The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.2.1 Structure and union specifiers

struct/union syntax

struct-or-union-specifier:
struct-or-union *identifier*$_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union *identifier*
struct-or-union:
struct
union
struct-declaration-list:
struct-declaration
struct-declaration-list *struct-declaration*
struct-declaration:
specifier-qualifier-list *struct-declarator-list* **;**
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{opt}$
type-qualifier *specifier-qualifier-list*$_{opt}$
struct-declarator-list:
struct-declarator
struct-declarator-list **,** *struct-declarator*
struct-declarator:
declarator
declarator$_{opt}$ **:** *constant-expression*

Commentary

The use of **{** and **}** to delimit the sequence of members is consistent with them being used to delimit the sequence of statements in a compound statement.

C++

The C++ Standard uses the general term *class* to refer to these constructs. This usage is also reflected in naming of the nonterminals in the C++ syntax. The production *struct-or-union* is known as *class-key* in C++ and also includes the keyword **class**. The form that omits the brace enclosed list of members is known as an *elaborated-type-specifier* (7.1.5.3) in C++.

Other Languages

Some languages (e.g., Pascal) use the keyword **record** to denote both types and the keyword **end** to denote the end of the declaration (which is consistent with the delimiters used in these languages for compound statements, e.g., **begin/end**). The following example shows how Ada permits developers to specify which word, and bits within a word, are occupied by a member.

```
1  WORD : constant := 4;
2
3  type STATUS_WORD is
4     record
5     system_mask : array(0..7) of BOOLEAN;
6     key         : INTEGER range 0..3;
7     inst_addr   : ADDRESS;
8     end record;
9
10 for STATUS_WORD use
11    record at mod 8  -- address at which object is allocated storage
12    system_mask at 0*WORD range  0.. 7; -- occupies bits 0 through 7, inclusive
13    key         at 0*WORD range 10..11; -- occupies 2 bits
14    inst_addr   at 1*WORD range  8..31; -- allocate member in second word
15    end record;
```

The following is an example of a record declaration in Cobol.

```
        01 WEEKLY-SALES
           05 SALES-TABLE          OCCURS 50 TIMES.
              10 PRODUCT-CODE         PIC X(10).
              10 PRODUCT-PRICE        PIC S9(4)V99.
              10 PRODUCT-COUNT        PIC XXX.
                 88 LAST_PRODUCT                  VALUE "999".
```

Common Implementations

Some translators (gcc, and Plan 9 C[9]) support the concept of anonymous structure and union members. In the example below the member name of a union type is omitted. When resolving names along a chain of selections a translator has to deduce when a member of a union is intended.

```
struct {
       int mem1;
       union {
              char mem2;
              long mem3;
             }; /* anonymous */
       } x;

void f(void)
{
x.mem3=3;
}
```

Cyclone C[3] supports tagged union types, using the keyword **tunion**, which contain information that identifies the current member.

Coding Guidelines

The visibility issues associated with object identifiers declared in an *init-declarator* also apply to identifiers declared in a *struct-declarator*.

?? init-declarator one per source line

Cg 1390.1 No more than one *struct-declarator* shall occur on each visible source code line.

The discussion on the layout of declarators also applies to declarators in structure definitions. Deciding which members belong in which structure type can involve trade-offs amongst many factors. This issue is discussed in more detail elsewhere.

declarator list of

limit struct/union nesting

Usage

A study by Sweeney and Tip[8] of C++ applications found that on average 11.6% of members were dead (i.e., were not read from) and that 4.4% of object storage space was occupied by these dead data members. Usage information on member names and their types is given elsewhere (see Table **??** and Table **??**).

Table 1390.1: Number of occurrences of the given token sequence. Based on the visible source of the .c files (.h files in parentheses).

Token Sequence	Occurrences	Token Sequence	Occurrences
enum {	456 (1,591)	**struct** id **;**	76 (13,384)
enum id **;**	0 (0)	**struct** id id	122,974 (27,589)
enum id **{**	474 (1,059)	**union {**	297 (725)
enum id id	2,922 (633)	**union** id **;**	0 (11)
struct {	1,567 (6,503)	**union** id **{**	105 (2,624)
struct id **{**	4,407 (1,311)	**union** id id	330 (231)

Constraints

member not types

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; 1391

Commentary

Allowing structure or union members to have an incomplete type creates complications (the types would have to be completed at some point and a mechanism would need to be created to perform it) for little benefit. Requiring that the members have a complete type also simplifies the job of the translator. It becomes be possible to assign offsets, relative to the first member, to each member as it is encountered. The exception is for the last member where there is no following member requiring an offset. The requirement on there being at least one other named member, when the last member has an incomplete type, arises because of how the size of the declared type is calculated.

flexible array member 1430

structure size with flexible member 1432

The intent of the exception case is to support an implementation model where the storage for the last member follows the storage allocated to the other members in the same type (i.e., the storage for the last member does not need to be allocated in a separate storage area, with references to the member being implicitly dereferenced), but the amount allocated is decided during program execution.

Members having function type are supported in object-oriented languages. Although a proposal was submitted to the C committee, WG14/{N424, N445, N446, N447}, to add classes to C99 the additional complexity was not considered to be in the spirit of C.

spirit of C

C90

Support for the exception on the last named member is new in C99.

C++

It is a design feature of C++ that class types can contain incomplete and function types. Source containing instances of such constructs is making use of significant features of C++ and there is unlikely to be any expectation of being able to successfully process it using a C translator.

The exception on the last named member is new in C99 and this usage is not supported in the C++ Standard.

The following describes a restriction in C++ that does not apply in C.

annex C.1.7p3

Change: *In C++, a* **`typedef`** *name may not be redefined in a class declaration after being used in the declaration*

Example:

```
typedef int I;
struct S {
          I i;
          int I;  // valid C, invalid C++
         };
```

Rationale: *When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.*

Other Languages

Other languages follow C in requiring members to be declared with complete types. Object-oriented language allow class definitions to include function definitions.

Example

An awkward edge case that needs to be handled by any implementation claiming to support incomplete member types.

```
1  struct x { struct y y; }; /* Constraint violation in Standard C. */
2  struct y { struct x x; };
```

1392 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.

Commentary

This requirement applies to the exception case and it describes contexts that all require types of known size. Requiring support for this usage would require the array storage for the last member to be allocated somewhere other than after the storage for the member that preceded it.

