ISO/IEC JTC 1/SC 22/WG14 **N 2366**

March 30, 2019 **v 1**

Improved Rules for Tag Compatibility
Modification request for C2x

Martin Uecker

With some minor changes to the tag compatibility rules, we can remove some existing consistencies, facilitate generic programming using macros, and enable the construction of algebraic types. This is a substantially extended and revised version of N2105[1] taking into account the comments from the reflector.

1. INTRODUCTION: TYPE IDENTITY AND COMPATIBILITY

1.1. Overview

In a structural type system equivalence of types is determined by comparing the structure of the types in question. In a nominal type system, only types with the same name are equivalent. C mixes concepts from a structural type system with elements of a nominal type system in sometimes complicated ways. To understand all rules, one has to differentiate between identity of types ('same type') and compatibilty of types which is a relaxed notion of equivalence between types (**6.2.7**) and of crucial importance to the C language.

Identity:. For all types except tagged types (i.e. structs, unions, enums), the types are identified by their structural descriptions which can be part of a declaration (**6.7**) or part of type names (**6.7.7**). If these descriptions at two different locations in the programm are identical also the denoted types are identical ('same type'). Typedef names can be introduced as synonyms using typedef. A typedef name can be declared multiple times for same type (**6.7, p3**). For tagged types completely different rules apply which also take into account whether the syntactical description contains a tag and whether it contains a description of the structure of the type (i.e. struct-declaration-list or enumerator-list). Currently, each declaration of a union or struct with struct-declaration list declares a different type irrespective of tag name and structure. The full details and the proposed changes are discussed later.

Compatibility:. Additional rules specify compatibility between types which are not identical. For non-tagged types these are based only on the structure of the type. Inside a translation unit all tagged types which are not identical are always considered incompatible irrespective of tag name or structure. In contrast, between different translation units tagged types are considered compatible exactly when both structure and tagname are identical.

Note:. There is also some concept of two types having the same representation, which is even weaker than compatibility. It seems that compatibility of two types implies that they must have the same representation.

1.2. Criticism

While all other types follow basic structural typing rules for identity and compatibility, tagged types have much more complicated rules which are different when applied to types declared inside the same or in different translation units. Implementations and tools performing global program analysis have to track the translation unit of the type to determine compatibility and aliasing sets. While it is a common and expected property of type systems that mutual compatibility of types is transitive, this is not the case in C: The sets of types which are mutually compatible to each other do not form equivalence classes. In the following example, TYPE 1 is compatible to TYPE 2 and TYPE 3, but TYPE 2 and TYPE 3 are not compatible.

```c
// translation unit 1:
typedef struct foo { int a; } TYPE1;

// translation unit 2:
typedef struct foo { int a; } TYPE2;

void bar(void)
{
  typedef struct foo { int a; } TYPE3;
}
```

When the analysis is restricted to a single translation unit, this situation is not visible. On the other hand, global analysis is important in many situations - such as correctness proofs, global optimization (e.g. link-time optimization), or refactoring.
Another major downside of the current rules is that it is not practical to declare tagged types inline: They always require a common previous declaration. While syntactically it is legal to declare tagged types inline, such types can not be made compatible to any other structurally equivalent declaration. These declarations are therefor almost completely useless. An example of such useless declarations are declarations in prototype scope. Removing this restriction does not only remove such surprising limitations, but also opens up a lot of new possibilities, e.g. to use these types in macros to define generic types.

2. PART 1: CONSISTENT RULES FOR COMPATIBILITY

Proposed change # 1:. Remove the five words "declared in separate translation units" in **6.2.7**.

This is the core of the proposal. The effect of this change would be that the rules for compatibility of structs and unions introduced in **6.2.7**. would not only apply across but also inside the same translation unit. As the only effect of this change is to make more types compatible, this is a very safe change to make. The advantages and also potential downsides are are discussed later.

