

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.2.7 Compatible type and composite type

### Commentary

This clause primarily deals with type compatibility across translation units. Few developers are aware of the concept of *composite type*. It is needed to handle the cases where there is more than one declaration of the same identifier in the same scope and name space.

### Coding Guidelines

Translators are not required to perform type compatibility checks across translation units, and it is very rare to find one that does. Some static analysis tools perform various kinds of cross translation unit type checking.<sup>[1]</sup> Without automated tool support, these checks are unlikely to be carried out. The approach of these coding guideline is to recommend practices that remove the need to perform these checks, e.g., having a single, textually, point of declaration for identifiers.

---

Two types have *compatible type* if their types are the same.

### Commentary

This defines the term *compatible type*. It is possible for two types to be compatible when their types are not the same type. When are two types the same? Other sentences in the standard specify when two types are the same.

### C++

The C++ Standard does not define the term *compatible type*. It either uses the term *same type* or *different type*. The terms *layout-compatible* and *reference-compatible* are defined by the C++ Standard. The specification of layout-compatible structure (9.2p14) and layout compatible union (9.2p15) is based on the same set of rules as the C cross translation unit compatibility rules. The purpose of layout compatibility deals with linking objects written in other languages; C is explicitly called out as one such language.

### Other Languages

All languages that have more than one type specify rules for how objects and values of different types may occur together as operands of operators. Languages in the Pascal family have a much stricter concept of type compatibility than C. It is possible for two types to be incompatible because their type names are different, even if their underlying representation is identical. Languages like Basic, Perl, and PL/1 take a much more laid-back approach, trying to provide conversions between almost any pair of types.

### Coding Guidelines

Developers who have seen the advantages of a stricter definition of compatible type might be tempted to try to create guidelines that implement such a system; for instance, requiring name equivalence for types in C. However, an increase in strictness of the type system can only, practically, be enforced using automatic tool support. Developer training will also be required. Experience suggests that it takes months, if not a year or more, for some developers to be able to design and make use of good type categories. In:

```

1  typedef int frequency;
2  typedef int dpi;
3
4  extern frequency color_red;
5
6  dpi printer_resolution(char *printer_name)
7  {
8  /* ... */
9  return color_red;
10 }
```

the type of the expression returned by `printer_resolution` is compatible with the return type.

The same source rewritten in Ada would result in a diagnostic being issued; the type of the return expression is not compatible with the declared return type of the function. Yes, the developer-defined types,

identifier ??  
declared in one file

compatible type  
if  
same type

qualified type  
to be compatible  
pointer types  
to be compatible  
array type  
to be compatible  
function  
compatible types

footnote 650  
46

type compatibility  
general guideline

frequency and dpi, do have the same underlying basic type, but it makes no sense to return a frequency from this function. The expected return type is dpi.

It is believed that type checking of this kind, coupled with source code that makes extensive use of the appropriate type definitions, reduces the cost of software development (because in many cases unintended or ill-formed constructs are flagged, and can be corrected at translation time). However, there have been no studies investigating the cost effectiveness of strong typing.

---

632 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.<sup>46)</sup>

compatible type  
additional rules

### Commentary

Another way of expressing the idea behind these rules is that compatible types always have the same representation and alignment requirements. The following are the five sets of type compatibility rules:

1. Same type.
2. Structure/union/enumerated types across translation units (the following C sentences).
3. Type specifier rules.
4. Type qualifier rules.
5. Declarator rules.

631 **compati-  
ble type**  
if

**type specifier**  
syntax

**type qualifier**  
syntax

**declarator**  
syntax

---

633 Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements:

compatible  
separate trans-  
lation units

### Commentary

These requirements apply if the structure or union type was declared via a typedef or through any other means. Because there can be more than one declaration of a type in the same translation unit, these requirements really apply to the composite type in each translation unit.

