Issue 1018: `enum` compatibility

Number: n3980
Date: 2026-11-15
Author: Martin Uecker

Introduction

Before C23 it was the case that if an incomplete type did not specify its content, also its size was not known. C23 introduced enumerations with fixed underlying type that can be forward declared. Since then, „not knowing the content“ and „not knowing the size“ are two different conditions. A complete type is currently defined as a type with known size.

6.2.5 Types

> … At various points within a translation unit an object type may be incomplete28) (lacking sufficient information to determine the size of objects of that type) or complete (having sufficient information).

In general, the standard assumes that we can determine type compatibility for two complete tagged types.

6.2.7 Compatible type and composite type

> 1 Moreover, two **complete** structure, union, or **enumerated types declared with the same tag are compatible if members satisfy the following requirements:**
> — 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 a complete array type, the other is declared with a complete array type
> — 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 with the following additional requirement for enumerated types.
>
> 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.

This is broken in C23 because the conditions related to members can not be verified for an enumerated type with fixed underlying type if it only has a forward declaration. The issue also affects compatibility across translation units.

> 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 preceding requirements;
> **— both have the same tag and are completed somewhere in their respective translation units and they fulfill the preceding requirements;**
> — both have the same tag and at least one of the two types is not completed in its translation unit.

We spell out two solutions which seem the most reasonable, although various other options exist.

Solution A: The simplest solution is to treat types without content as incompatible to other types, just as we do for incomplete types. The first sentence could be changed to:

> Moreover, two complete structure, union, or enumerated types declared with the same tag are compatible if **both declare their content and** the members satisfy the following requirement:

Rationale: This is the easiest fix. We should then consider whether we want to adapt the complete / incomplete notion, or add new terminology. One potential issue (that already exists) is that adding enumeration constants to an API makes the type incompatible, which reduces types safety.

```
{ enum e : int; extern enum e a; }              // reference
{ enum e : int; extern enum e a; }              // incompatible
{ extern enum e : int { A = 1 } a; }            // incompatible

{ extern enum e : int { C = 1 } a; }            // reference
{ extern enum e : int { C = 1 } a; }            // compatible
{ extern enum e : int { C = 2 } a; }            // incompatible
{ enum e : int; extern enum a; }                // incompatible
{ extern enum e : int { B = 1 } a; }            // incompatible
```

Solution B: We could relax the requirement of having the exact same list of enumeration constants for type compatibility for enumerated types with fixed underlying type.

> Moreover, two complete structure, union, or enumerated types declared with the same tag are compatible **if both are enumerated types and both have a fixed underlying type that is compatible, or when** the members satisfy the following requirement:
>
> ….
>
> For two enumerations, corresponding members**, when declared as part of both types,** shall have the same values. If one has a fixed underlying type, then the other shall have a compatible fixed underlying type.

Rationale: This is would more generally fit to the idea that enumerated types define a set of (named) values. The types are compatible as long as the common subset of members is consistent. For example, this could be used to selectively expose part of an API or to extend it later, enabling better type checking across translation units. It could also enable interesting logic with _Generic.

```
{ enum e : int; extern enum e a; }              // reference
{ enum e : int; extern enum e a; }              // compatible
{ extern enum e : int { A = 1 } a; }            // compatible
{ extern enum e : int { B = 1 } a; }            // compatible

{ enum e : int { C = 1 }; }                     // reference
{ extern enum e : int { C = 1 } a; }            // compatible
{ extern enum e : int { C = 2 } a; }            // incompatible
{ enum e : int; extern enum e a; }              // compatible
{ extern enum e : int { B = 1 } a; }            // compatible
{ extern enum e : int { C = 1, D = 2 } a; }     // compatible

// similar example using array types
{ extern int (*a)[][3]; }                       // reference
{ extern int (*a)[1][3]; }                      // compatible
{ extern int (*a)[2][4]; }                      // incompatible
```