3. PART 2: INCOMPLETE TYPES

If one type is not completed, then **6.2.7, p1** implies that types with the same tag are compatible when declared in the same translation unit. Already according to the existing rules it can happen that incomplete types must at the same time be compatible to different mutually incompatible types from a different translation unit.

```c
// TU 1
void f(void)
{
  struct s;
  extern struct s *a, *b;
}
// TU 2
void h(void)
{
  struct s { int a; };
  extern struct s *a;
}

```

```c
void i(void)
{
  struct s { double a; };
  extern struct s *b;
}
```

With the proposed change, this would then also be possible inside a single translation unit. To fix both problems, we require that incomplete types must be compatible to only a single equivalence class of compatible types. To achieve this, we add one constraint:

Proposed Change #2:. At the end of **6.2.7, p2.** add the following sentence:
"For all incomplete types that are not completed in their translation unit, it must be possible to specify completed types so that this condition is fulfilled."

Based on these changee, we consider three examples:

Example 1:. The incomplete type is never completed and with our proposal is now considered to be compatible to a different type with the same tag.

```c
struct s { int a; } x;
void f(void)
{
  struct s; // new type considered compatible
  struct s *p = &x;
}
```

Example 2:. In constrast, when the type is later completed and then turns out to be incompatible, this is then undefined behavior (as before).

```c
struct s { int a; } x;
void f(void)
{
  struct s; // new type
  struct s *p = &x;
  struct s { double b; }; // .. turns out to be incompatible
}
```

Example 3:. Finally, as was previously only possible when considering separate translation units, it is now possible that an incomplete type which is never completed could at the same time be required to be compatible to different mutually incompatible completed types.

```c
struct s { int a; } x;
void f(void)
{
  struct s { double b; } y; // different and not compatible
  {
    struct s; // new type
    struct s *px = &x; // compatible
    struct s *py = &y; // compatible
  }
}
```

This final example is undefined behavior because there is no possible completed type that simultaneously fulfills **6.2.7, p2.** with respect to both objects.

4. PART 3: MULTIPLE DECLARATIONS OF CONTENT

With the previous changes two tagged types with the same tag name and content are compatible, but two such declarations can still not occur in the same scope. To fully exploit all benefits of this proposal it must also be possible to declare the same type multiple times in the same scope. In the following, the existing rules and the required changes are discussed.

4.1. Summary of Existing Rules

In the following the existing rules regarding tags are summarized:

— Each declaration of a union or struct with struct-declaration list declares a different type irrespective of tagname and structure (**6.7.2.1, p8** and **6.7.2.3, p1**).
— Otherwise all declarations in the same scope with the same tag refer to same type with the content declared only once. The type is completed when the content is declared and incomplete before (**6.7.2.3, p4**).
— Each declaration with a different tagname or which is in different scope declares a different type (**6.7.2.3, p5**). In particular, it is possible to use the same tagname for a different type when it is declared in a different scope.

To illustrate the situation for the case with tag, the follow table shows whether a declaration declares a new **complete** or **incomplete** type or whether it refers to an **existing** (complete or incomplete) **type**:

tag	A. declaration with content	B. declaration without content	C. usage
1. not visible	complete type (new)	incomplete* type (new)	incomplete* type (new)
2. different scope	complete type (new)	incomplete* type (new)	existing type
3. same scope, complete	**constraint violation!**	existing type	existing type
4. same scope, incomplete	completes existing type	existing type	existing type

(*) only struct or union but not enum.

Four cases need to be differentiated:

(1) tag **not visible**
(2) tag visible from **different** (enclosing) **scope**
(3) tag visible from **same scope** and refers to **complete** type
(4) tag visible from **same scope** and refers to **incomplete** type

There are also three different usage forms:

A **declaration with** struct-declaration list (**content**) (**6.7.2.3, p6**)

```
struct foo { int a; };
struct foo { int a; } *x;
enum bar { BAR = 1 };
enum bar { BAR = 1 } *y;
```

B **declaration without** struct-declaration list (**content**) (**6.7.2.3, p7**)

```
struct foo ;
```

C **usage** (without struct-declaration list) (**6.7.2.3, p8,9**)

```
struct foo *x;
enum bar *y;
```

In a nutshell, the third part of the proposal is simply to make the exceptional case A.3 legal and refer to the existing type while adding the constraint that the content must be identical to the previous declaration.