642 **composite  
type**

In the following list of requirements, those that only apply to structures, unions, enumerations, or a combination thereof are explicitly called out as such. There is no requirement that prevents a structure type from being compatible with an appropriate union type; at the time of writing this issue is the subject of an outstanding DR (251). Two types are compatible if they obey both of the following requirements:

- *Tag compatibility.*
  1. If both types have tags, both shall be the same.
  2. If one, or neither, type has a tag, there is no requirement to be obeyed.
- *Member compatibility.* Here the requirement is that for every member in both types there is a corresponding member in the other type that has the following properties:
  1. The corresponding members have a compatible type.
  2. The corresponding members either have the same name or are unnamed.
  3. For structure types, the corresponding members are defined in the same order in their respective definitions.
  4. For structure and union types, the corresponding members shall either both be bit-fields having the same width, or neither shall be bit-fields.
  5. For enumerated types, the corresponding members (the enumeration constants) have the same value.

**C90**

*Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types;*

There were no requirements specified for tag names in C90. Since virtually no implementation performs this check, it is unlikely that any programs will fail to link when using a C99 implementation.

**C++**

The following paragraph applies when both translation units are written in C++.

- 3.2p5 *There can be more than one definition of a class type (clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (clause 14), non-static function template (14.5.5), static data member of a class template (14.5.1.3), member function template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.4) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements.*

There is a specific set of rules for dealing with the case of one or more translation units being written in another language and others being written in C++:

- 7.5p2 *Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a linkage-specification:*

...

*[Note: ... The semantics of a language linkage other than C++ or C are implementation-defined. ]*

which appears to suggest that it is possible to make use of defined behavior when building a program image from components translated using both C++ and C translators.

The C++ Standard also requires that:

- 7.1.5.3p3 *The class-key or **enum** keyword present in the elaborated-type-specifier shall agree in kind with the declaration to which the name in the elaborated-type-specifier refers.*

**Other Languages**

In many languages that support separate compilation, the implementation is required to store information on the types of identifiers defined in one unit that may be used in other units. This information is then read by a translator when already-translated units are referenced. Such a separate compilation model means that there is only ever one declaration of a type. (there is the configuration-management issue caused by changes to an externally visible type needing to be percolated through to any units that reference it through retranslation.)

**Common Implementations**

It is very rare for an implementation to perform cross translation unit checking of structure, union, or enumerated types. Most linkers (translation phase 8) simply match-up identifiers from different translation units, that have the same spelling. The extra checks performed by C++ implementations usually only apply to identifiers having function type (they are needed to support function overloading).

**Coding Guidelines**

Having the same identifier declared in more than one source file opens the door to a modification of a declaration in one source file not being mirrored by equivalent changes in other source files. If there is only one source file containing the declaration, a developer header file, this problem disappears.

Dev ??

The declaration of an incomplete structure or union type may occur in a header file, provided any subsequent completion of its type occurs in only one of the source files making up a complete program.

translation phase 8

type definition only one

identifier ?? declared in one file

**Example**

```

_____ file_1.c _____
1 union u { int m1 : 3; int : 8; int : 8; };

_____ file_2.c _____
1 union u { int m1 : 3; int : 8;          }; /* Not compatible with declaration in file_1.c */

```

634 If one is declared with a tag, the other shall be declared with the same tag.

**Commentary**

The same tag, as in the spelling of the tags, shall be identical (subject to translator limits on the length of internal identifiers). Nothing is said about members here because one or more of the types may be incomplete.

**C90**

This requirement is not specified in the C90 Standard.

Structures declared using different tags are now considered to be different types.

tag  
declared  
with same

```

_____ xfile.c _____
1 #include <stdio.h>
2
3 extern int WG14_N685(struct tag1 *, struct tag1 *);
4
5 struct tag1 {
6     int m1,
7         m2;
8     } st1;
9
10 void f(void)
11 {
12     if (WG14_N685(&st1, &st1))
13     {
14         printf("optimized\n");
15     }
16     else
17     {
18         printf("unoptimized\n");
19     }
20 }

_____ yfile.c _____
1 struct tag2 {
2     int m1,
3         m2;
4     };
5 struct tag3 {
6     int m1,
7         m2;
8     };
9
10 int WG14_N685(struct tag2 *pst1,
11              struct tag3 *pst2)
12 {
13     pst1->m1 = 2;
14     pst2->m1 = 0; /* alias? */
15
16     return pst1->m1;
17 }

```

An optimizing translator might produce **optimized** as the output of the program, while the same translator with optimization turned off might produce **unoptimized** as the output. This is because translation unit `y.c` defines `func` with two parameters each as pointers to different structures, and translation unit `x.c` calls `WG14_N685func` but passes the address of the same structure for each argument.