1393 The expression that specifies the width of a bit-field shall be an integer constant expression that has a nonnegative value that shall not exceed the ~~number~~width of ~~bits in~~ an object of the type that ~~is~~would be specified ~~if~~were the colon and expression ~~are~~ omitted.

bit-field maximum width

Commentary

The intent of a bit-field declaration is to specify the number of bits in a types value representation, a negative value has no meaning. Objects having a bit-field type can be used wherever expressions having certain integer types can be used. This usage can only be guaranteed to be defined if bit-fields do not have a width greater than these types. Implementations are required to supported a width of at least 1 for bit-fields of type **_Bool**. However, they may also support greater widths, up to a maximum of CHAR_BIT.

bit-field value is m bits

expression wherever an int may be used

_Bool rank

The wording was changed by the response to DR #262 (making it clear that any padding bits are not counted).

C90

The C90 wording ended with " . . . of bits in an ordinary object of compatible type.", which begs the question of whether bit-fields are variants of integer types or are separate types.

C++

The C++ issues are discussed elsewhere.

bit-field value is m bits

Coding Guidelines

Specifying a width equal to the number of bits that would have been used in the value representation, had the colon and expression been omitted, would appear to be redundant. However, the source may be translated using a variety of different implementations in different host environments and the equality in the number of bits used may not apply to all of them. Also, when declared using plain **int** the signedness of a bit-fields type will be the same as other bit-fields declared using plain **int**. How the width is specified (through the replacement of a macro name, or a constant expression) ties in with the general software engineering topic of source configuration and is outside the scope of these coding guidelines.

bit-field int

1394 If the value is zero, the declaration shall have no declarator.

Commentary

This construct is discussed elsewhere.

1415 bit-field zero width

C++

Only when declaring an unnamed bit-field may the `constant-expression` *be a value equal to zero.* 9.6p2

Source developed using a C++ translator may contain a declaration of a zero width bit-field that include a declarator, which will generate a constraint violation if processed by a C translator.

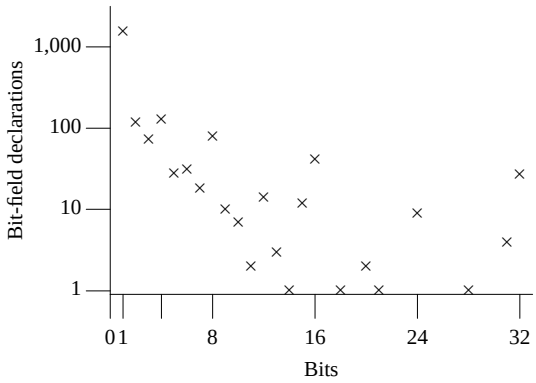

Figure 1393.1: Number of bit-field declarations specifying a given number of bits. Based on the translated form of this book's benchmark programs. (Declarations encountered in any source or header file were only counted once, the contents of system headers were ignored.)

```
1  struct {
2           int mem_1;
3           unsigned int mem_2:0; //                    no diagnostic required
4                                 /* constraint violation, diagnostic required */
5         } obj;
```

There is an open C++ DR (#057) concerning the lack of a prohibition against declarations of the form:

```
1  union {int : 0;} x;
```

---

bit-field shall have type

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. 1395

Commentary

In this context use of the type specifier **int** is equivalent to either **signed int** or **unsigned int**. It is not a different type (as **char** is from **signed char** and **unsigned char**) and so need not appear in the list here.

standard integer types, extended integer types

The phrase *implementation-defined type* refers to any other standard integer type that an implementation chooses to support in a bit-field declaration, or any an implementation-defined extended integer type. Using " . . . some other implementation-defined type." is making use of implementation-defined behavior.

C90

The following wording appeared in a semantics clause in C90, not a constraint clause.

A bit-field shall have a type that is a qualified or unqualified version of one of ***int****,* ***unsigned int****, or* ***signed int****.*

Programs that used other types in the declaration of a bit-field exhibited undefined behavior in C90. Such programs exhibit implementation-defined behavior in C99.

C++

9.6p3 *A bit-field shall have integral or enumeration type (3.9.1).*

Source developed using a C++ translator may contain bit-fields declared using types that are a constraint violation if processed by a C translator.

```
1  enum E_TAG {a, b};
2
3  struct {
4          char m_1  : 3;
5          short m_2 : 5;
6          long m_3  : 7;
7          enum E_TAG m_4 : 9;
8         } glob;
```

Other Languages

Languages that provide a mechanism for specifying the layout of aggregate types (e.g., Ada and CHILL) do not usually place restrictions on the types that the objects may have. Although they may limit the range of possible widths for some types.

Common Implementations

Some implementations support bit-fields declared using other integer types (`gcc` and the SVR4 C compiler). This usage is not purely to intended to support value representations containing a greater number of bits. Implementations often use the alignment of the type specifier used as the alignment of the addressable storage unit to be used to hold the bit-field. For instance, if the alignment used for the type **`short`** was 2 and that for **`int`** 4, then the alignment of the addressable storage unit used to hold bit-fields declared using the two types would be 2 and 4 respectively.

While some processors (e.g., Intel Pentium) include instructions for operating on what are sometimes called *packed floating-point* values, the representation used for these *packed* values is the same as that used for their *unpacked* form. The term *packed* being applied in the case where instructions operate on two or more floating-point values as a single unit (e.g., four 32-bit values, represented in 128 bits, are added to another four 32-bit values being represented in 128 bits).

Coding Guidelines

The discussion leading to the guideline recommendation dealing with the use of a single integer type is applicable here.

?? object
int type only

Use of implementation-defined integer types, whose rank is less than that of **`int`**, almost certainly implies that a developer is trying to control the layout of objects in storage. While there is a guideline recommendation covering this usage, bit-fields are often treated as a special case. There are sometimes application domain storage layout requirements. For instance, interpreting different sequences of bits at various bit offsets, from some starting point, as corresponding to different value representations.

storage layout

?? extensions
cost/benefit

An alternative to declaring members as bit-field types is to declare them as non-bit-field integer types and use bitwise operations to extract the required sequence of value bits. This approach removes any dependency on how implementations allocate bit-field members to storage locations. The costs are a potential increase in effort needed to comprehend the source (unless the details of the bit sequence selection are hidden behind a macro call) and a potential increase in execution time (it is likely to be easier to optimize the generated machine code for an access using a member selection operator than a sequence of developer selected bitwise operations). Provided macros are used to hide the implementation details, neither of these costs is likely to be significant.