Note 1:. Conceptually, this is not really different to declaring a typedef name for the same type multiple times. Making this possible also for tags would further improve the consistency of the language.

Note 2:. For declarations without tag there is only one case and the declared type is always unique. With the first part of the proposal, these unique types are compatible if they fulfill the conditions of **6.2.7**.

Note 3:. Unrelated to this proposal, one could consider deprecating C.1 which only exists for structs or unions but not for enums (**6.7.2.3, p8**). It is inconsistent with the other cases which all require a visible declaration and violates the general principle of declaration before use. Then A would allways declare the content of a type, B would always refer to a type that may be completed elsewhere in the same scope, and C would always refer to a type which must already exist.

4.2. Proposed Changes

To make redeclaration of the same type possible, we suggest a couple of changes to **6.7.2.1** “Structure or union specifiers”, **6.7.2.2** “Enumeration specifiers”, and **6.7.2.3** “Tags”. In addition, some minor changes to the footnotes are suggested (see Appendix).

Proposed Change #3:. In **6.7.2.1, p8** the first sentence “The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. ...” is removed and in the last sentence “The type is incomplete until immediately after the } that terminates the list, and complete thereafter.” the term “of the first declaration of the type” is added behind “list”. In **6.7.2.2, p4**, we change “The enumerated type is incomplete until immediately after the } that terminates the list of enumerator declarations, and complete thereafter” to include the term “of the first declaration of the type” after “list of enumerator declarations”.

Proposed Change #4:. In **6.7.2.3, p1**, we remove the following constraint: “A specific type shall have its content defined at most once.”

Proposed Change #5:. In **6.7.2.3, p2** the constraint “Where two declarations that use the same tag declare the same type, they shall both use the same choice of struct, union, or enum.” is replaced by “Where two type specifiers for structure, union, or enumerated types refer to the same type, they shall both use the same choice of struct, union, or enum and the same tag. If they are declarations with a member declaration list or enumerator list, they shall declare the same members in the same order.” with additional explanations in a footnote.

Proposed Change #6:. In **6.7.2.3, p4**, we suggest ot remove the following sentence completely because it is redundant: “Irrespective of whether there is a tag or what other declarations of the type are in the same translation unit, the type is incomplete until immediately after the closing brace of the list defining the content, and complete thereafter.”

Note: These changes imply that the type is then complete inside later declarations. One could think that this could allow the definition of recursive types that contain themselves, but because these later declarations are required to be identical to the first complete declaration this cannot happen.

```
struct foo {
        char (*x)[8];
};

struct foo {
        char (*x)[sizeof(struct foo)];
};
```

In this example the `sizeof` would be allowed because the type is complete but the requirement for the types to be the same implies that `8 == sizeof(struct foo)` which then fulfills new constraint in **6.7.2.3, p2**. (Alternatively, one could consider making the type incomplete again inside the repeated struct-declaration-list.)

5. ADVANTAGES

There are several advantages:

(1) The language is simplified as constraints are removed.
(2) The rules of the language are more consistent as the same rules apply inside a single translation unit as between different translation units.
(3) Global analysis is easier as compatibility of types depends only on inherent properties of the types and not also on the relative location in the program code.
(4) Formal reasoning about aliasing is simplified as compatibility of structs and unions is now transitive.
(5) Functions can no longer become undefined/defined merely by moving them from one translation unit to another (assuming no other interaction with the environment). The following example is valid only if the two function are located in different translation units:

```
int bar(void)
{
        struct ex { int a; } x = { 1 };
        return foo(&x);
}

int foo(void *p)
{
        struct ex { int a; }* x = p;
        return x->a;
}
```

(6) Structs or unions in prototype scope are not useless anymore because they are compatible to types with the same tag and structure.
(7) It is now practical to define generic types in macros.