### Other Languages

Very few other languages make use of the concept of tags.

### Coding Guidelines

While the spelling of tag names might be thought not to alter the behavior of a program, the preceding example shows how a translator might rely on the information provided by tag names to make optimization decisions. While translators that take such information into account are rare, at the time of this writing, the commercial pressure to increase the quality of generated machine code means that such translators are likely to become more common. The guideline recommendation dealing with a single point of declarations is applicable here.

Another condition is where two unrelated structure or union types, in different translation units, use the same tag name. Identifiers in the tag name space have internal linkage and the usage may occur in a strictly conforming program. However, the issue is not language conformance but likelihood of developer confusion. The guideline recommendation dealing with reusing identifier names is applicable here.

### Example

```

1  extern struct S {                                file_1.c
2      int m1;
3      } x;

1  extern struct S {                                file_2.c
2      int m1;
3      } x;

1  extern struct T {                                file_3.c
2      int m1;
3      } x;

```

The declarations of `x` in `file_1.c` and `file_2.c` are compatible. The declaration in `file_3.c` is not compatible.

If both are complete types, then the following additional requirements apply:

635

### Commentary

The only thing that can be said about the case where one type is complete and the other is incomplete is that their tag names match. There is no danger of mismatch of member names, types, or relative ordering with a structure if the type is incomplete. If both are complete, there is more information available to be compared.

Except in one case, a pointer to an incomplete structure type, a type used in the declaration of an object or function must be completed by the end of the translation unit that contains it.

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. 636

## Commentary

For structure types, this requirement ensures that the same storage location is always interpreted in the same way when accessed through a specific member. It is also needed to ensure that the offsets of the storage allocated for successive members is identical for both types. For union types, this requirement ensures that the same named member is always interpreted in the same way when accessed.

What does *corresponding member* mean if there are no names to match against? The ordering rule is discussed elsewhere.

637 members  
corresponding

The standard does not give any explicit rule for breaking the recursion that can occur when checking member types for compatibility. For instance:

```

----- file_3.c -----
1 extern struct S4 {
2     struct S4 *mem;
3     } glob;

----- file_4.c -----
1 extern struct S4 {
2     struct S4 *mem;
3     } glob;

```

Enumeration constants always have an implied type of **int**.

enumeration  
constant  
type

The reply to DR #013 explains that the Committee feels the only reasonable solution is for the recursion to stop and the two types to be compatible.

## C90

*... if they have the same number of members, the same member names, and compatible member types;*

The C90 Standard is lax in that it does not specify any correspondence for members defined in different structure types, their names and associated types.

## C++

— each definition of *D* shall consist of the same sequence of tokens; and  
 — in each definition of *D*, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of *D*, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a **const** object with internal or no linkage if the object has the same integral or enumeration type in all definitions of *D*, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of *D*; and

3.2p5

The C Standard specifies an effect, compatible types. The C++ Standard specifies an algorithm, the same sequence of tokens (not preprocessing tokens), which has several effects. The following source files are strictly conforming C, but undefined behavior in C++.

```

----- file_1.c -----
1 extern struct {
2     short s_mem1;
3     } glob;

----- file_2.c -----
1 extern struct {
2     short int s_mem1;
3     } glob;

```

9.2p14 Two POD-struct (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types (3.9).

9.2p15 Two POD-union (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types (3.9).

Layout compatibility plays a role in interfacing C++ programs to other languages and involves types only. The names of members plays no part.

### Other Languages

Those languages that have a more sophisticated model of separate compilation usually ensure that textually, the same sequence of tokens is seen by all separately translated source files.

### Coding Guidelines

Using different names for corresponding members is unlikely to change the generated machine code. However, it could certainly be the cause of a great deal of developer confusion. The guideline recommendation specifying a single point of declaration is applicable here.