A developer may want to minimize the amount of storage used by a type and have no interest in the actual storage layout selected. Recommending against the use of other implementation-defined type because storage minimization is not guaranteed on other implementations is a rather rigid interpretation of guidelines.

storage layout

Dev ??

A bit-field may be declared using a type whose rank is less than that of **`int`** provided no use is made of member layout information.

Dev ??

A bit-field may be declared using a type whose rank is greater than that of **`int`**.

Semantics

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. 1396

Commentary

structure type
sequentially allocated objects
union type
overlapping members

These issues are discussed elsewhere.

C++

This requirement can be deduced from 9.2p12 and 9.5p1.

Structure and union specifiers have the same form. 1397

Commentary

types

They have the same syntactic form (the only syntactic difference is the keyword used). As types both structures and unions are either identified by their tag name, a typedef name, or are anonymous, within individual translation units (their members are not used to determine type compatibility). However, between different translation units, type compatibility includes requirements on the members in a structure or union type.

footnote 46

compatible
separate translation units

C++

The C++ Standard does not make this observation.

Other Languages

In some languages (e.g., Pascal and Ada) a variant record (union) can only occur within a record (structure) declaration. It is not possible to declare the equivalent of a union at the outer most declaration level.

The keywords **`struct`** and **`union`** indicate that the type being specified is, respectively, a structure type or a union type. 1398

Commentary

tag name
same struct, union or enum

This sentence was added by the response to DR #251. The reason for it is discussed elsewhere.

The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. 1399

Commentary

This wording specifies that the form: *struct-or-union identifier*$_{opt}$ **`{`** *struct-declaration-list* **`}`** declares a new type. Other forms of structure declaration that omit the braces either declare an identifier as a tag or refer to a previous declaration.

Other Languages

Whether or not a structure or union type definition is a new type may depend on a languages type compatibility rules. Languages that use structural equivalence may treat different definitions as being the same type (usually employing rules similar to those used by C for type compatibility across translation units).

compatible
separate translation units

The struct-declaration-list is a sequence of declarations for the members of the structure or union. 1400

Commentary

Say in words what is specified in the syntax.

If the struct-declaration-list contains no named members, the behavior is undefined. 1401

Commentary

The syntax does not permit the *struct-declaration-list* to be empty. However, it is possible for members to be unnamed bit-fields.

1414 bit-field unnamed

C++

An object of a class consists of a (possibly empty) sequence of members and base class objects. 9p1

Source developed using a C++ translator may contain class types having no members. This usage will result in undefined behavior when processed by a C translator.

Other Languages

The syntax of languages invariably requires at least one member to be declared and do not permit zero sized types to be defined.

Common Implementations

Most implementations issue a diagnostic when they encounter a *struct-declaration-list* that does not contain any named members. However, many implementations also implicitly assume that all declared objects have a nonzero size and after issuing the diagnostic may behave unpredictably when this assumption is not met.

Coding Guidelines

A discussion of the cost/benefit of the use of such a construct (perhaps based on its apparent harmlessness versus possible unpredictable translator behavior). In practice occurrences of this construct are rare (until version 3.3.1 gcc reported "internal compiler error" for many uses of objects declared with such types) and is not worth a guideline recommendation.

Example

```
1  #include <stdio.h>
2
3  struct S {
4            int : 0;
5           };
6
7  void f(void)
8  {
9  struct S arr[10];
10
11 printf("arr contains %d elements\n", sizeof(arr)/sizeof(struct S));
12 }
```

1402 The type is incomplete until after the } that terminates the list.

struct type incomplete until

Commentary

This sentence is a special case of one discussed elsewhere.

tag incomplete until

Example

```
1  struct S {
2            int m1;
3            struct S m2; /* m2 refers to an incomplete type (a constraint violation). */
4           }             /* S is complete now. */;
```

```
5  struct T {
6            int m1;
7           } x = { sizeof(struct T) }; /* sizeof a completed type. */
```

In the second definition the closing **}** (the one before the x) completes the type and the **sizeof** operator can be applied to the type.

struct member type

A member of a structure or union may have any object type other than a variably modified type.[103)] 1403

Commentary

member 1391 not types

Other types are covered by a constraint. As the discussion for that C sentence points out, the intent is to enable a translator to assign storage offsets to members at translation time. Apart from the special case of the last member, the use of variably modified types would prevent a translator assigning offsets to members (because their size is not known at translation time).

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard.

Other Languages

Java uses references for all non-primitive types. Storage for members having such types need not be allocated in the class type that contains the member declaration and there is no requirement that the number of elements allocated to a member having array type be known at translation time.

Table 1403.1: Occurrence of structure member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
int	15.8	**unsigned short**	7.7	**char** *	2.3	**void** *()	1.3
other-types	12.7	**struct**	7.2	**enum**	1.9	**float**	1.2
unsigned char	11.1	**unsigned long**	5.2	**long**	1.8	**short**	1.0
unsigned int	10.4	**unsigned**	4.0	**char**	1.8	**int** *()	1.0
struct *	8.8	**unsigned char** []	3.1	**char** []	1.5		

Table 1403.2: Occurrence of union member types (as a percentage of the types of all such members). Based on the translated form of this book's benchmark programs.

Type	%	Type	%	Type	%	Type	%
struct	46.9	**unsigned int**	3.8	**double**	1.9	**char** []	1.3
other-types	11.3	**char** *	2.8	**enum**	1.7	**union** *	1.1
struct *	8.3	**unsigned long**	2.4	**unsigned char**	1.5		
int	6.0	**unsigned short**	2.1	**struct** []	1.3		
unsigned char []	4.3	**long**	2.1	**(struct** * **)** []	1.3		

In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). 1404

Commentary

The ability to declare an object that consists of a specified number of bits is only possible inside a structure or union type declaration.

Other Languages

Some languages (e.g., CHILL) provide a mechanism for specifying how the elements of arrays are laid out and the number of bits they occupy. Languages in the Pascal family support the concept of subranges. A subrange allows the developer to specify the minimum and maximum range of values that an object needs to be able to represent. The implementation is at liberty to allocate whatever resources are needed to satisfy this requirement (some implementations simply allocate an integers worth of storage, while others allocate the minimum number of bytes needed).

