The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.2.6.1 General

types representation

The representations of all types are unspecified except as stated in this subclause. 569

Commentary

footnote 31 complex component representation qualifiers representation and alignment pointer to void same representation and alignment as floating types characteristics

These representations are requirements on the implementation. Previous clauses in the C Standard specify cases where some integer types, complex types, qualified types and pointer types must share the same representation. Subclause 5.2.4.2.2 describes one possible representation of floating-point numbers. Other clauses in the standard specify when two or more types have the same representation. However, they say nothing about what the representation might be.

C90

This subclause is new in C99, although some of the specifications it contains were also in the C90 Standard.

C++

3.9p1 *[Note: 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types.*

These C++ subclauses go into some of the details specified in this C subclause.

Other Languages

Most languages say nothing about the representation of types. The general acceptance of the IEC 60559 Standard means there is sometimes some discussion about using a floating-point representation that conforms to this standard. Java is intended to be processor-independent. To achieve this aim, the representation from a developer's point of view of all arithmetic types are fully specified.

Coding Guidelines

Code that makes use of representation information leaves itself open to several possible additional costs:

- The representation can vary between implementations. The potential for differences in representations between implementations increases the likelihood that there will be unexpected effort required to port programs to new environments.
- The need for readers to consider representation information increase the cognitive effort needed to comprehend code. This increase in required effort can increase the time needed by developers to complete source code related tasks.
- The failure to consider representation information by developers can lead to faults being introduced when existing code is modified.
- The additional complexity introduced by the need to consider representation information increases the probability that the maximum capacity of a reader's cognitive ability will be exceeded (i.e., the code will be too complicated to comprehend). Human response to information overload is often to ignore some of the information, which in turn can lead to an increase in the number of mistakes made.

cognitive effort

overconfidence

Developers often have a misplaced belief in their ability to use representation information to create more efficient programs (e.g., less time to execute or less machine code generated).

Cg 569.1 A program shall not make use of information on the representation of a type.

Dev 569.1 A program may make use of representation information provided this usage is documented and a rationale given for why it needs to be used.

570 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

object contiguous sequence of bytes

Commentary

A bit-field can occupy part of one or two bytes (it may occupy more than two bytes, but the third and subsequent bytes will be fully occupied). It can also share a byte with other bit-fields.

The representation of a type must contain sufficient bytes so that it can represent the range of values the standard specifies it must be able to represent. An implementation may choose to use more bytes; for instance, to enable it to represent a greater range of values, or to conform to some processor storage requirement. Requiring that the bytes of an object be contiguous only becomes important when the object is operated on as a sequence of bytes. Such operations are part of existing practice; for instance, using the library function `memcpy` to copy arrays and structures. The number of bytes in an object can be found by using the **`sizeof`** operator.

sizeof result of

In the case of pointers, the response to DR #042 (which involved accesses using `memcpy`, but the response has since been taken to apply to all kinds of access) pointed out that " . . . objects are not "the largest objects into which the arguments can be construed as pointing."", and " . . . a contiguous sequence of elements within an array can be regarded as an object in its own right." and "the non-overlapping halves of array . . . can be regarded as objects in their own rights." In the following example the storage `p_1` and `p_2` can be considered to be pointing at different objects:

```
#include <string.h>

#define N 20

char a[2*N];

void f(void)
{
char *p_1 = a,
     *p_2 = a+N;

/* ... */
memcpy(p_2, p_1, N);
}
```

The ordering of bytes within an object containing more than one of them is not specified, and there is no way for a strictly conforming program can obtain this information. The order that bytes appear in may depend on how they are accessed (e.g., shifting versus access via a pointer).

