C23 Proposal - WG14 N2863
Title: Improved Rules for Tag Compatibility (updates N2366)
Author: Martin Uecker
Date: 2021-11-28

Introduction

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

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 { A a; B b; }

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

// c-file

typedef sum_decl(float, double) sum_float_double;

typedef product_decl(int, sum_float_double)
    prod_int_sum_float_double;

void foo(product_int_sum_float_double x);

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

Example 3 (proposed rules)

```
// header

#define product(A ,B) struct { A a; B b; }

#define sum(A, B) \
struct { 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.

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.

Implementation Experience

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

This preliminary implementation demonstrates feasibility of implementation, compatibility with existing C code, and did not reveal any unexpected problems.

Suggested Wording Changes

6.2.7 Compatible type and composite type

1 Two types have compatible type if their types 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 structure, union, or enumerated types ~~**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 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.

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;
– tags **and enumeration constants** may be redeclared as specified in 6.7.2.3.

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. The type is incomplete until immediately after the } that terminates the list **of the first declaration of the type**, and complete thereafter.

6.7.2.2 Enumeration specifiers

5 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. The enumerated type is incomplete until immediately after the } that terminates the list of enumerator declarations **of the first declaration of the type**, and complete thereafter.

6.7.2.3 Tags

~~1 A specific type shall have its content defined at most once.~~

~~2~~ 1 ~~Where~~ If two declarations ~~that~~ use the same tag **in the same scope** ~~declare the same type~~, they shall both use the same choice of struct, union, or enum. **If both have a member-declaration or enumerator-list, both declarations shall specify the same type as defined below.**

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

Semantics

5 **Two declarations for structure, union, or enumerated types with a member-declaration or a enumerator-list, declare the same type if they fulfill all requirements of compatible types (6.2.7), and if corresponding members of structure and union types have the same type. XXX)**—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 142) until immediately after the closing brace of the list defining the content, and complete thereafter.~~

XXX) Members of an anonymous structure or union are considered members of the containing structure or union (6.7.2.1).

6 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. ~~Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.~~

Editorial note: footnote 142 to be moved to the definition of incomplete type, 6.2.5.

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

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; };

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

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