Coding Guidelines

Why would a developer want to specify the number of bits to be used in an object representation? This level of detail is usually considered to be a low level implementation information. The following are possible reasons for this usage include:

- *Minimizing the amount of storage used by structure objects.* This remains, and is likely to continue to remain, an important concern in applications where available storage is very limited (usually for cost reasons).
- *There is existing code*, originally designed to run in a limited storage environment. The fact that storage requirements are no longer an issue is rarely a cost effective rationale for spending resources on removing bit-field specifications from declarations.
- *Mapping to a hardware device.* There are often interfaced via particular storage locations (organized as sequences of bits), or transfer data is some packed format. Being able to mirror the bit sequences of the hardware using some structure type can be a useful abstraction (which can require the specification of the number of bits to be allocated to each object).
- *Mapping to some protocol imposed layout of bits.* For instance, the fields in a network data structure (e.g., TCP headers).

The following are some of the arguments that can be made for not using bit-fields types:

- Many of the potential problems associated with objects declared to have an integer type, whose rank is less than **int**, also apply to bit-fields. However, one difference between them is that developers do not habitually use bit-fields, to the extent that character types are used. If developers don't use bit-fields out of habit, but put some thought into deciding that their use is necessary a guideline recommendation would be redundant (treating guideline recommendations as prepacked decision aids).
- It is making use of representation information.
- The specification of bit-field types involves a relatively large number of implementation-defined behaviors, dealing with how bit-fields are allocated in storage. However, recommending against the use of bit-fields only prevents developers from using one of the available techniques for accessing sequences of bits within objects. It is not obvious that bit-fields offer the least cost/benefit of all the available techniques (although some coding guideline documents do recommend against the use of bit-fields).

?? object int type only
coding guidelines introduction
types representation

Dev ?? Bit-fields may be used to interface to some externally imposed storage layout requirements.

1405 Such a member is called a *bit-field*;[104)]

bit-field

Commentary

This defines the term *bit-field*. Common usage is for this term to denote bit-fields that are named. The less frequently used unnamed bit-fields being known as *unnamed bit-fields*.

1414 bit-field unnamed

Other Languages

Languages supporting such a type use a variety of different terms to describe such a member.

---

its width is preceded by a colon. 1406

Commentary

Specifying in words the interpretation to be given to the syntax.

Other Languages

Declarations in languages in the Pascal family require the range of values, that need to be representable, to be specified in the declaration. The number of bits used is implementation-defined.

---

bit-field interpreted as

A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.[105)] 1407

Commentary

value representation, object representation

Both the value and object representation use the same number of bits. In some cases there may be padding between bit-fields, but such padding cannot be said to belong to any particular member.

C++

The C++ Standard does not specify (9.6p1) that the specified number of bits is used for the value representation.

Coding Guidelines

symbolic name

Using a symbolic name to specify the width might reduce the effort needed to comprehend the source and reduce the cost making changes to the value in the future.

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. 1408

Commentary

standard unsigned integer, _Bool large enough to store 0 and 1

This is a requirement on the implementation. It is implied by the type **_Bool** being an unsigned integer type (for signed types a single bit bit-field can only hold the values 0 and -1). These are also the only two values that are guaranteed to be represented by the type **_Bool**.

C90

Support for the type **_Bool** is new in C99.

---

bit-field addressable storage unit

An implementation may allocate any addressable storage unit large enough to hold a bit-field. 1409

Commentary

There is no requirement on implementations to allocate the smallest possible storage unit. They may even allocate more bytes than `sizeof(int)`.

Other Languages

struct/union syntax 1390

Languages that support some form of object layout specification often require developers to specify the storage unit and the bit offset, within that unit, where the storage for an object starts.

Common Implementations

bit-field packed into 1410, bit-field shall have type 1395

Many implementations allocate the same storage unit for bit-fields as they do for the type **int**. The only difference being that they will often allocate storage for more than one bit-field in such storage units. Implementations that support bit-field types having a rank different from **int** usually base the properties of the storage unit used (e.g., alignment and size) on those of the type specifier used.

Coding Guidelines

bit-field shall have type 1395, representation information using ??

Like other integer types, the storage unit used to hold bit-field types is decided by the implementation. The applicable guidelines are the same.

Example

```
#include <stdio.h>

struct {
        char m_1;
        signed int m_2 :3;
        char m_3;
       } x;

void f(void)
{
if ((&x.m_3 - &x.m_1) == sizeof(int))
   printf("bit-fields probably use the same storage unit as int\n");
if ((&x.m_3 - &x.m_1) == 2*sizeof(int))
   printf("bit-fields probably use the same storage unit and alignment as int\n");
}
```

1410 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.

bit-field packed into

Commentary

This is a requirement on the implementation. However, any program written to verify what the implementation has done, has to make use of other implementation-defined behavior. This requirement does not guarantee that all adjacent bit-fields will be packed in any way. An implementation could choose its addressable storage unit to be a byte, limiting the number of bit-fields that it is required to pack. However, if the storage unit used by an implementation is a byte, this requirement means that all members in the following declaration must allocated storage in the same byte.

```
struct {
        int mem_1 : 5;
        int mem_2 : 1;
        int mem_3 : 2;
       } x;
```

C++

This requirement is not specified in the C++ Standard.

Allocation of bit-fields within a class object is implementation-defined. 9.6p1

1411 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.

bit-field overlaps storage unit

Commentary

One of the principles that the C committee derived from the spirit of C was that an operation should not expand to a surprisingly large amount of machine code. Reading a bit-field value is potentially three operations; load value, shift right, and zero any unnecessary significant bits. If implementations were required to allocate bit-fields across overlapping storage units, then accessing such bit-fields is likely to require at least twice as many instructions on processors having alignment restrictions. In this case it would be necessary to load values from the two storage units into two registers, followed by a sequence of shift, bitwise-AND, and bitwise-OR operations. This wording allows implementation vendors to chose whether they want to support this usage, or leave bits in the storage unit unused.

spirit of C

alignment

Other Languages

Even languages that contain explicit mechanisms for specifying storage layout sometimes allow implementations to place restrictions on how objects straddle storage unit boundaries.

Common Implementations

Implementations that do not have alignment restrictions can access the appropriate bytes in a single load or store instruction and do not usually include a special case to handle overlapping storage units. Some processors include instructions[6] that can load/store a particular sequence of bits from/to storage.

representation information using ??

Coding Guidelines

The guideline recommendation dealing with the use of representation information are applicable here.