The terms *little-endian* (byte with lowest address occupies the least significant position) and *big-endian* (byte with lowest address occupies the most significant position) originate from Jonathan Swift's book,

endian

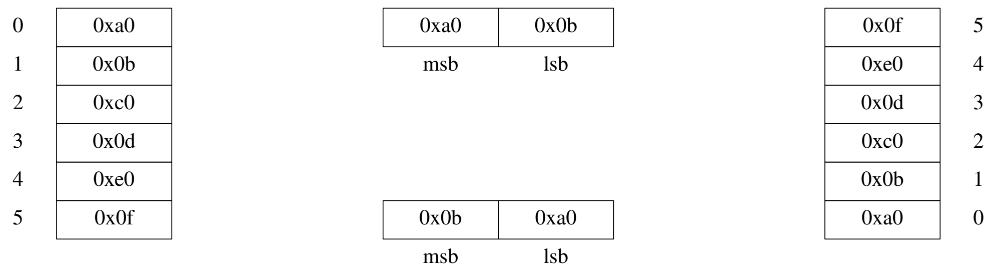

Figure 570.1: Developers who use little-endian often represent increasing storage locations going down the page. Developers who use big-endian often represent increasing storage locations going up the page. The value returned by an access to storage location *0*, using a pointer type that causes 16 bits to be read, will depend on the *endianness* of the processor.

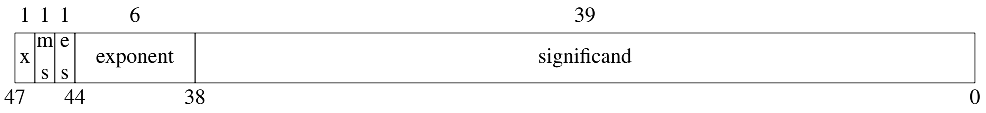

Figure 570.2: The Unisys A Series[5] uses the same representation for integer and floating-point types. For integer values bit 47 is unused, bit 46 represents the sign of the significand, bits 45 through 39 are zero, and bits 38 through 0 denote the value (a sign and magnitude representation). For floating values bit 47 represents the sign of the exponent and bits 46 through 39 represent the exponent (the representation for double-precision uses an additional word with bits 47 through 39 representing higher order-bits of the exponent and bits 38 through 0 representing the fractional portion of the significand).

Gulliver's Travels. Swift invented the terms to describe the egg-eating habits of two groups of people who got so worked up about which way was best that they went to war with each other over the issue.

C++

1.8p5 *An object of POD[4)] type (3.9) shall occupy contiguous bytes of storage.*

The acronym POD stands for *Plain Old Data* and is intended as a reference to the simple, C model, or laying out objects. A POD type is any scalar type and some, C compatible, structure and union types.

In general the C++ Standard says nothing about the number, order, or encoding of the bytes making up what C calls an object, although C++ does specify the same requirements as C on the layout of members of a structure or union type that is considered to be a POD.

Other Languages

Java requires integer types to behave as if they were contiguous; how the underlying processor actually represents them is not visible to a program.

Common Implementations

The Motorola DSP563CCC[3] uses two 24-bit storage units to represent floating-point values. The least significant 24 bits is used to represent the exponent (in the most significant 14 bits, the remaining bits being reserved). The significand is represented in the most significant storage unit (in two's complement).

Table 570.1: Byte order (indicated by the value of the digits) used by various processors for some integer and floating types, in different processor address spaces (all address spaces if none is specified).

Vendor	16-bit integer	32-bit integer	64-bit integer	32-bit float	64-bit float
AT&T 3B2		4321 (data space)/ 1234 (program space)			
DEC PDP–11	12	3412		3412 (F format)	78563412 (D format)
DEC VAX	12	1234	12345678	3412 (F format)	78563412 (D format)
NSC32016		1234 (data space)/ 4321 (program space)			

Coding Guidelines

sizeof constraints

Information on the number of bytes in a type is needed by the memory-allocation functions. The **`sizeof`** operator provides a portable way of obtaining this information. It is common C practice to copy values from one object to another using some form of block copy of bytes between storage locations. Perhaps part of the reason for this is lack of awareness, by developers, that objects having a structure type can be assigned, or in the case of objects having an array type, because there is no appropriate assignment operator available for this type category.

There are no portable constructs that provide information on the order or encoding of the bytes in an object. The only way to obtain this information is to use constructs whose behavior is implementation-defined. For these cases the guideline recommendation on using representation information is applicable.

representation information using 569.1

571 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[40]

unsigned char pure binary

Commentary

This is a requirement on the implementation. It prevents an implementation from using any of the bits in one of these types for other purposes. For instance, an implementation cannot define the type **unsigned char** to be 9 bits, the most significant bit being a parity bit and the other bits being the value bits; all 9 bits would have to participate in the representation of the type (or the byte size reduced to 8 bits).