### Example

In the two files below the two declarations of x1 are compatible, but neither x2 or x3 are compatible across translation units.

```

----- file_1.c -----
1 extern struct S1 {
2     int m1;
3 } x1;
4 extern struct S2 {
5     int m1;
6 } x2;
7 extern struct S3 {
8     int m1;
9 } x3;

----- file_2.c -----
1 extern struct S1 {
2     int m1;
3 } x1;
4 extern struct S2 {
5     long m1;
6 } x2;
7 extern struct S3 {
8     int m_1;
9 } x3;

```

For two structures, corresponding members shall be declared in the same order.

### Commentary

On first reading this sentence appears to be back to front; shouldn't the order decide what the corresponding members are? In fact, the wording is a constructive definition of the term *corresponding*. The standard defines a set of requirements that must be met, not an algorithm for what to do. It is the developer's responsibility to

identifier ??  
declared in one file

members  
corresponding

ensure that the requirements are met. In the case of unnamed bit-fields, this ordering requirement means there is only one way of creating a correspondence that meets them.

These requirements ensure that members of the same structure type have the same offsets within different definitions (in different translation units) of that structure type. There is no corresponding requirement on union types because all members start at the same address offset. The order in which union members are declared makes no difference to their member storage layout.

pointer  
to union  
members  
compare equal  
storage  
layout

Enumeration constants may also be declared in different orders within their respective enumerated type definitions.

## C90

*... for two structures, the members shall be in the same order;*

The C90 Standard is lax in that it does not specify how a correspondence is formed between members defined in different structure definitions. The following two source files could have been part of a strictly conforming program in C90. In C99 the behavior is undefined and, if the output depends on `glob`, the program will not be strictly conforming.

```

_____ file_1.c _____
1 extern struct {
2     short s_mem1;
3     int i_mem2;
4     } glob;

_____ file_2.c _____
1 extern struct {
2     int i_mem2;
3     short s_mem1;
4     } glob;

```

While the C90 Standard did not require an ordering of corresponding member names, developer expectations do. A diagnostic, issued by a C99 translator, for a declaration of the same object as a structure type with differing member orders, is likely to be welcomed by developers.

### Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here. For each structure type, its member names are in their own unique name space. There is never an issue of other structure types, containing members using the same set of names, influencing other types.

?? identifier  
declared in one file

### Example

In the following example none of the members are not declared in the same order; however, it only matters in the case of structures.

```

_____ file_1.c _____
1 extern struct S1 {
2     int m1;
3     char m2;
4     } x;
5 extern union U1 {
6     int m1;
7     char m2;
8     } y;
9
10 extern enum {E1, E2} z;

```

---

```

1  extern struct S2 {
2      char m2;
3      int m1;
4      } x;
5  extern union U1 {
6      char m2;
7      int m1;
8      } y;
9
10 extern enum {E2=1, E1=0} z;

```

---

For two structures or unions, corresponding bit-fields shall have the same widths.

638

### Commentary

common initial sequence

The common initial sequence requirement ensures that the decision on where to allocate bit-fields within a storage unit is deterministic. If corresponding bit-fields have the same width they will be placed in the same location, relative to other members, in both structure types. In the case of union types a bit-field can occur in any position within the lowest address storage unit. Requiring that the members have the same width ensures that implementations assign them the same bit offsets within the storage unit.

### Common Implementations

bit-field shall have type

Implementations assign locations within storage units to bit-fields based on their width (and type if non-**int** bit-fields are supported as an extension). These factors are dealt with by these type compatibility requirements.

### Coding Guidelines

identifier ?? declared in one file

The guideline recommendation specifying a single point of declaration is applicable here.

---

For two enumerations, corresponding members shall have the same values.

639

### Commentary

The only requirement is on the value, not on how that value was specified in the source. For instance, it could have been obtained through a direct assignment of any number of constant expressions (all returning the same value), or implicitly calculated to be one greater than the previous enumeration constant value.

### C90

*... for two enumerations, the members shall have the same values.*

The C90 Standard is lax in not explicitly specifying that the members with the same names have the same values.

