Regarding DR314

DR314 asks three questions. The proposed committee response to these questions is flawed by neither quoting the sections of the Standard that apply nor explaining the reasoning behind the answers. I believe that the answers to the first two questions uncontroversial and can be easily justified by the text of the standard. The third proposed answer has the potential to cause much mischief and invalidate a lot of reasonable, widespread code.

The issues concern type compatibility of objects with struct type declared in different translation units. I will present a series of examples, and so how the text of the Standard applies. The part of the Standard that controls is Subclause 6.2.7, Paragraphs 1 and 2, quoted in the entirety below. I've underlined the text in Paragraph 1 that is particularly important:

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 6.7.5 for declarators.[46)] 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 complete types, 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, and such that if one member of a corresponding pair is declared with a name, the other member 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.

2 All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

DR314 Question 1

Before the examples, I'll suggest improved wording for question 1 in the DR.

Question 1: Does 6.2.7#2 refer to the types immediately after the declarations, or the types at any point where the declarations are in scope?

Subclause 6.2.7 Paragraph 2 makes a statement about *all* declarations of the same object or function, regardless of where the declarations that object or function are. It requires that all declarations of the same object or function, even if those declarations are in different translation units of the program, to have compatible type.

Note also that if an object with struct or union type is declared with an incomplete type, and that type is later completed in the same scope, the type of the declaration is the completed type (Subclause 6.2.5, Paragraph 22). Under such conditions, the type of the object or function is the completed type, and that type must be compatible with any other declarations.

Example 1

```
// Translation Unit 1
struct S {int a;};
extern struct S *x;

// Translation Unit 2
struct S {int a;};
extern struct S *x;
```

There is no undefined behavior in these two translation units. Subclause 6.2.7 Paragraph 2 is met because both declarations of the object **`x`** have pointer types that are compatible. The pointer types are compatible because the types they point to (namely **`struct s)`** are compatible types. We know that **`struct S`** in translation unit 1 is compatible with **`struct S`** in translation unit 2 by Paragraph 1 because:

- Both are declared with the same tag, namely **`S`**
- Both are complete types, and
    - Both have a one-to-one correspondence between their struct members.
    - The corresponding struct members have compatible type, namely **`int`**
    - The struct members are declared in the same order
    - The corresponding struct members have the same name, namely **`a`**

Example 2

```
// Translation Unit 1
struct S {int a;};
extern struct S *x;

// Translation Unit 2
struct S;
extern struct S *x;
```

There is no undefined behavior in these two translation units. Subclause 6.2.7 Paragraph 2 is met because both declarations of the object **`x`** have pointer types that are compatible. The pointer types are compatible because the types they point to (namely **`struct s)`** are compatible types. We know that **`struct S`** in translation unit 1 is compatible with **`struct S`** in translation unit 2 by Paragraph 1 because:

- Both are declared with the same tag, namely S
- Both are not complete types (therefore, members do not have to match).

Example 3

```
// Translation Unit 1
struct S {int a;};
extern struct S *x;

// Translation Unit 2
struct S;
extern struct S *x;

// Translation Unit 3
struct S {float a;};
extern struct S *x;
```

There is undefined behavior.

The declaration of **`x`** in translation unit 1 *has* compatible type with **`x`** in translation unit 2 (just like Example 2).

The declaration of **`x`** in translation unit 2 *has* compatible type with **`x`** in translation unit 3 (just like Example 2).

But, the declaration of **`x`** in Translation Unit 1 *does not have* compatible type with **`x`** in translation unit 3: In translation unit 1 **`struct S`** has an int member, and in translation unit 3 **`struct S`** has a float member.

Subclause 6.2.7 Paragraph 2 requires that all declarations of the same object or function have compatible type. It doesn't matter if some of the declarations of the same object have compatible type, if one of the declarations does not compatible type with any of the others, there is undefined behavior.

Example 3 is a simplified version of Example 4 from DR314.

Example 4 (Question 2 in DR314)

```
// Translation Unit 1:
extern struct t *x;
struct s;
struct t { struct s *a; };

// Translation Unit 2:
extern struct t *x;
struct s { int p; };
struct t { struct s *a; };

// Translation Unit 3:
extern struct t *x;
struct s { long q; };
struct t { struct s *a; };
```

Although the object x is initially declared to be a pointer to incomplete type struct t, that type is completed in the same scope as the declaration of x in all three translation units. Therefore by Subclause 6.2.5 Paragraph 22, the type of x is a pointer to struct t, a complete type whose sole member named **a** has type pointer to **struct s**.