This requirement implies that using the type **unsigned char** to access the bytes in an object guarantees that all of the bits in that object will be accessed (read or written). There is no such requirement for any other types. For instance, the type **int** may contain bits in its object representation that do not participate in the value representation of that object. Taking the address of an object and casting it to pointer-to **unsigned char** makes all of the bits in its object representation accessible. This requirement is needed to implement the library function `memcpy`, among others.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. 3.9.1p1

The C++ Standard does not include unsigned bit-fields in the above requirement, as C does. However, it is likely that implementations will follow the C requirement.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

The lack of such a guarantee for the other types only becomes visible when programs get involved with details of the representation. The guideline on not using representation information is applicable here.

569.1 representation information using

572 Values stored in non-bit-field objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes.

Commentary

This means that non-bit-field objects cannot share the sequence of bytes they occupy with any other object (members of union types share the same storage, but only one value can be stored in such a type at any time).

C90

This level of detail was not specified in the C90 Standard (and neither were any of the other details in this paragraph).

C++

*The object representation of an object of type `T` is the sequence of N **unsigned char** objects taken up by the object of type `T`, where N equals `sizeof(T)`.* 3.9p4

That describes the object representation. But what about the value representation?

3.9p4

The value representation of an object is the set of bits that hold the value of type T. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.[37)]

This does not tie things down as tightly as the C wording. In fact later on we have:

3.9.1p1 *For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types.*

QED.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

Developers sometimes calculate the range of values that a type, in a particular implementation, can represent based on the number of bits in its object representation (which for most commonly used processors will deliver the correct answer). Such calculations are the result of developers focusing too narrowly on the details in front of them. There are object-like macros in the **<limits.h>** header that provide this information.

```
#include <limits.h>

unsigned short lots_of_assumptions(void)
{
return (1 << (sizeof(short)*CHAR_BIT)) - 1;
}
```

representation information using 569.1

The following is a special case of the guideline recommendation dealing with the use of representation information (just in case developers regard this as a special case).

Rev 572.1

The possible range of values that an integer type can represent shall not be calculated from the number of bits in its object representation.

---

value copied using unsigned char

The value may be copied into an object of type **unsigned char [*n*]** (e.g., by **memcpy**); 573

Commentary

Using the type **unsigned char** guarantees that all of the bits in the original object representation will be copied because there cannot be any unused bits in that type. While code may be written to copy the value into an array of any type, the standard only guarantees the behavior for this case, and when the element type is the same as the object being copied. Once the value has been copied into an object having type array of **unsigned char** copying them back to an object of the original type will restore the original pattern of bits, and hence the original value. The type pointer to **unsigned char** is a commonly used interface mechanism that offers a generic way of copying one region of storage to another region of storage. Although the library functions provide this functionality, there is existing code that does such copies as loops within the code itself rather than via calls to library functions.

C90

This observation was first made by the response to DR #069.

Other Languages

Many languages do not support the equivalent of the memory copying and related library functions. Neither do they define the behavior of casting pointers to different types, which enables developers to provide such functionality. So this discussion does not apply to them.

Common Implementations

All implementations known to your author use the same number of bits in the representation of the types **`unsigned char`** and **`signed char`** and there are no trap representations in the object representation of the type **`signed char`**. In such implementations values may be copied using an array of any character type.

Coding Guidelines

Copying objects a byte at a time is making use of representation information. However, this usage is sufficiently common in existing source to be regarded as a cultural norm.

Dev 569.1 An object may be copied by treating its contents as a sequence of objects having type **`unsigned char`**.

---

574 the resulting set of bytes is called the *object representation* of the value. object representation

Commentary

This defines the term *object representation* (it was first used by the response to DR #069). An object representation is the complete sequence of bits making up any object. It differs from the value representation in that it may contain additional bits (which do not form part of the value). An object representation may be thought of in terms of being a sequence of bits, while the value representation is the interpretation of the contents of storage according to a type. value representation

Other Languages

The term *object* usually has a very different meaning in object-oriented languages. In other languages, this term, if used, is often not defined in this way.