Example

The extent to which any of the following members are put in the same storage unit is implementation-defined.

```
1  struct T {
2            signed int m_1 :5;
3            signed int m_2 :5; /* Straddles an 8-bit boundary. */
4            signed int m_3 :5;
5            signed int m_4 :5; /* Straddles a 16-bit boundary. */
6            signed int m_5 :5;
7            signed int m_6 :5;
8            signed int m_7 :5; /* Straddles a 32-bit boundary. */
9           };
```

The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. 1412

Commentary

An implementation is required to chose one of these two orderings The standard does not define an order for bits within a byte, or for bytes within multibyte objects. Either of these orderings is consistent with the relative order of members required by the Standard.

byte addressable unit
object contiguous sequence of bytes
member 1422 address increasing
unary & operand constraints

It is not possible to take the address of an object having a bit-field type, and so bit-field member ordering cannot be deduced using pointer comparisons. However, the ordering can be deduced using a union type.

Common Implementations

While there is no requirement that the ordering be the same for each sequence of bit-field declarations (within a structure type), it would be surprising if an implementation used a different ordering for different declarations. Many implementations use the allocation order implied by the order in which bytes are allocated within multibyte objects.

representation information using ??

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

Example

```
1  /*
2   * The member bf.m_1 might overlap the same storage as m_4[0] or m_4[1]
3   * (using a 16-bit storage unit).  It might also be the most significant
4   * or least significant byte of m_3 (using int as the storage unit).
5   */
6  union {
7         struct {
8                 signed int m_1 :8;
9                 signed int m_2 :8;
```

```
10              } bf;
11          int m_3;
12          char m_4[2];
13        } x;
```

1413 The alignment of the addressable storage unit is unspecified.

alignment addressable storage unit

Commentary

This behavior differs from that of the non-bit-field members, which is implementation-defined.

1421 member alignment

C++

The wording in the C++ Standard refers to the bit-field, not the addressable allocation unit in which it resides. Does this wording refer to the alignment within the addressable allocation unit?

Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. 9.6p1

Common Implementations

Implementations that support bit-field types having a rank different from **int** usually base the properties of the alignment used on those of the type specifier used.

1395 bit-field shall have type

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

1414 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[106)]

bit-field unnamed

Commentary

Memory mapped devices and packed data sometimes contains sequences of bits that have no meaning assigned to them (sometimes called *holes*). When creating a sequence of bit-fields that map onto the meaningful values any holes also need to be taken into account. Unnamed bit-fields remove the need to create an *anonymous* name (sometimes called a *dummy* name) to denote the bit sequences occupied by the holes. In some cases the design of a data structure might involve having some spare bits, between certain members, for future expansion.

Other Languages

Languages that support some form of layout specification usually use a more direct method of specifying where to place objects (using bit offset and width). It is not usually necessary to specify where the holes go.

Coding Guidelines

Any value denoted by the sequence of bits specified by an unnamed bit-field is not accessible to a conforming program. The usage is purely associated with specifying representation details. There is no minimization of storage usage justification and the guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

1415 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.

bit-field zero width

Commentary

This special case provides an additional, developer accessible, mechanism for controlling the layout of bit-fields in structure types (it has no meaningful semantics for members of union types). It might be thought that this special case is redundant, a developer either working out exactly what layout to use for a particular implementation or having no real control over what layout gets used in general. However, if an implementation supports the allocation of bit-fields across adjacent units a developer may be willing to trade less efficient use of storage for more efficient access to a bit-field. Use of a zero width bit-field allows this choice to be made.

1411 bit-field overlaps storage unit

footnote 103

103) A structure or union can not contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3. 1416

Commentary

variable modified only scope

It would have been possible for the C committee to specify that members could have a variably modified type. The reasons for not requiring such functionality are discussed elsewhere.

C90

Support for variably modified types is new in C99.

C++

Variably modified types are new in C99 and are not available in C++.

footnote 104

104) The unary & (address-of) operator cannot be applied to a bit-field object; 1417

Commentary

unary & operand constraints

Such an occurrence would be a constraint violation.

thus, there are no pointers to or arrays of bit-field objects. 1418

Commentary

bit-field 1395 shall have type

The syntax permits the declaration of such bit-fields and they are permitted as implementation-defined extensions. The syntax for declarations implies that the declaration:

```
1  struct {
2          signed int abits[32] : 1;
3          signed int *pbits : 3;
4          } vector;
```

bit-field 1395 shall have type

declares `abits` to have type array of bit-field, rather than being a bit-field of an array type (which would also violate a constraint). Similarly `pbits` has type pointer to bit-field.

spirit of C

One of the principles that the C committee derived from the spirit of C was that an operation should not expand to a surprisingly large amount of machine code. Arrays of bit-fields potentially require the generation of machine code to perform relatively complex calculations, compared to non-bit-field element accesses, to calculate out the offset of an element from the array index, and to extract the necessary bits.

byte addressable unit

The C pointer model is based on the byte as the smallest addressable storage unit. As such it is not possible to express the address of individual bits within a byte.

Other Languages

Some languages (e.g., Ada, CHILL, and Pascal) support arrays of objects that only occupy some of the bits of a storage unit. When translating such languages, calling a library routine that extracts the bits corresponding to the appropriate element is often a cost effective implementation technique. Not only does the offset need to be calculated from the index, but the relative position of the bit sequence within a storage unit will depend on the value of the index (unless its width is an exact division of the width of the storage unit). Pointers to objects that do not occupy a complete storage unit are rarely supported in any language.

footnote 105

105) 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. 1419

Commentary

bit-field int

This issue is discussed elsewhere.

C90

This footnote is new in C99.

1420 106) An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts. footnote 106

Commentary

Bit-fields, named or otherwise, in general are useful for padding to conform to externally imposed layouts (however, giving names to the padding bits can provide useful source code information).

1421 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type. member alignment

Commentary

The standard does not require the alignment of other kinds of objects to be documented. Developers sometimes need to be able to calculate the offsets of members of structure types (the `offsetof` macro was introduced into C90 to provide a portable method of obtaining this information). Knowing the size of each member, the relative order of members, and their alignment requirements is invariably sufficient information (because implementations insert the minimum padding between members necessary to produce the required alignment). 1422 member address increasing

While all members of the same union object have the same address, the alignment requirements on that address may depend on the types of the members (because of the requirement that a pointer to an object behave the same as a pointer to the first element of an array having the same object type). pointer to union members compare equal; additive operators pointer to object

C++

The C++ Standard specifies (3.9p5) that the alignment of all object types is implementation-defined.

Other Languages

Most languages do not call out a special case for the alignment of members.