There is undefined behavior because the declaration of **x** in translation unit 2 does not have compatible type with the declaration of **x** in translation unit 3. The types are not compatible because the type **struct t** in translation unit 2 does not have compatible type with the type **struct t** in translation unit 3. Those types are not compatible because the **a** member of **struct t** in translation unit 2 does not have compatible type with the **a** member of **struct t** in translation unit 3. Those types are not compatible because the type **struct s** in translation unit 2 does not have compatible type with the type **struct s** in translation unit 3.

Note that the declaration of **x** in translation unit 1 is compatible with **x** in translation unit 2 and **x** in translation unit 3. The reason is that the type **struct s** in translation unit 1 is a compatible type with **struct s** in translation unit 2 and the type **struct s** in translation unit 1 is a compatible type with **struct s** in translation unit 3.

But, since Subclause 6.2.7 Paragraph 2 requires all declarations of **x** to have compatible type, there is undefined behavior.

Example 5

```
// Translation Unit 1
#include <stdio.h>
struct s {int i;};
static struct s x = {0};
extern void f(void);
int main()
{
 f();
 return x.i;
}

// Translation Unit 2
struct s {float f;};
static struct s y = {3.14};
void f()
{
 return;
}
```

There is no undefined behavior. Note that Subclause 6.2.7 Paragraph 2 only requires that the declarations of the same objects or functions have compatible type. The only object or function declared more that once is function **`f`**, and both of its declarations have compatible type.

There is no requirement to ever ask if the **`struct s`** in translation unit 1 has compatible type as **`struct s`** in translation unit 2. **`struct s`** in both translation units is purely a "local" type.

I believe that this is the answer expected by most C programmers. When I write a struct declaration I don't ask myself the question, "Is there somewhere else in the large program, including all of the code written by others or in the library, that someone else wrote a struct with the same name that I have to match?"

Example 6

```
// Translation Unit 1
#include <stdio.h>
struct s {int i;};
extern struct s x = {0};
extern void f(void);
int main()
{
 f();
 return x.i;
}

// Translation Unit 2
struct s {float f;};
extern struct s y = {3.14};
void f()
{
 return;
}
```

There is no undefined behavior. Although the objects x and y now have external linkage, there is still only one declaration of x and y, and there is no requirement to ask if the **`struct s`** in translation unit 1 has compatible type as **`struct s`** in translation unit 2.

Example 7

```
// Translation Unit 1
#include <stdio.h>
struct s {int i;};
extern struct s x = {0};
extern void f(void);
int main()
{
 f();
 return x.i;
}

// Translation Unit 2
struct s {float f;};
extern struct s y = {3.14};
void f()
{
 return;
}

// Translation Unit 3
struct s {int i;};
extern struct s x;

// Translation Unit 4
struct s {float f;};
extern struct s y;
```

There is no undefined behavior. Although there are two declarations of the object **`x`** (translation unit 1 and translation unit 3), both have compatible type under the rules of Subclause 6.2.7 Paragraph 1. Likewise, the two declarations of object y (translation unit 2 and translation unit 4) have compatible type. There is no requirement that object **`x`** have compatible type with object **`y`**, because they are different objects. There is no reason to care if **`struct s`** in translation unit 1 is a compatible type with **`struct s`** in translation unit 2.

Although this example might seem forced, I am sure that similar code appears in many large programs. Large programs are built upon subsystems containing subsystems, and libraries calling other libraries. It is common in such environments to share lots of code, with a large interface, locally within a subsystem, and then have a more constrained interface to other subsystems. For example, if I write a program using X windows, I don't care if two or more modules within X windows use a **`struct s`**, as long as nothing in the interface to X windows that I use has a **`struct s`**. I'll declare my

objects of **`struct s`** consistently, and they can declare their objects of **`struct s`** consistently. Neither one of us needs to worry about either other's "private" use of the struct named **`s`**.

Conclusion

In C, struct, union, and enum types in different translation units only need be compatible if necessary to determine whether a set of declarations of a particular object or a particular function need to be compatible.

If there is a need to determine whether two or more struct, union, and enum types in different translation units are compatible types, Subclause 6.2.7 Paragraph 1 contains the definition needed.