Coding Guidelines

The term *object representation* was introduced in C99 and is not used by developers in this context. Many developers have some familiarity with object-oriented languages. It is likely that they will assume this term has a very different meaning from its actual C99 definition. It is probably a good idea not to use the term *object representation* in guidelines unless plenty of explanatory information is also provided on the terminology.

---

575 Values stored in bit-fields consist of *m* bits, where *m* is the size specified for the bit-field. bit-field value is m bits

Commentary

This is a requirement on the implementation. Bit-fields cannot contain any padding bits. A previous requirement specified the representation for bit-fields having an unsigned type. The integer constant given in the declaration of the bit-field member corresponds to the value of *m*. There is a maximum value that *m* can take.

571 unsigned char pure binary

bit-field maximum width

C++

9.6p1

The `constant-expression` may be larger than the number of bits in the object representation (3.9) of the bit-field's type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field.

This specifies the object representation of bit-fields. The C++ Standard does not say anything about the representation of values stored in bit-fields.

C++ allows bit-fields to contain padding bits. When porting software to a C++ translator, where the type **`int`** has a smaller width (e.g., 16 bits), there is the possibility that some of the bits will be treated as padding bits on the new host. In:

```
struct T {
          unsigned int m1:18;
         };
```

the member `m1` will have 18 value bits when the type **`unsigned int`** has a precision of 32, but only 16 value bits when **`unsigned int`** has a precision of 16.

Other Languages

A few languages (e.g., Ada, CHILL, and PL/1) provide functionality for enabling developers to specify the number of bits of storage to use in representing objects having integer types. Languages in the Pascal family support the specification of types that are subranges of the integer type. Some implementations chose to pack members of records having such types into the smallest number of bits, effectively having the same outcome as a C bit-field definition. But this implementation detail is hidden from the developer and an implementation is at liberty to use exactly the same amount of storage for all integer types.

The object representation is the set of *m* bits the bit-field comprises in the addressable storage unit holding it. 576

Commentary

byte
addressable unit

This defines the object representation for bit-fields. It differs from the object representation for other types in that it is based on addressable storage units rather than bytes. This is because the storage occupied by a bit-field might only partially fill two bytes and share that storage with other bit-fields.

Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations. 577

Commentary

equality operators
syntax

To be exact, two object representations whose values are interpreted using the same type can compare equal (using an equality operator) and have different object representations (two object representations can only be compared by treating them as an array of **`unsigned char`** objects and can be compared by using, for instance, the library function `memcmp`). This difference can occur if there are padding bits in the object representation that differ. Thus, although the values compare equal, the object representations are different. Two object representations can have the same bit pattern, but compare unequal when interpreted using two different types (e.g., integer and floating types).

There are also cases that do not involve padding bits in the object representation, two values comparing equal, but the object representations being different. This can occur when there is more than one representation for the same value; for instance, plus and minus zero in signed magnitude notation, or when the values are different but compare equal (i.e., plus and minus floating-point zero).

x == y
x not same as y

NaN

NaN always compares unequal to anything, including NaN.

C++

3.9p3 *For any POD type T, if two pointers to T point to distinct T objects `obj1` and `obj2`, if the value of `obj1` is copied into `obj2`, using the `memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`.*

This handles the first case above. The C++ Standard says nothing about the second value compare case.

Coding Guidelines

This statement highlights the dangers of dealing with object representations. They can contain padding bits whose value may not be explicitly controlled by developers.

Example

```
#include <string.h>

extern int glob_1,
           glob_2;

int f(void)
{
if (memcmp(&glob_1, &glob_2, sizeof(glob_1)) == 0)
   {
   if (glob_1 == glob_2)
      return 1;
   else
      return 2;
   }
return 3;
}
```

578 Certain object representations need not represent a value of the object type.

Commentary

On some processors, pointers have a limit on the maximum amount of memory they can access; for instance, 24 bits of a possible 32-bit pointer representation can refer to an object.

The IEC 60559 Standard specifies that certain bit patterns do not represent floating-point numbers. However, these bit patterns are still values within the standard. They represent such quantities as the infinities and NaNs. These are all values of the object type.

floating types can represent

C90

This observation was not explicitly made in the C90 Standard.

C++

The value representation of an object is the set of bits that hold the value of type T. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.[37)] 3.9p4

37) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Languages C. Footnote 37

By implication this is saying what C says. It also explicitly specifies that these representation issues are implementation-defined.