### C++

3.2p5 — *each definition of D shall consist of the same sequence of tokens; and*

The C++ requirement is stricter than C. In the following two translation units, the object `e_glob` are not considered compatible in C++:

---

```

1  extern enum {A = 1, B = 2} e_glob;

```

---

```

_____ file_2.c _____
1 extern enum {B= 2, A = 1} e_glob;

```

### Common Implementations

The debugging information written out to the object file in some implementations uses an implied enumeration constant ordering (i.e., that given in the definition). This can cause some inconsistencies in the display of enumeration constants, but debuggers are outside the scope of the standard.

### Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here.

?? identifier  
declared in one file

### Example

In the following two files members having the same name appear in the same order, but their values are different.

```

_____ file_1.c _____
1 extern enum {E1, E2} z;

```

```

_____ file_2.c _____
1 extern enum {E1 = 99, E2} z;

```

640 All declarations that refer to the same object or function shall have compatible type;

### Commentary

This *shall* is not in a constraint, making it undefined behavior if different declarations of the same object are not compatible types. There can be more than one declaration referring to the same object or function if

same object  
have com-  
patible types  
same function  
have com-  
patible types

- they occur in different translation units, in which case they will have external linkage;
- they occur in the same translation unit with internal linkage, or external linkage.

It is not possible to have multiple declarations of identifiers that have no linkage. The case:

```

_____ file_1.c _____
1 extern int i;

```

```

_____ file_2.c _____
1 extern enum {e1, e2} i;

```

no linkage  
identifier declara-  
tion is unique

(assuming that this enumerated type is compatible with the type `int`) is something of an oddity.

### C++

— each definition of *D* shall consist of the same sequence of tokens; and

3.2p5

— in each definition of *D*, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of *D*, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a **const** object with internal or no linkage if the object has the same integral or enumeration type in all definitions of *D*, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of *D*; and

*After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4).*

The C++ Standard is much stricter in requiring that the types be identical. The **int/enum** example given above would not be considered compatible in C++. If translated and linked with each other the following source files are strictly conforming C, but undefined behavior in C++.

```

_____ file_1.c _____
1  extern short es;

_____ file_2.c _____
1  extern short int es = 2;

```

### Other Languages

Languages that have a type system usually require that declarations of the same object or function, in separately translated source files, have compatible (however that term is defined) types.

### Common Implementations

It is very rare for an implementation to do cross translation unit type compatibility checking of object and function declarations. C++ implementations use some form of name mangling to resolve overloaded functions (based on the parameter types). Linkers supporting C++ usually contain some cross translation unit checks on function types. If C source code is translated by a C++ translator running in a C compatibility mode, many of these link-time checks often continue to be made.

### Coding Guidelines

The guideline recommendation specifying a single point of declaration is applicable here. Within existing code, old-style function declarations often still appear in headers. Such usage does provide a single declaration. However, these declarations do not provide all of the type information that it is possible to made available. For economic reasons developers may choose to leave these existing declarations unchanged.

### Example

Most implementations will translate and create a program image of the following two files without issuing any diagnostic:

```

_____ file_1.c _____
1  extern int f;
2
3  int main(void)
4  {
5  f++;
6  return 0;
7  }

_____ file_2.c _____
1  extern int f(void)
2  {
3  return 0;
4  }

```

otherwise, the behavior is undefined.

### Commentary

The most common behavior is to create a program image without issuing a diagnostic. The execution-time behavior of such a program image is very unpredictable.

### C++

*A violation of this rule on type identity does not require a diagnostic.*

3.5p10

The C++ Standard bows to the practical difficulties associated with requiring implementations to issue a diagnostic for this violation.

### Other Languages

Some languages require that a diagnostic be issued for this situation. Others say nothing about how their translators should behave in this situation.

### Common Implementations

The linkers in most implementations simply resolve names without getting involved in what those names refer to. In many implementations storage for all static duration objects is aligned on a common boundary (e.g., a two- or eight-byte boundary). This can mean, for scalar types, that there is sufficient storage available, whatever different types the same object is declared with. (Linkers usually obtain information on the number of bytes in an object from the object-file containing its definition, ignoring any such information that might be provided in the object-files containing declarations of it.) The effect of this link-time storage organization strategy is that having the same object declared with different scalar types, in different translation units, may not result in any unexpected behaviors (other objects being modified as a side-effect of assigning to the object) occurring.