Common Implementations

Most implementations use the same alignment requirements for members as they do for objects having automatic storage duration. It is possible for the offset of a member having an array type to depend on the number of elements it contains. For instance, the Motorola 56000 supports pointer operations on circular buffers, but requires that the alignment of the buffer be a power of 2 greater than or equal to the buffer size. alignment; Motorola 56000

Coding Guidelines

The discussion on making use of storage layout information is applicable here. storage layout

1422 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. member address increasing

Commentary

Although not worded as such, this is effectively a requirement on the implementation. It is consistent with a requirement on the result of comparisons of pointers to members of the same structure object. Prior to the publication of the C Standard there were several existing practices that depended on making use of information on the relative order of members in storage; including: structure members later compare later

- Accessing individual members of structure objects via pointers whose value had been calculated by performing arithmetic on the address of other members (the `offsetof` macro was invented by the committee to address this need).
- Making use of information on the layout of members to overlay the storage they occupy with other objects.

By specifying this ordering requirement the committee prevented implementations from using a different ordering (for optimization reasons), increasing the chances that existing practices would continue to work as expected (these practices also rely on other implementation-defined behaviors). The cost of breaking existing 1421 member alignment

code and reducing the possibility of being able to predict member storage layout was considered to outweigh any performance advantages that might be obtained from allowing implementations to choose the relative order of members.

C++

The C++ Standard does not say anything explicit about bit-fields (9.2p12).

Other Languages

Few other languages guarantee the ordering of structure members. In practice, most implementations for most languages order members in storage in the same sequences as they were declared in the source code. The **packed** keyword in Pascal is a hint to the compiler that the storage used by a particular record is to be minimized. A few Pascal (and Ada) implementations reorder members to reduce the storage they use, or to change alignments to either reduce the total storage requirements or to reduce access costs for some frequently used members.

Common Implementations

The quantity and quality of analysis needed to deduce when it is possible to reorder members of structures has deterred implementors from attempting to make savings, for the general case, in this area. Some impressive savings have been made by optimizers[4] for languages that do not make this pointer to member guarantee.

Palem and Rabbah[7] looked at the special case of dynamically allocated objects used to create tree structures. Building a tree structure usually requires the creation of many objects having the same type. A common characteristic of some operations on tree structures is that an access to an object, using a particular member name, is likely to be closely followed by another access to an object using the same member name. Rather than simply reordering members, they separated out each member into its own array, based on dynamic profiles of member accesses (the Trimaran[1] and `gcc` compilers were modified to handle this translation internally; it was invisible to the developer). For instance:

```
1  struct T {
2             int m_1;
3             struct T *next;
4            };
5  /*
6   * Internally treated as if written
7   */
8  int m_1[4];
9  struct T *(next[4]);
```

dynamically allocating storage for an object having type `struct T` resulted in storage for the two arrays being allocated. A second dynamic allocation request requires no storage to be allocated, the second array element from the first allocation could be used. If tree structures are subsequently walked in an order that is close to the order in which they are built, there is an increased probability that members having the same name will be in the same cache line. Using a modified `gcc` to process seven data intensive benchmarks resulted in an average performance improvement of 24% on Intel Pentium II and III, and 9% on Sun Ultra-Sparc-II.

Franz and Kistler[2] describe an optimization that splits objects across non-contiguous storage areas to improve cache performance. However, their algorithm only applies to strongly typed languages where developers cannot make assumptions about member layout, such as Java.

pointer compressing members

Zhang and Gupta[10] developed what they called the *common-prefix* and *narrow-data* transformations. These compress 32-bit integer values and 32-bit address pointers into 15 bits. This transformation is dynamically applied (the runtime system checks to see if the transformation can be performed) to the members of dynamically allocated structure objects, enabling two adjacent members to be packed into a 32-bit word (a bit is used to indicate a compressed member). The storage optimization comes from the commonly seem behavior: (1) integer values tend to be small (the runtime system checks whether the top 18 bits are all 1's or all 0's), and (2) that the addresses of the links, in a linked data structure, are often close to the address of the object they refer to (the runtime system checks whether the two address have the same top

17 bits). Extra machine code has to be generated to compress and uncompress members, which increases code size (average of 21% on the user code, excluding linked libraries) and lowers runtime performance (average 30%). A reduction in heap usage of approximate 25% was achieved (the Olden benchmarks were used).

Olden benchmark

Coding Guidelines

The order of storage layout of the members in a structure type is representation information that is effectively guaranteed. It would be possible to use this information, in conjunction with the `offsetof` macro to write code to access specific members of a structure, using pointers to other members. However, use of information on the relative ordering of structure members tends not to be code based, but data based (the same object is interpreted using different types). The coding guideline issues associated with the layout of types are discussed elsewhere.

storage layout

1423 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.

pointer to structure points at initial member

Commentary

Although not worded as such, this is effectively a requirement on the implementation. The only reason for preventing implementations inserting padding at the start of a structure type is existing practice (and the resulting existing code that treats the address of a structure object as being equal to the address of the first member of that structure).

Other Languages

Most languages do not go into this level of representation detail.

Coding Guidelines

The guideline recommendation dealing with the use of representation information is applicable here.

?? representation information using

1424 There may be unnamed padding within a structure object, but not at its beginning.

structure unnamed padding

Commentary

Unnamed padding is needed when the next available free storage, for a member of a structure type, does not have the alignment required by the member type. Another reason for using unnamed padding is to mirror the layout algorithm used by another language, or even that used by another execution environment.

alignment 1421 member alignment

The standard does not guarantee that two structure type having exactly the same member types have exactly the same storage layout, unless they are part of a common initial sequence.

structural compatibility common initial sequence

C90

There may therefore be unnamed padding within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

C++

This commentary applies to POD-struct types (9.2p17) in C++. Such types correspond to the structure types available in C.

Other Languages

No language requires implementations to pad members so that there is no padding between them. Few language specifications call out the fact that there may be padding within structure objects.

Common Implementations

Implementations usually only insert the minimum amount of unnamed padding needed to obtain the correct storage alignment for a member.

Coding Guidelines

The presence of unnamed padding increases the size of a structure object. Developers sometimes order members to minimize the amount of padding that is likely to be inserted by a translator. Ordering the members by size (either smallest to largest, or largest to smallest) is a common minimization technique. This is making use of layout information and a program may depend on the size of structure objects being less than a certain value (perhaps there may be insufficient available storage to be able to run a program if this limit is exceeded). However, it is not possible to tell the difference between members that have been intentionally ordered to minimize padding, rather than happening to have an ordering that minimizes (or gets close to minimizing) padding. Consequently these coding guidelines are silent on this issue.