Other Languages

Most languages do not get involved in specifying this level of detail.

Common Implementations

This situation is sometimes seen for pointer types. A processor may place restrictions on what storage locations can be addressed (perhaps because the program does not have access rights to them, or because there is no physical storage at those locations). The representation of the floating-point significand on the Motorola DSP563CCC[3] reserves the bit patterns `0x000001` through `0x3fffff` and `0xc00000` through `0xffffff`.

Coding Guidelines

Although such object representations may exist, creating a value having such a representation invariably relies on undefined or implementation-defined behavior:

- An object that has not been initialized can contain an object representation that is not a value in the object type.
- For pointer and floating-point types, there are implementations where some object representations do not represent a value of the object type. Creating such a representation invariably requires modifying the object through types other than the underlying pointer or floating-point type.

---

trap representation reading is undefined behavior

If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. 579

Commentary

The pattern of bits held in an object has no meaning until they are interpreted as having a particular type. A pattern of bits that does not correspond to a value of the type used for the read access may be treated by the host processor in special ways— for instance, raising an exception. This statement points out this fact.

The exception for character types is to support the practice of copying objects using a pointer-to character type. This would not work if accessing one or more values, having a character type, represented undefined behavior.

C90

The C90 Standard specified that reading an uninitialized object was undefined behavior. But, it did not specify undefined behaviors for any other representations.

C++

The C++ Standard does not explicitly specify any such behavior.

Other Languages

Most languages specified some form of undefined or implementation-defined behavior for a read access to an uninitialized object. But they are usually silent on other kinds of *non-values*.

Common Implementations

There are so few processors that have a sequence of bits in the value representation that does not represent a value, it is not possible to determine any common behaviors.

pointer cause undefined behavior
pointer 590 segmented architecture

Some processors check that the value of address operands is within the bounds of addressable storage. This kind of checking is common in processors that represent addresses using a *segment+offset* representation. In this case the segment number is often checked to ensure that it denotes a segment that can be accessed by the current process.

Coding Guidelines

object initial value indeterminate
representation information using 569.1

A guideline recommending that such values not be read is acting after the fact, they should not have been created in the first place. These issues are dealt with under uninitialized objects and making use of representation information.

If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[41)] 580

Commentary

The representation could be produced by doing one of the following:

- Assigning it from another object of the same type; in which case the behavior is undefined because of the previous sentence.

- Treating all or part of the object as a type different from its declared type; in which case the behavior is also undefined. effective type
- manipulating the values of bits directly using a sequence of operations on operands. While creating the value may not be undefined behavior, storing it into the appropriate type could be.

If a pointer to character type is used to copy all of the bits in an object to another object, the transfer will be performed a byte at a time. A trap representation might be produced after some bytes have been copied, but not all of them. An exception to the general case is thus needed for character types.

C90

The C90 Standard did not explicitly specify this behavior.

C++

The C++ Standard does not explicitly discuss this issue.

Other Languages

Other languages do not usually give special status to values copied using character types.

Common Implementations

Few implementations do anything other than treat whatever bit pattern they are presented with as a value. In other cases processors raise some kind of exception, or set some status bits.

Coding Guidelines

These side effects are dealt with in more detail elsewhere. object value accessed if type

---

581 Such a representation is called a *trap representation*. trap representation

Commentary

This defines the term *trap representation*. The standard does not require that use of this representation cause a processor to generate trap (the original source for this terminology). As specified in the previous two sentences, the behavior is undefined. The term *trap representation* is not often used by developers because there are few implementations that contain such a representation. The Committee's response to DR #222 said, in part: "A TC should remove the notion of objects of struct or union type having a trap representation . . . ".

C90

This term was not defined in the C90 Standard.

C++

Trap representations in C++ only apply to floating-point types.

Common Implementations

Support for a trap representations outside of pointer and floating-point types is very rarely seen. A few processors use tagged memory; for instance, the Tera computer system[1] has four access bits for each 64-bit object, two of these bits are available for the translator implementor to use. Possible uses include stack limit checking and runtime type exception signaling. (If a location's trap bit is set and the corresponding trap-disable bit in the pointer is clear, a trap will occur.)