Generic data types:. The proposed changes simplify a widely used programming idiom: When defining generic types in macros depending on a parameter those can now directly be used in-place instead of having to declare each instance separately before use. For example, this affects common implementations of type-generic container data structures (e.g. BSD's `sys/queue.h` and `sys/tree.h`).
By removing the requirement to declare the types first, useful data types can be declared inline, which makes it practical to compose complex generic data structures from smaller building blocks inside macros. In particular, a full system of algebraic types can then easily be constructed also in C similar to many other languages, as illustrated in the following example:

```
#define product_type(A, B) \
struct product_tag { A a; B b; }

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


void foo(product_type(int, sum_type(float, double)) x);

product_type(int, sum_type(float, double)) y;
...
foo(y); // proposal: types of x and y become compatible
...
```

For another example, see the recent discussion on the reflector about a sum type '_Either' used to specify a return type for error handling.

6. POTENTIAL DISADVANTAGES

Effects on strictly conforming programs:. As the only two changes of this proposal are to make more types compatible and to make full redeclaration of tagged types possible, the main effect is to make more programs legal. The only strictly conforming progams affected are those who explicitely test for compatibility of types using **`_Generic`**. It is very unlikely that such a program would depend on the fact that otherwise identical tagged types are not compatible.

Loss of type safety:. For APIs where two different but structural equivalent structs without tags (or with identical tags) are used to denote types with different semantics, the compiler would not produce an error anymore when arguments of such types are accidentally switched.
In the case with identical tags and according to the old rules, the two types must be declared in different scopes. For this reason, it is very unlikely that this situation occurs in existing programs. Without tags the two types are useless except when used as part of a typedef. Thus, the relevant example is the following:

```c
typedef struct { int x; } A;
typedef struct { int x; } B;

void foo(A a, B b);

A a;
B b;
foo(b, a);
```

With the proposed changes the types are now compatible and it is possible to call `foo(b, a)` without the compiler complaining about incompatible types anymore. Of course, this is exactly the same situation as with other types, but as an error was given before programmers may now rely on this behavior. A simply remedy for the programmer to make the API safe again is to add tags. Because the situation of switched typedef names is easily detected, a compiler could also add a warning when the "wrong" typedef name is used when compiling in C2X mode (a helpful compiler message could suggest to add tags).

Implementation complexity:. For the most part, implementing a test for structural equivalence for tagged types is not more difficult than for other types where it already exists, except that recursion is possible. Recursion can be handled by storing the pairs of visited types in a list or a cache (which typically exists anyway in the compiler so speed up testing for compatibility). Assuming compatibility for already visited pairs then yields well-defined semantics and avoids infinite recursion. A basic implementation of this algorithm can be found here: The only function which needed to be added (`is_compat_struct`) is reproduced here:

```c
static bool is_compat_struct(
  const struct struct_info *a,
  const struct struct_info *b,
  const struct pair* v)
{
  const struct pair v2 = { a, b, v };

  // pair seen before -> assume equivalence
  for (; NULL != v; v = v->link)
    if (   ((a == v->a) && (b == v->b))
        || ((a == v->b) && (b == v->a)))
      return true;

  if (   (a->N != b->N)
      || (0 != strcmp(a->tag, b->tag)))
     return false;

  for (int i = 0; i < a->N; i++)
    if (   (0 != strcmp(a->members[i].name,
                        b->members[i].name))
        || !is_compatible_r(a->members[i].type,
                            b->members[i].type, &v2))
      return false;

  return true;
}
```

Note: This is not expected to be a significant burden to implementors. It should be noted that many existing compilers already implement a version of this algorithm, because they need to to be able to determine compatibility of types across translation units when performing global optimization - removing the difference between compatibility of types declared in the same translation unit and types declared between translation unit would simplify such compilers.