Unnamed padding occupies storage bytes within an object. The pattern of bits set, or unset, within these bytes can be accessed explicitly by a conforming program (using `memcpy` or `memset` library functions). They may also accessed implicitly during assignment of structure objects. It is the values of these bytes that is a potential cause of unexpected behavior when the `memcmp` (amongst others) library function is used to compare two objects having structure type.

footnote 43

Example

```
1  #include <stdlib.h>
2
3  /*
4   * In an implementation that requires objects to have an address that is a
5   * multiple of their size, padding is likely to occur as commented.
6   */
7  struct S_1 {
8              char mem_1; /* Likely to be internal padding following this member. */
9              long mem_2; /* Unlikely to be external padding following this member. */
10            };
11 struct S_2 {
12             long mem_1; /* Unlikely to be internal padding following this member. */
13             char mem_2; /* Likely to be external padding following this member. */
14            };
15
16 void f(void)
17 {
18 struct S_1 *p_s1 = malloc(4*sizeof(struct S_1));
19 struct S_2 *p_s2 = malloc(4*sizeof(struct S_2));
20 }
```

The size of a union is sufficient to contain the largest of its members. 1425

Commentary

structure 1428 trailing padding

A union may also contain unnamed padding.

---

union member at most one stored

The value of at most one of the members can be stored in a union object at any time. 1426

Commentary

union type overlapping members

union 1427 members start same address

union member when written to

This statement is a consequence of the members all occupying overlapping storage and having their first byte start at the same address. The value of any bytes of the object representation that are not part of the value representation, of the member last assigned to, are unspecified.

Other Languages

Pascal supports a construct, called a *variant tag*, that can be used by implementations to check that the member being read from was the last member assigned to. However, use of this construct does require that developers explicitly declare such a tag within the type definition. A few implementations perform the check

suggested by the language standard. Ada supports a similar construct and implementations are required to perform execution time checks, when a member is accessed, on what it calls the *discriminant* (which holds information on the last member assigned to).

Common Implementations

The RTC tool[5] performs runtime type checking and is capable of detecting some accesses (it does not distinguish between different pointer types and different integer types having the same size) where the member read is different from the last member stored in.

1427 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.

union members start same address

Commentary

Although not worded as such, this is effectively a requirement on the implementation. A consequence of this requirement is that all members of a union type have the same offset from the start of the union, zero. A previous requirement dealt with pointer equality between different members of the same union object. This C sentence deals with pointer equality between a pointer to an object having the union type and a pointer to one of the members of such an object.

pointer to union members compare equal

C++

This requirement can be deduced from:

> *Each data member is allocated as if it were the sole member of a* ***struct****.* 9.5p1

Other Languages

Strongly typed languages do not usually (Algol 68 does) provide a mechanism that returns the addresses of members of union (or structure) objects. The result of this C requirement (that all members have the same address) are not always specified, or implemented, in other languages. It may be more efficient on some processors, for instance, for members to be aligned differently (given that in many languages unions may only be contained within structure declarations and so could follow other members of a structure).

Common Implementations

The fact that pointers to different types can refer to the same storage location, without the need for any form of explicit type conversion, is something that optimizers performing points-to analysis need to take into account.

Coding Guidelines

The issues involved in having pointers to different types pointing to the same storage locations is discussed elsewhere.

pointer qualified/unqualified versions

1428 There may be unnamed padding at the end of a structure or union.

structure trailing padding

Commentary

The reasons why an implementation may need to add this padding are the same as those for adding padding between members. When an array of structure or union types is declared, the first member of the second and subsequent elements needs to have the same alignment as that of the first element. In:

1424 structure unnamed padding

```
1  union T {
2            long m_1;
3            char m_2[11];
4           };
```

it is the alignment requirements of the member types, rather than their size, that determines whether there is any unnamed padding at the end of the union type. When one member has a type that often requires alignment on an even address and another member contains an odd number of bytes, it is likely that some unnamed padding will be used.

C++

The only time this possibility is mentioned in the C++ Standard is under the **sizeof** operator:

5.3.3p2 *When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array.*

Other Languages

The algorithms used to assign offsets to structure members are common to implementations of many languages, including the rationale for unnamed padding at the end. Few language definitions explicitly call out the fact that structure or union types may have unnamed padding at their end.

Common Implementations

Most implementations use the same algorithm for assigning member offsets and creating unnamed padding for all structure and union types in a program, even when these types are anonymous (performing the analysis to deduce whether the padding is actually required is not straight-forward). Such an implementation strategy is likely to waste a few bytes in some cases. But it has the advantage that, for a given implementation and set of translator options, the same structure declarations always have the same size (there may not be any standard's requirement for this statement to be true, but there is sometimes a developer expectation that it is true).

Coding Guidelines

Unnamed padding is a representation detail associated with storage layout. That this padding may occur after the last declared member is simply another surprise awaiting developers who try to make use of storage layout details. The guideline recommendation dealing with the use of representation information is applicable here.

storage layout representation information using ??

As a special case, the last element of a structure with more than one named member may have an incomplete array type; 1429

Commentary

The Committee introduced this special case, in C99, to provide a standard defined method of using what has become known as the *struct hack*. Developers sometimes want a structure object to contain an array object whose number of elements is decided during program execution. A standard, C90, well defined, technique is to have a member point at dynamically allocated storage. However, some developers, making use of representation information, caught onto the idea of simply declaring the last member be an array of one element. Storage for the entire structure object being dynamically allocated, with the storage allocation request including sufficient additional storage for the necessary extra array elements. Because array elements are contiguous and implementations are not required to perform runtime checks on array indexes, the additional storage could simply be treated as being additional array elements. This C90 usage causes problems for translators that perform sophisticated flow analysis, because the size of the object being accessed does not correspond to the size of the type used to perform the access. Should such translators play safe and treat all structure types containing a single element array as their last member as if they will be used in a *struct hack* manner?

The introduction of flexible array members, in C99, provides an explicit mechanism for developers to indicate to the translator that objects having such a type are likely to have been allocated storage to make use of the *struct hack*.

The presence of a member having an incomplete type does not cause the structure type that contains it to have an incomplete type.

C90

The issues involved in making use of the *struct hack* were raised in DR #051. The response pointed out declaring the member to be an array containing fewer elements and then allocating storage extra storage for

additional elements was not strictly conforming. However, declaring the array to have a large number of elements and allocating storage for fewer elements was strictly conforming.

```
#include <stdlib.h>
#define HUGE_ARR 10000 /* Largest desired array. */

struct A {
           char x[HUGE_ARR];
         };

int main(void)
{
struct A *p = (struct A *)malloc(sizeof(struct A)
                                 - HUGE_ARR + 100); /* Want x[100] this time. */
p->x[5] = '?'; /* Is strictly conforming. */
return 0;
}
```

Support for the last member having an incomplete array type is new in C99.