The IEC 60559 Standard specifies support for signaling NaNs. While these trigger the invalid exception when accessed, they are not trap representations (although their use may be intended to act as a trap for when certain kinds of situations occur during program execution).

Note. Some processors are said to *trap* for reasons other than accessing a trap representation. For instance, use of an invalid instruction is often defined, by processor specifications, to cause a trap.

---

582 40) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. footnote 40

Commentary

The bit with the highest position might be used as a sign bit.

C90

Integer type representation issues were discussed in DR #069.

Other Languages

Most languages rely on the definitions given for this representation provided by other standards and don't repeat them in modified form.

(Adapted from the *American National Dictionary for Information Processing Systems*.) 583

Commentary

ISO 2382

The ISO definition is in ISO 2382–1.

---

unsigned char value range

A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$. 584

Commentary

CHAR_BIT macro
UCHAR_MAX value

This duplicates requirements given elsewhere for CHAR_BIT and UCHAR_MAX.

C++

The C++ Standard includes the C library by reference, so a definition of CHAR_BIT will be available in C++.

3.9.1p4 *Unsigned integers, declared **unsigned**, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.*[41)]

From which it can be deduced that the C requirement holds true in C++.

---

footnote 41

41) Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it. 585

Commentary

object initial value indeterminate

Such initialization could be performed, for instance, by the implementation on function entry. However, an implementation does not have access to any special instructions that could not also be used in the translation of the rest of a C program. It would need to make use of the exception granted for character types. An implementation could also choose to not explicitly initialize such a variable (the most common situation), using the pattern of bits it happens to contain. Whether this pattern of bits is a trap representation is something that an implementation does not need to be concerned about.

C90

The C90 Standard did not discuss trap representation and this possibility was not discussed.

C++

The C++ Standard does not make this observation about possible implementation behavior.

Other Languages

This technique, for helping developers find uninitialized variables, has been used by implementations of various languages.

Common Implementations

The technique of giving uninitialized objects a representation that caused some form of unexpected behavior, if they are read without first being given an explicit value in the code, has long been used by implementations. It predates the design of the C language.

---

value stored in structure
value stored in union

When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[42)] 586

Commentary

As the footnote points out, there are two possible ways of copying structure or union objects. Depending on the method used any padding bits may, or may not, also be copied.

footnote 42

Scalar types may include padding bits in their representation, structure types may have padding between members, and both structure and union types may have padding after the last member.

structure unnamed padding
structure trailing padding
footnote DR283

The response to DR #283 clarified the situation with regard to wording that appeared in C90 but not C99.

C90

The sentences:

With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined 41).

41) The "byte orders" for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately sized array of character type), but must be accounted for when conforming to externally-imposed storage layouts.

appeared in C90, but does not appear in C99.

If a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined in C90 and unspecified in C99.

C++

This specification was added in C99 and is not explicitly specified in the C++ Standard.

Other Languages

The values of padding bytes, or even the very existence of such bytes, is not usually discussed in other languages.

Common Implementations

Many implementations simply copy all of the bytes making up an object having structure or union type; it is the simplest option. Whether the copy occurs a byte at a time or in multiple bytes (it is usually much more efficient to copy as many bytes as possible in a single instruction) is an implementation detail that is hidden from the developer. For padding bytes to play a part in the choice of algorithm used to make the copy there would have to be a significant percentage of the number of bytes needing to be copied.

In the case of an object having union type a translator may, or may not be able to deduce which member was last assigned to. In the former case it can copy the member assigned to, while in the latter case it has to assume that the largest member is the one that has to be copied. Whether padding bytes are copied can depend on the context in which the copy occurs.

Cyclone C[2] supports tagged union types, which contain information that identifies the current member.

Coding Guidelines

A program cannot rely on the padding bits of an object having structure or union type remaining unchanged when one of its members is modified, or the entire object is assigned a new value. Similarly, a program cannot assume that because one structure object was assigned to another structure object, a call to `memcmp` will return zero.

Copying a structure object using the assignment operator is still not widely used by developers, even though support for both structure assignment, parameter passing, and functions returning a structure has been available since the C90 Standard. Many developers continue to use the `memcpy` library function to copy objects having structure type. In this case any padding bytes will also be copied and the two objects will compare equal after the copy operation.

