make the language more consistent and facilitate the implementation and use of generic data structures (such as the generic implementation of tree and list types provided with BSD systems, or the recently discussed _Either type, or many other). See N2366 for a detailed discussion.

Example 2 (C17)

```
// header

#define PRODUCT_DECL(A, B) \
     struct prod { A a; B b; }

#define SUM_DECL(A, B) \
     struct sum { bool flag; union { A a; B b; }; }


// unique forward declarations for all combinations, header?

typedef SUM_DECL(float, double) sum_float_double;

typedef PRODUCT_DECL(int, sum_float_double)
     prod_int_sum_float_double;


// c file

void foo(product_int_sum_float_double x);

void bar(prod_int_sum_float_double y)
{
     foo(y);
}
```

Example 3 (C23, proposed)

```
// header

#define PRODUCT(A ,B) \
     struct prod { A a; B b; }

#define SUM(A, B) \
     struct sum { bool flag; union { A a; B b; }; }

// c file

void foo(PRODUCT(int, SUM(float, double)) x);

void bar(PRODUCT(int, SUM(float, double)) y)
{
     foo(y);   // compatible type
}
```

Summary of the Proposed Rules

Two changes are proposed:

1. Tagged types that have the same tag name and content become compatible not only across translation units but also inside the same translation unit, using exactly the same rules.

2. Redeclaration of the same tagged types is allowed using a new structural definition of same type for tagged types.

Implementation Experience

A prototype implementation of the proposed rules exist for GCC.
https://github.com/uecker/gcc/tree/tagcompat

Changes from N3003

1. Insert “ `structure, union or enum type”` at the end of 6.2.7 and in the footnote to improve clarity based on comments from the reflector. (highlighted)

2. Change “their members fulfill the requirements above” to “they fulfill the requirements above” to be compatible with N3021 in case it is adopted into C23. (highlighted)

Changes from N2863

1. Types without tags remain incompatible, so that

typedef struct { int x; } a_t;
typedef struct { int x; } b_t;

remain two incompatible types.
2. Incomplete types remain incompatible inside a single translation unit and for a redeclaration the redeclared type becomes incompatible again inside declaration until the end of the declaration.
3. Explicit wording for anonymous structure and union types was added.
4. As the rules stay closer to the existing rules fewer textual changes are required.

Changes from N2366

The rules for redeclaration were revised based on comments from the reflector and experience with the prototype implementation.

The second change proposed in N2366 related to compatibility of incomplete types that are never completed. This change is not included here and will be revisited at a late time, as the addressed problem is independent from the proposed changes and also affects other types not discussed here.

Suggested Wording Changes (underlined text exists only if N3021/N3030 is accepted)

6.2.7 Compatible type and composite type

1 Two types are compatible types if they are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators. 56) Moreover, two **complete** structure, union, or enumerated types **declared with the same tag** ~~**declared in separate translation units**~~ are compatible if their ~~**tags and**~~ members satisfy the following requirements: ~~**If one is declared with a tag, the other shall be**~~ ~~**Both are declared with the same tag.**~~ ~~**If both are completed anywhere within their respective translation units, then the following additional requirements apply:**~~ there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values; if one has a fixed underlying type then the other shall have a compatible fixed underlying type. **For determining type compatibility, anonymous structures or unions are considered a regular member of the containing structure or union type, and the type of an anonymous structure or union is considered compatible to the type of another anonymous structure or union, respectively, if their members fulfill above requirements.**
Furthermore, two structure, union, or enumerated types declared in separate translation units are compatible in the following cases:

- both are declared without tags and they fulfill the requirements above;
- both have the same tag and are completed somewhere in their respective translation units and they fulfill the requirements above;
- both have the same tag and at least one of the two types is not completed in its translation unit.

Otherwise, the structure, union or enum types are incompatible. XXX)

Footnote XXX) A structure, union, or enum type without a tag or an incomplete structure, union or enum type is not compatible with any other structure, union or enum type declared in the same translation unit,

6.7. Declarations

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

– a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;
– **enumeration constants and** tags may be redeclared as specified in 6.7.2.2 **and** 6.7.2.3**, respectively**.

6.7.2.1 Structure and union specifiers

11 ~~The presence of a member declaration list in a struct-or-union specifier declares a new type, within a translation unit.~~ The member declaration list is a sequence of declarations for the members of the structure or union. If the member declaration list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined.**YYY)** ~~The type is incomplete until immediately after the } that terminates the list, and complete thereafter.~~

Footnote YYY) For further rules affecting compatibility and completeness of structure or union types see 6.2.7 and 6.7.2.3.

6.7.2.2 Enumeration specifiers

5 Enumeration constants can be redefined in the same scope with the same value as part of a redeclaration of the same enumerated type.

~~5~~**6** Each enumerated type shall be compatible with char , a signed integer type, or an unsigned integer type. The choice of type is implementation-defined, 140) but shall be capable of representing the values of all the members of the enumeration. **ZZZ)** ~~The enumerated type is incomplete until immediately after the } that terminates the list of enumerator declarations, and complete thereafter.~~

Footnote ZZZ) For further rules affecting compatibility and completeness of enumerated types see 6.2.7 and 6.7.2.3.

6.7.2.3 Tags

~~1 A specific type shall have its content defined at most once.~~
~~2~~ **1** Where two declarations that use the same tag declare the same type, they shall both use the same choice of struct, union, or enum. **If two declarations of the same type have a member-declaration or enumerator-list, both declarations shall fulfill all requirements of compatible types (6.2.7) with the additional requirement that corresponding members of structure or union types shall have the same (and not merely compatible) types.**

Semantics

5 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. Irrespective of whether there is a tag or what other declarations of the type are in the same translation unit, the type is incomplete 144) **from the of beginning of the specifier** until immediately after the closing brace of the list defining the content, and complete thereafter **until the beginning of the next specifier that redeclares the same type later in the same translation unit (if any) or otherwise until the end of the translation unit.**

Editorial note: footnote 144 should be moved to the definition of incomplete type, 6.2.5.

14 EXAMPLE 3 The following example shows allowed redeclarations of the same structure, union, or enumerated type in the same scope:

```
struct foo { struct { int x; }; };
struct foo { struct { int x; }; };

union bar { int x; float y; };
union bar { float y; int x; };

typedef struct q { int x; } q_t;
typedef struct q { int x; } q_t;

void foo(void)
{
     struct S { int x; };
     struct T { struct S s; };
     struct S { int x; };
     struct T { struct S s; };
}

enum X { A = 1, B = 1 + 1 };
enum X { B = 2, A = 1 };
```

15 EXAMPLE 4 The following example shows invalid redeclarations of the same structure, union, or enumerated type in the same scope:

```
struct foo { int (*p)[3]; };
struct foo { int (*p)[]; };     // member has different type

union bar { int x; float y; };
union bar { int z; float y; }; // member has different name

typedef struct { int x; } q_t;
typedef struct { int x; } q_t; // not the same type

struct S { int x; };

void foo(void)
{
  struct T { struct S s; };
  struct S { int x; };
  struct T { struct S s; }; // struct S not the same type
}

enum X { A = 1, B = 2 };
enum X { A = 1, B = 3 }; // different enumeration constant

enum R { C = 1 };
enum Q { C = 1 }; // conflicting enumeration constant
enum Q { C = C }; // ok!
```