7. COMPARISON TO N1237

N1237 points out the importance of being able to classify compatible types into equivalence classes for optimization and global program analysis. Because the existing rules do not ensure this, the following amendment is proposed there:
"6.2.7 after paragraph 2 insert: There shall exist a partition of all the structure and union types in the program into disjoint classes such that (a) if two types are in the same class, then they are compatible and (b) whenever two structure or union types are required to be compatible, including by (a), they are in the same class. If there does not exist such a partition, the behavior is undefined."
This would simply make the behavior undefined if the set of all types cannot be partitioned into equivalence classes of compatible types. With our proposal where compatibility of types is only based on correspondence of inherent properties of the types themselves (structure and tag names) and not on relative location, it is easy to prove that all complete tagged types can always be partitioned into equivalence classes. For incomplete types we solve the problem by requiring that it is possible to specifiy a completed type. In the following example (from N1237) all struct types would be compatible:

```
// TU 1
void f(void)
{
  struct s { int a; };
  extern struct s a, b;
}

void g(void)
{
  struct s { int a; };
  extern struct s c, d;
}

// TU 2
void h(void)
{
  struct s { int a; };
  extern struct s a, c;
}

void i(void)
{
  struct s { int a; };
  extern struct s b, d;
}
```

Appendix: pages with diffmarks of the proposed changes

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **`_Atomic`**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[49)]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope. [50)]

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[51)] corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[52)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 Further, there is the **`_Atomic`** qualifier. The presence of the **`_Atomic`** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic, qualified, or unqualified type" whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.

28 A pointer to **`void`** shall have the same representation and alignment requirements as a pointer to a character type.[52)] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

29 **EXAMPLE 1** The type designated as "`float *`" has type "pointer to **`float`**". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type — its type is "pointer to const-qualified **`float`**" and is a pointer to a qualified type.

30 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type **`float`**. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

[49)]Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[50)]An incomplete type can only be used when the size of an object of that type is not needed. 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. The specification has to be complete before such a function is called or defined.

[51)]See 6.7.3 regarding qualified array and function types.

[52)]The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; **`signed char`** shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

— the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that produce such a value;

— the `+`, `-`, `*`, `/`, and `%` operators where one operand is a negative zero and the result is zero;

— compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

4 If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that would produce such a value is undefined.

5 The values of any padding bits are unspecified.[58] A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

6 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The *width* of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

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.[59] 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

[58]Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

[59]Two types need not be identical to be compatible.

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. For all incomplete types that are not completed in their translation unit, it must be possible to specify completed types so that this condition is fulfilled.

3 A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

— If both types are array types, the following rules are applied:

- If one type is an array of known constant size, the composite type is an array of that size.
- Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
- Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
- Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
- Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

The element type of the composite type is the composite type of the two element types.

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.

— If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[60] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

Forward references: array declarators (6.7.6.2).

5 **EXAMPLE** Given the following two file scope declarations:

```
    int f(int (*)(), double (*)[3]);
    int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
    int f(int (*)(char *), double (*)[3]);
```

6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the `_Alignas` keyword.

2 A *fundamental alignment* is a valid alignment less than or equal to `_Alignof (max_align_t)`. Fundamental alignments shall be supported by the implementation for objects of all storage durations. The alignment requirements of the following types shall be fundamental alignments:

— all atomic, qualified, or unqualified basic types;

[60] As specified in 6.2.1, the later declaration might hide the prior declaration.

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
  struct-or-union identifier$_{\text{opt}}$ **{** *member-declaration-list* **}**
  struct-or-union identifier

struct-or-union:
  `struct`
  `union`

member-declaration-list:
  member-declaration
  member-declaration-list member-declaration

member-declaration:
  specifier-qualifier-list member-declarator-list$_{\text{opt}}$ **;**
  static_assert-declaration

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{\text{opt}}$
  type-qualifier specifier-qualifier-list$_{\text{opt}}$
  alignment-specifier specifier-qualifier-list$_{\text{opt}}$

member-declarator-list:
  member-declarator
  member-declarator-list **,** *member-declarator*

member-declarator:
  declarator
  declarator$_{\text{opt}}$ **:** *constant-expression*

Constraints

2 A member declaration that does not declare an anonymous structure or anonymous union shall contain a member declarator list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted.[128] If the value is zero, the declaration shall have no declarator.