The issue of copying and assigning structure objects is discussed elsewhere.

footnote 42

Example

```
#include <string.h>

struct T {
          double m_1;
          short m_2;
         } x, y;

_Bool f(void)
{
x = y;
return memcmp(&x, &y, sizeof(x)) == 0;
}

void g(void)
{
union {
       char a[5];
       char b[2];
      } u;

u.a[4] = 7;
/*
 * The following assignment to the member b effectively turns
 * most of the elements of member a into padding bytes.
 */
u.b[1] = 2;
/*
 * After the following assignment it is not guaranteed
 * that u.a[4] has the value 7.
 */
u.a[0] = 3;
}
```

~~The values of padding bytes shall not affect whether the value of such an object is a trap representation.~~ The value of a structure or union object is never a trap representation, even though the value of a member of a structure or union object may be a trap representation. 587

Commentary

This is a requirement on the implementation. If it is possible for such a type to have a trap representation, it is the implementation's responsibility to ensure that the padding bytes never take on values that generate such a representation.

The wording was changed by the response to DR #222.

C90

This requirement is new in C99.

C++

This wording was added in C99 and is not explicitly specified in the C++ Standard.

Common Implementations

Trap representations are not usually associated with non-scalar types.

~~Those bits of a structure or union object that are in the same byte as a bit-field member, but are not part of that member, shall similarly not affect whether the value of such an object is a trap representation.~~ 588

Commentary

This sentence was deleted by the response to DR #222.

As an example, consider an implementation where a particular pattern of bits within a 32-bit storage unit constitutes a trap representation. If a member of a structure type is declared to have a width of 28 bits and the translator assigns storage for it within the 32-bit storage unit without allocating any member to the remaining 4 bits, then it is the implementation's responsibility to ensure that either:

- These padding bits never take on values that generate a trap representation. This could enable the translator to generate machine code that loaded the value held in the 32-bit storage unit in a single instruction, followed by instructions to scrap off the unwanted 4 bits.
- For accesses to the bit-field, generate machine code that never loads the 32-bit storage unit directly, but uses a sequence of loads of narrower quantities (16 or 8 bits). These narrower quantities are joined together to create the value of the 28-bit wide bit-field. This ensures that, although the 32-bit storage unit may hold a trap representation it is never accessed as such.

Common Implementations

While a few processors[4] contain instructions that can load a specific number of bits from storage, it is likely that the underlying hardware operations will access the complete byte or word to extract the required bits. The number of processors that support bit-field access is low. Your author does not know of any processor supporting such instructions and a trap representation.

---

589 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values~~, but the value of the union object shall not thereby become a trap representation~~.

union member when written to

Commentary

This is a requirement on the implementation. If trap representations are supported, then any padding bytes must never take on values that would cause the value of the union object to become a trap representation.

Any bytes in a union type's object representation that are not part of the representation of any of the members are padding bytes. A consequence of these bytes taking unspecified values is that accessing the value of other members is undefined behavior. (The standard does not require the value representation of any scalar type to completely overlap the value representation of another scalar type; their object representations may contain padding bytes at various offsets such that a complete overlap cannot occur.)

structure trailing padding

The wording was changed by the response to DR #222.

Common Implementations

For non-bit-field members, most implementations store operation into a member object does not affect the values held in other bytes of a union object. For bit-field members, an implementation may chose to write a value into all of the bytes occupied by the storage unit containing the bit-field. (This is an optimization that simplifies the machine code generated for a read access; no masking operation is needed to zero bits that are not part of the value representation if store operations set them to zero.)

Trap representations, when they are supported, are invariably associated with scalar types only.

---

590 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.43)

Commentary

This is a requirement on the implementation. Operators operate on the value representation, not the object representation. Padding bits, which are one way a value can have more than one object representation, do not affect operations on the value representation. It is also possible for there to be more than value representation denoting the same value.

footnote 43

C90

This requirement was not explicitly specified in the C90 Standard.

C++

This requirement was added in C99 and is not explicitly specified in the C++ Standard (although the last sentence of 3.9p4 might be interpreted to imply this behavior).

Other Languages

Most languages do not get involved in specifying this level of representation detail.