642 *A composite type* can be constructed from two types that are compatible;

composite type

### Commentary

This C paragraph defines the term *composite type*; which are rarely talked about outside of C Standards-related discussions. Composite types arise because C allows multiple declarations, in some circumstances, of the same object or function. Two declarations of the same identifier only need be compatible; neither are they required to be token-for-token identical. In practice composite types are applied to types occurring in the same translation unit. There is no construction of composite types across translation units.

### C++

One of the two types involved in creating composite types in C is not supported in C++ (function types that don't include prototypes) and the C++ specification for the other type (arrays) is completely different from C.

644 [array](#)  
composite type

Because C++ supports operator overloading type qualification of pointed-to types is a more pervasive issue than in C (where it only has to be handled for the conditional operator). The C++ Standard defines the concept of a *composite pointer type* (5.9p2). This specifies how a result type is constructed from pointers to qualified types, and the null pointer constant and other pointer types.

[conditional operator](#)  
pointer to qualified types

### Other Languages

In most languages two types can only be compatible if they are token-for-token identical, so the need for a composite type does not usually occur.

### Coding Guidelines

If the guideline recommendation specifying a single point of declarations is followed, a composite type will only need to be created in one case— in the translation unit that defines an object, or function, having external linkage.

?? [identifier](#)  
declared in one file

**Example**

Assume the type `int` is the type that all enumerated types are compatible with:

```
1 enum ET {r, w, b};
2
3 extern enum ET obj;
4 extern int obj;
```

Possible composite type are the enumeration, ET, or the basic type `int`.

---

it is a type that is compatible with both of the two types and satisfies the following conditions:

643

**Commentary**

A composite type is formed by taking all the available information from both type declarations to create a type that maximizes what is known. The specification in the standard lists the properties of the composite type in relation to the two types from which it is created. It is not a set of rules to follow in constructing such a type. The possible composite types meeting the specification is not always unique.

---

— If one type is an array of known constant size, the composite type is an array of that size;

644

**Commentary**

For them to be compatible, the other array would have to be an incomplete array type. In this case encountering a declaration of known constant size also completes the type.

**C90**

*If one type is an array of known size, the composite type is an array of that size;*

Support for arrays declared using a nonconstant size is new in C99.

**C++**

An incomplete array type can be completed. But the completed type is not called the composite type, and is regarded as a different type:

<sup>3.9p7</sup> *... ; the array types at those two points (“array of unknown bound of T” and “array of N T”) are different types.*

The C++ Standard recognizes the practice of an object being declared with both complete and incomplete array types with the following exception:

<sup>3.5p10</sup> *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4).*

---

otherwise, if one type is a variable length array, the composite type is that type.

645

**Commentary**

This situation can only occur if the array type appears in function prototype scope. In this scope a VLA type is treated as if the size was replaced by `*`. The case of one of the array types having a constant size is covered by the previous rule; the remaining possibilities are another VLA type or an incomplete array type. In both cases a VLA type will be compatible with them, providing their element types are compatible.

VLA  
size treated as \*

array type  
to be compatible

**C90**

Support for VLA types is new in C99.

**C++**

Variable length array types are new in C99. The C++ library defined container classes (23), but this is a very different implementation concept.

**Example**

The composite type of:

```
1 extern int i(int m, int p4[m]);
2 extern int i(int n, int p4[n]);
```

is:

```
1 extern int i(int m, int p4[*]); /* Parameter names are irrelevant. */
```

---

646— 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.

function  
composite type

**Commentary**

The other type could be an old-style function declaration. If the other type is also a function prototype, the compatibility requirements mean that additional information can only be provided if the parameters have a pointer-to function type (which is an old-style function declaration). This issue is discussed in more detail elsewhere.

parameter  
qualifier in  
composite type

**C++**

All C++ functions must be declared using prototypes. A program that contains a function declaration that does not include parameter information is assumed to take no parameters.

```
1 extern void f();
2
3 void g(void)
4 {
5 f(); // Refers to a function returning int and having no parameters
6     /* Non-prototype function referenced */
7 }
8
9 void f(int p) /* Composite type formed, call in g linked to here */
10             // A different function from int f()
11             // Call in g does not refer to this function
12 { /* ... */ }
```

**Other Languages**

Most languages either require that the types of parameters be given or that the types not be given. C is unusual in allowing both kinds of function declarations. Both C++ and Java require that the parameter types always be given.