5 A bit-field shall have a type that is a qualified or unqualified version of **`_Bool`**, **`signed int`**, **`unsigned int`**, or some other implementation-defined type. It is implementation-defined whether atomic types are permitted.

Semantics

6 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

7 Structure and union specifiers have the same form. The keywords **`struct`** and **`union`** indicate that the type being specified is, respectively, a structure type or a union type.

8 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

[128]While the number of bits in a **`_Bool`** object is at least **`CHAR_BIT`**, the width of a **`_Bool`** can be just 1 bit.

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

9 A member of a structure or union may have any complete object type other than a variably modified type.[129] In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[130] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[131] If the value 0 or 1 is stored into a nonzero-width bit-field of type **`_Bool`**, the value of the bit-field shall compare equal to the value stored; a **`_Bool`** bit-field has the semantics of a **`_Bool`**.

11 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[132] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

13 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

14 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

15 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

16 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

17 There may be unnamed padding at the end of a structure or union.

18 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

[129] A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[130] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[131] As specified in 6.7.2 above, if the actual type specifier used is **`int`** or a typedef-name defined as **`int`**, then it is implementation-defined whether the bit-field is signed or unsigned.

[132] An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

Semantics

3 The identifiers in an enumerator list are declared as constants that have type **`int`** and may appear wherever such are permitted.[133] An enumerator with = defines its enumeration constant as the value of the constant expression. If the first enumerator has no =, the value of its enumeration constant is 0. Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

4 Each enumerated type shall be compatible with **`char`**, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined,[134] 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.

5 **EXAMPLE** The following fragment:

```
      enum hue { chartreuse, burgundy, claret=20, winedark };
      enum hue col, *cp;
      col = claret;
      cp = &col;
      if (*cp != burgundy)
            /* ... */
```

makes `hue` the tag of an enumeration, and then declares `col` as an object that has that type and `cp` as a pointer to an object that has that type. The enumerated values are in the set $\{0, 1, 20, 21\}$.

Forward references: tags (6.7.2.3).

6.7.2.3 Tags

Constraints

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

~~Where two declarations that use the same tag declare~~ Where two type specifiers for structure, union, or enumerated types refer to the same type, they shall both use the same choice of **`struct`**, **`union`**, or **`enum`** ~~.~~ and the same tag. If they are declarations with a member declaration list or enumerator list, they shall declare the same members in the same order. [135]

2 A type specifier of the form

`enum` *identifier*

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

Semantics

3 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~~[136] ~~until immediately after the closing brace of the list defining the content, and complete thereafter.~~

Two declarations of structure, union, or enumerated types which are in different scopes or use

[133]Except when identically redeclared as part of the same type, ihe identifiers of enumeration constants declared in the same scope are all required to be distinct from each other and from other identifiers declared in ordinary declarators.

[134]An implementation can delay the choice of which integer type until all enumeration constants have been seen.

[135]This implies that if there are multiple declarations of the same type with member declaration list they shall declare members with the same types, alignments, and names in the same order and with corresponding bit fields having the same size, and if there is a enumerator list, it shall declare the same enumeration constants in the same order. Because nested declarations of a type which appear inside a repeated declaration of the same type are repeated declaration of the corresponding type, this rule applies recursively.

[136]~~An incomplete type can only be used when the size of an object of that type is not needed. 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.) The specification has to be complete before such a function is called or defined.~~

different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

4 A type specifier of the form

> *struct-or-union identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**

or

> **`enum`** *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**

or

> **`enum`** *identifier*$_{\text{opt}}$ **{** *enumerator-list* **, }**

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided,[136] the type specifier also declares the identifier to be the tag of that type.

5 A declaration of the form

> *struct-or-union identifier* **;**

specifies a structure or union type and declares the identifier as a tag of that type.[137]

6 If a type specifier of the form

> *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.[137]

7 If a type specifier of the form

> *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.

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

```
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`.

9 The following alternative formulation uses the **`typedef`** mechanism:

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

[136]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 of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.
[137]A similar construction with **`enum`** does not exist.