Common Implementations

pointer segmented architecture

While processors that have a linear address space create few complications for vendors of translators, there are a number of reasons why hardware vendors might want to divide up a processor's address space into discrete *segments* (as always the driving forces are cost and performance). The most well-known segmented architecture is the Intel x86 processor family (whose early family members' 64 K segment size was considered generous for the first few years of their life). Segmented architectures did not die with the introduction of the Pentium. Some vendors of 32-bit processors have introduced 2^{32} segment sizes as a way of transitioning existing code to 64-bit address spaces. At the other end of the scale, low-cost embedded processors may have a flat address space, but support pointers capable of accessing only part of it (instructions involving such short-range pointers invariably occupy less storage and are often faster than other pointers).

The following discussion is based on techniques adopted by vendors of translators targeting the Intel 80286 member of the Intel x86 family and supporting pointers that handle various forms of segmented addressing. Some, or all, of these issues are likely to be applicable to other processor architectures. There are several ways instructions can specify an address in storage on the Intel 80286, including the following:

- *Specifying a 16-bit offset value.* This offset value is added to the value held in a segment register (the choice of segment register, one of four, is implicit in the instruction) to form an address in storage. This form is the simplest and fastest, but addressable storage was limited to a single 64 K segment. The value of a pointer object is the 16-bit offset (i.e., it is a 16-bit pointer).
- *Specifying a 16-bit pointer value and the segment register to use in forming the final address in storage.* This form does not have the 64 K restriction, but incurs the cost of having to generate code to load the chosen segment register with an appropriate value. The value of a pointer object has two parts — the 16-bit offset and the 16-bit value loaded into the segment register (i.e., it is a 32-bit pointer).

Pointers using the 16-bit representation are sometimes referred to as *near* pointers (after the keyword that is usually used in their declaration— e.g., `char near *p`); while pointers using the 32-bit representation are sometimes referred to as *far* pointers (after the keyword that is usually used in their declaration— e.g., `char far *p`).

Additional complications are caused by the fact that the two 16-bit components do not specify the value of a 32-bit address but the value in the 20-bit address space supported by the Intel 80286. The address is formed by left shifting the segment value by four and adding the offset to it (e.g., `(seg << 4) + offset`). This means that two different pairs of 16-bit values can specify the same address (e.g., the hexadecimal pairs *ABCD:0000* and *ABCC:0010* both specify the address *0xABCD0*). Additional complications arise because of vendors' desire to optimize the machine code generated for operations involving pointers. For instance, the machine code needed to compare or subtract two pointer values will depend on whether the operands are both near pointers (only need to deal with the offset on the basis that the segment register values will be the same) or some other combination (where both components need to be considered).

Using a normalized form for the representation of far pointers, where the bits from the two components don't overlap (e.g., arranging that the offset is always less than 0x10), allows equality and relational tests to be performed without having to add the two values together. However, the results obtained from pointer arithmetic have to be converted to a normalized form.

Developers writing programs that need to simultaneously contain more than 64 K of object data had to be very careful how they mixed any operations that involved the two kinds of pointers (the above discussion has been simplified and most implementations supported more than two kinds of pointers).

Some of the issues involved in the representation of the null pointer constant on a segmented architecture are discussed elsewhere.

null pointer conversion yields null pointer

591 Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

object representation more than one

Commentary

This is a requirement on the implementation. It ensures that a subsequent read operation can always access the value of the object without worrying that a previous store had stored a trap representation into it. There are ways of causing an object to hold a trap representation, but they all involve making use of undefined behavior. The standard is silent on the case where there is more than one possible value representation for the stored value.

C90

The C90 Standard was silent on the topic of multiple representations of object types.

C++

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

Common Implementations

The following discusses the case where there is more than one value representation for the same value. On segmented architectures, an implementation may define some canonical representation for pointer types (this has the advantage of simplifying some operations— e.g., pointer compare). All values are converted to this form when they are stored into a pointer object. An implementation may not exhibit the expected behavior if it reads a pointer value that does not use this canonical representation (which might be created by manipulating the object representation of an object having a pointer type).

590 pointer segmented architecture

592 **Forward references:** declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1).

References

1. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. *1990 International Conference on Supercomputing*, June 11-15 1990. Published as Computer Architecture News 18:3.
2. 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.
3. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??
4. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
5. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.