---

647— 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.

**Commentary**

This is a recursive definition. If a parameter has pointer-to function type, then a composite type will be constructed for its parameters, and so on. Wording elsewhere in the standard specifies that the composite type of parameters is the unqualified version of their types.

parameter  
qualifier in  
composite type

**C++**

C++ allows functions to be overloaded based on their parameter types. An implementation must not form a composite type, even when the types might be viewed by a C programmer as having the same effect:

```

1  /*
2  * A common, sloppy, coding practice. Don't declare
3  * the prototype to take enums, just use int.
4  */
5  extern void f(int);
6
7  enum ET {E1, E2, E3};
8
9  void f(enum ET p) /* composite type formed, call in g linked to here */
10                 // A different function from void f(int)
11                 // Call in g does not refer here
12 { /* ... */ }
13
14 void g(void)
15 {
16 f(E1); // Refers to a function void (int)
17       /* Refers to definition of f above */
18 }
```

**Example**

It is thus possible to slowly build up a picture of what the final function prototype is:

```

1  void f();
2  void f(int p1[], const int p2, float *      p3);
3  void f(int p1[2],      int p2, float *      p3);
4  /*
5  * Composite type is: void f(int p1[2], int p2, float *p3);
6  */
7  void f(int p1[], const int p2, float * volatile p3);
8  /*
9  * Composite type is unchanged.
10 */
```

---

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

648

**Commentary**

The possible types, not already covered before, to which these rules can be applied are array, structure, union, and pointer. Although it is not possible to declare an array of incomplete type, an array of pointer-to functions would create a recursive declaration that would need to be processed by these rules.

**C++**

The C++ Standard has no such rules to apply recursively.

**Other Languages**

Languages that support nesting of derived types usually apply any applicable rules recursively.

## Commentary

Any change to an object's type, to a composite type, will not affect the way that accesses to it are handled. Any operation that accesses an object after its first declaration, but before a second declaration, will behave the same way after any subsequent declaration is encountered.

It is possible for references to an identifier, having a function type, to be different after a subsequent declaration. This occurs if the type of a parameter is not compatible with its promoted type (the issue of the composite type of function types is also discussed elsewhere). For instance:

integer promotions  
parameter  
qualifier in  
composite type

```

1  extern void f();
2
3  void g_1(void)
4  {
5  f('w');
6  }
7
8  extern void f(char);
9
10 void g_2(void)
11 {
12 f('x');
13 }
```

the first call to `f` uses the information provided by the first, old-style, declaration of `f`. At the point of this first call, the behavior is undefined (assuming the function definition is defined using a prototype). At the point of the second call to `f`, in `g_2`, the composite type matches the definition and the behavior is defined.

function  
definition  
ends with ellipsis

## C90

The wording in the C90 Standard:

*For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.*

was changed to its current form by the response to DR #011, question 1.

## C++

*Two names that are the same (clause 3) and that are declared in different scopes shall denote the same object, reference, function, type, enumerator, template or namespace if*

3.5p9

- both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and
- both names refer to members of the same namespace or to members, not by inheritance, of the same class; and
- when both names denote functions, the function types are identical for purposes of overloading; and

This paragraph applies to names declared in different scopes; for instance, file scope and block scope externals.

*When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only function declarations can be overloaded; object and type declarations cannot be overloaded.*

13p1

The following C++ requirement is much stricter than C. The types must be the same, which removes the need to create a composite type.

*After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.*

The only composite type in C++ are composite pointer types (5.9p2). These are only used in relational operators (5.9p2), equality operators (5.10p2, where the term common type is used), and the conditional operator (5.16p6). C++ composite pointer types apply to the null pointer and possibly qualified pointers to **void**.