C++

Support for the last member having an incomplete array type is new in C99 and is not available in C++.

Common Implementations

All known C90 implementations exhibit the expected behavior for uses of the *struct hack*. However, some static analysis tools issue a diagnostic on calls to `malloc` that request an amount of storage that is not consistent (e.g., smaller or not an exact multiple) with the size of the type pointed to by any explicit cast of its return value.

Coding Guidelines

Is the use of flexible arrays members more or less error prone than using any of the alternatives?

The *struct hack* is not widely used, or even widely known about by developers (although there may be some development communities that are familiar with it). It is likely that many developer will not be expecting this usage. Use of a member having a pointer type, with the pointed-to object being allocated during program execution, is a more common idiom (although more statements are needed to allocate and deallocate storage; and experience suggests that developers sometimes forget to free up the additional pointed-to storage, leading to storage leakage).

From the point of view of static analysis the appearance of a member having an incomplete type provides explicit notification of likely usage. While the appearance of a member having a completed array type is likely to be taken at face value. Without more information on developer usage, expectations, and kinds of mistakes made it is not possible to say anything more on these possible usages.

1430 this is called a *flexible array member*.

flexible array member

Commentary

This defines the term *flexible array member*.

C++

There is no equivalent construct in C++.

1431 ~~With two exceptions~~ In most situations, the flexible array member is ignored.

flexible array member ignored

Commentary

The following are some situations where the member is ignored:

- forming part of a common initial sequence, even if it is the last member,

- compatibility checking across translation units, and
- if an initializer is given in a declaration (this is consistent with the idea that the usage for this type is to allocate variably sized objects via `malloc`).

structure size with flexible member

~~First, the size of the structure shall be equal to the offset of the last element of an otherwise identical structure that replaces the flexible array member with an array of unspecified length.[106)]~~ 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. 1432

Commentary

The C99 specification required implementations to put any padding before the flexible array member. However, several existing implementations (e.g., GNU C, Compaq C, and Sun C) put the padding after the flexible array member. Because of the efficiency gains that might be achieved by allowing implementations to put the padding after the flexible array member the committee decided to sanction this form of layout.

The wording was changed by the response to DR #282.

~~Second~~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; 1433

Commentary

The structure object acts as if it effectively grows to fill the available space (but it cannot shrink to smaller than the storage required to hold all the other members).

the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. 1434

Commentary

This is a requirement on the implementation. It effectively prevents an implementation inserting additional padding before the flexible array member, dependent on the size of the array. Fixing the offset of the flexible array member makes it possible for developers to calculate the amount of additional storage required to accommodate a given number of array elements.

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. 1435

Commentary

In the following example:

```
1  struct T {
2            int mem_1;
3            float mem_2[];
4           } *glob;
5
6  glob=malloc(sizeof(struct T) + 1);
```

insufficient storage has been allocated (assuming `sizeof(float) != 1`) for there to be more than zero elements in the array type of the member `mem_2`. However, the requirements in the C Standard are written on the assumption that it is not possible to create a zero sized object, hence this *as-if* specification.

Other Languages

Few languages support the declaration of object types requiring zero bytes of storage.

1436 EXAMPLE ~~Assuming that all array members are aligned the same, after the declarations:~~

EXAMPLE
flexible member

```
~~struct s { int n; double d[]; };~~
~~struct ss { int n; double d[1]; };~~
```

~~the three expressions:~~

```
~~sizeof (struct s)~~
~~offsetof(struct s, d)~~
~~offsetof(struct ss, d)~~
```

~~have the same value. The structure **structs** has a flexible array member **d**.~~
If **sizeof (double)** is 8, then after the following code is executed:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

and assuming that the calls to **malloc** succeed, the objects pointed to by **s1** and **s2** behave, for most purposes, as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

Following the further successful assignments:

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) +  6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]);        // valid
*dp = 42;                // valid
dp = &(s2->d[0]);        // valid
*dp = 42;                // undefined behavior
```

The assignment:

```
*s1 = *s2;
```

only copies the member **n** ; if any of the array elements are within the first **sizeof(structs)** bytes of the structure, these might be copied or simply overwritten with indeterminate values. ~~and not any of the array elements. Similarly:~~

```
~~struct s  t1 = { 0 };            // valid~~
~~struct s  t2 = { 2 };            // valid~~
~~struct ss tt = { 1, { 4.2 }};    // valid~~
~~struct s  t3 = { 1, { 4.2 }};    // invalid: there is nothing for the 4.2 to initialize~~

~~t1.n = 4;                        // valid~~
~~t1.d[0] = 4.2;                   // undefined behavior~~
```

Commentary

Flexible array members are a new concept for many developers and this extensive example provides a mini-tutorial on their use.

The wording was changed by the response to DR #282.

footnote 106

~~106) The length is unspecified to allow for the fact that implementations may give array members different alignments according to their lengths.~~ 1437

Commentary

Motorola 56000

One reason for an implementation to use different alignments for array members of different lengths is to take advantage of processor instructions that require arrays to be aligned on multiples of their length.

The wording was changed by the response to DR #282.

Forward references: tags (6.7.2.3). 1438

References

1. Anon. Trimaran home page. www.trimaran.org, 2003.
2. M. Franz and T. Kistler. Splitting data objects to increase cache utilization. Technical Report Technical Report No. 98-34, Department of Information and Computer Science, University of California, Irvine, 1998.
3. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.
4. T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, 2000.
5. A. Loginov, S. H. Yong, S. Horowitz, and T. Reps. Debugging via run-time type checking. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering 4th International Conference (FASE 2001)*, pages 217–232. Springer-Verlag, Apr. 2001.
6. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
7. K. V. Palem and R. M. Rabbah. Bridging processor and memory performance in ILP processors via data-remapping. Technical Report GIT-CC-01-014, Georgia Institute of Technology, June 2001.
8. P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 324–332, June 1998.
9. K. Thompson. Plan 9 C compilers. In *Plan 9 Programmer's Manual*. AT&T Bell Laboratories, 1995.
10. Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In R. N. Horspool, editor, *Compiler Construction* 11^{th} *International Conference, CC 2002*, pages 14–28. Springer-Verlag, Apr. 2002.