If declarations of the same function do not have the same type, the C++ link-time behavior will be undefined. Each function declaration involving different adjusted types will be regarded as referring to a different function.

```

1  extern void f(const int);
2  extern void f(int);          /* Conforming C, composite type formed */
3                               // A second (and different) overloaded declaration

```

### Example

The following illustrates various possibilities involving the creation of composite types (the behavior is undefined because the types are not compatible with each other).

```

1  extern int a[];
2
3  void f(void)
4  {
5  extern int a[3]; /* Composite type applies inside this block only. */
6  }
7
8  void g(void)
9  {
10 extern int a[4]; /*
11                * Compatible with file scope a[],
12                * composite type applies to this block.
13                */
14 }
15
16 void h(void)
17 {
18 float a;
19 {
20 extern int a[5]; /* No prior declaration visible. */
21 }
22 }
23
24 extern int a[5]; /* Composite type is int a[5]. */

```

46) Two types need not be identical to be compatible.

650

### Commentary

Neither do two types have to consist of the same sequence of tokens to be the same types. It is possible for different sequences of tokens to represent the same type. The standard does not specify what it means for two types to be identical.

**C++**

The term *compatible* is used in the C++ Standard for *layout-compatible* and *reference-compatible*. Layout compatibility is aimed at cross-language sharing of data structures and involves types only. The names of structure and union members, or tags, need not be identical. C++ reference types are not available in C.

**Other Languages**

Languages take different approaches to the concept of compatibility. Pascal and many languages influenced by it use name equivalence (every type is unique and denoted by its name). Other languages, for instance CHILL, support *structural equivalence* (two types are the same if the underlying representation is the same, no matter how the declarations appear in the text). In fact CHILL specifies a number of different kinds of equivalence, which are used in different contexts.

structural  
equivalence

```

1  /*
2  * Some languages specify that the first two definitions of X are not
3  * compatible, because the structure of their declarations is different.
4  * Languages that do not expose the internal representation of a type
5  * could regard the last definition of X as being compatible with the
6  * previous two. There are matching members and their types are the same.
7  */
8  struct X {
9      int a,
10     b;
11 };
12 struct X {
13     int a;
14     int b;
15 };
16 struct X {
17     int b;
18     int a;
19 };
20
21 /*
22 * From the structural point of view the following
23 * declarations are all different.
24 */
25 extern unsigned int ui_1;
26 extern unsigned int ui_2;
27 extern int unsigned int ui_3;

```

Some languages (e.g., Ada, Java, and Pascal) base type equivalence on names, hence this form is called *name equivalence*. The extent to which name equivalence is enforced for arithmetic types varies between languages. Some languages are strict, while others have a more relaxed approach (Ada offers the ability to specify which approach to use in the type declaration).

```

1  typedef unsigned int UINT_1;
2  typedef unsigned int UINT_2;
3
4  /*
5  * Some languages would specify that the following objects were not
6  * compatible because they were declared with types that had different
7  * names (an anonymous one in the last case). Others would be more
8  * relaxed, because the underlying types were the same.
9  */
10 extern UINT_1 ui_1;
11 extern UINT_2 ui_2;
12 extern unsigned int ui_3;
13
14 /*
15 * All languages based on name equivalence would treat the following

```

```

16  * as different types that were not compatible with each other.
17  */
18  typedef struct {
19          int x, y;
20      } coordinate;
21  typedef struct {
22          int x, y;
23      } position;
24  typedef int APPLES;
25  typedef int ORANGES;

```

### Coding Guidelines

Most coding guideline documents recommend a stricter interpretation of *identical* types than that made by the C Standard.

type com-631  
patibility  
general guideline

---

47) As specified in 6.2.1, the later declaration might hide the prior declaration. 651

### Commentary

To be exact a different declaration, in a scope between that of the file scope declaration and the current declaration, might hide the later declaration.

---

EXAMPLE Given the following two file scope declarations: 652

```

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

```

The resulting composite type for the function is:

```

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

```

### C++

The C++ language supports the overloading of functions. They are overloaded by having more than one declaration of a function with the same name and return type, but different parameter types. In C++ the two declarations in the example refer to different versions of the function `f`.

footnote  
47

outer scope  
identifier hidden

# References

Ltd, 1992.

1. D. M. Jones. The Model C Implementation. Knowledge Software