

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

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.2.6.2 Integer types

unsigned integer types object representation

For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). 593

### Commentary

Padding bits can be used for several reasons:

- A value representation may be stored in a larger object representation. This can occur if a processor only has a single load and store instruction, which always operate on a fixed number of bits.
- The host processor uses additional bits to indicate properties of the object representation; for instance, parity of the stored value, or tagged data, where the tag specifies type stored at that location.
- Representations are shared between different types (either to reduce processor transistor count, or for backward compatibility with existing code); for instance, integer types and floating-point types, or integer types and pointer types.

### C90

Explicit calling out of a division of the bits in the representation of an unsigned integer representation is new in C99.

### C++

Like C90 the grouping of bits into value and padding bits is not explicitly specified in the C++ Standard (3.9p2, also requires that the type **unsigned char** not have any padding bits).

### Other Languages

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

### Common Implementations

On some Cray processors the type **short** has 32 bits of precision but is held in 64 bits worth of storage.

The Unisys A Series unsigned integer type contains a padding bit that is treated as a sign bit in the signed integer representation (see Figure ??).

### Coding Guidelines

Programs whose behavior depends on the value of padding bits make use of representation information. The fact that modern processors rarely contain padding bits in their integer types does not mean that future processors will continue this trend. The main developer assumption that fails in the presence of padding bits is that there are  $\text{CHAR\_BIT} * \text{sizeof}(\text{integer\_type})$  bits in the value representation. The only reliable way of finding out the number of bits in the value representation is to look at the  $\text{MIN\_}$  and  $\text{MAX\_}$  macros defined in the header `<limit.h>`.

word addressing

representation information using

numerical limits

If there are  $N$  value bits, each bit shall represent a different power of 2 between 1 and  $2^{N-1}$ , so that objects of that type shall be capable of representing values from 0 to  $2^N - 1$  using a pure binary representation; 594

### Commentary

This is a requirement on the implementation. Representations ruled out by this requirement include BCD for integer types (some processors, including the Intel x86 processor family, contain instructions for operating on this representation) and Gray codes (consecutive decimal values are represented by binary values that differ in the state of a single bit; there is no requirement that ones start in the least significant bit and percolate up, so there are many possible bit sequences that are Gray codes).

**Table 594.1:** Pattern of bits used to represent decimal numbers using various coding schemes.

Decimal	Binary	Gray code	111 biased	2-out-of-5
0	0000	0000	0111	00011
1	0001	0001	1000	00101
2	0010	0011	1001	00110
3	0011	0010	1010	01001
4	0100	0110	1011	01010
5	0101	0111	1100	01100
6	0110	0101	1101	10001
7	0111	0100	1110	10010
8	1000	1100	1111	10100
9	1001	1101		11000
10	1010	1111		
11	1011	1110		
12	1100	1010		
13	1101	1011		
14	1110	1001		
15	1111	1000		

This requirement means there has to be a sequential ordering of the bits used in the value representation. A sequential ordering of bits means that the shift operators cause a uniform pattern of bit movement. For instance, the following two functions are equivalent:

shift-expression  
syntax

```

1  #include <limits.h>
2
3  /*
4   * This code assumes that sizeof(int) == 2
5   */
6
7  unsigned int obj_htons(unsigned int h)
8  {
9  /*
10 * Relies on type unsigned char copying all bits.
11 */
12 unsigned char *hp = (unsigned char *)&h;
13 unsigned int n;
14 unsigned char *np = (unsigned char *)&n;
15
16 np[0] = hp[1];
17 np[1] = hp[0];
18 return n;
19 }
20
21 unsigned int val_htons(unsigned int h)
22 {
23 #define MASK ((1U << CHAR_BIT) - 1)
24 return ((h & MASK) << CHAR_BIT) | ((h >> CHAR_BIT) & MASK);
25 }

```

Requirements on the value bits of signed integer types are phrased in terms of the value of the corresponding <sup>600</sup>value bits signed/unsigned bits in the corresponding unsigned integer type. There is no requirement that other non-integer scalar types be represented using a binary notation.

## C90

These properties of unsigned integer types were not explicitly specified in the C90 Standard.

## Other Languages

Even those languages that contain an unsigned integer type do not specify this level of detail.

### Coding Guidelines

When targeting hosts with relatively low-performance processors, there are advantages to making use of the mathematical properties of a pure binary representation— for instance, using the shift operator to replace multiplies and divides of positive values by powers of two; or using the sequence add, shift and add for multiplication by 10. A number of arguments against this usage are often heard:

- Performing these kinds of optimizations based on representation details at the source code level creates a dependency on that representation. However, the standard specifies what the representation must be.
- It is claimed that an optimizing translator would perform the mapping to shift and add instructions, and that developers should leave this kind of optimization to the translator. However, not all optimizers live up to their name.
- Using such constructs makes the code more difficult to comprehend. Is an expression that uses a shift operator, for instance, harder to comprehend than one that uses a multiply or divide? It is really a question of familiarity. Developers who frequently encounter this usage learn to quickly recognize it and comprehend its purpose. This is not true for developers that rarely encounter this usage.

The general issue is one of developer overconfidence and comprehension. Overconfidence, or perhaps tunnel vision, by the original developer in believing that an optimization performed at any point in the source will impact the efficiency of the program. Subsequent readers are likely to require greater cognitive resources to comprehend the use of sequences of bit manipulation operations that are intended to mimic the effects of an arithmetic operator. These issues are discussed in more detail in the respective operator sentences.

One issue, which is rarely considered, is the impact of bit manipulation on optimizers and static analysis tools. These do not usually analyze bit manipulation operations in terms of their arithmetic effect. They tend to treat these operations as creating a sequence of bits of unknown value. As such, these operations reduce the information that optimizers and static analysis tools are able to deduce from the source.

value representa-  
tion

this shall be known as the value representation.

595

#### Commentary

This defines the term *value representation* (it was first used in the response to DR #069). Given that, for most implementations the value representation and the object representation are bit-for-bit identical, it seems unlikely that this term will become commonly used by developers.

unsigned integer  
padding bit values

The values of any padding bits are unspecified.<sup>44)</sup>

596

#### Commentary

This is a special case of a more general one specified elsewhere. A strictly conforming program cannot access the value of these bits. Given that programs are not intended to access the values of these padding bits, there is no obvious reason for specifying the value as being implementation-defined. Such a specification would have required implementations to document their behavior in this area, which is likely to vary between objects in different contexts and be generally very difficult to specify for all cases.

#### C90

Padding bits were not discussed in the C90 Standard, although they existed in some C90 implementations.

#### C++

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

#### Common Implementations

Some hosts have one or more parity bits associated with each storage byte. The purpose is to provide confidence that storage has not been corrupted (to at least the additional degree of probability provided by the number of parity bits used) or even corrected hardware errors (if enough parity bits are available). These parity bits are normally handled by the storage hardware and are not visible at the software level. (It is likely that the processor knows nothing about how the bytes in storage are handled.)

integer<sup>622</sup>  
padding bit values

**Coding Guidelines**

The guideline recommendation dealing with the use of representation information applies to access to padding bits.

?? representation information using

597 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit.

signed integer types object representation

**Commentary**

The difference in the representation, from unsigned integer types, is that there is a sign bit. An alternative representation of signed numbers, not using a sign bit, is to use biased notation. This method of representation is used for the exponent in the IEC 60559 floating-point standard.

593 unsigned integer types object representation

IEC 60559

**Other Languages**

All known computer languages can represent signed integer types, so their implementations have some means of representing the sign. But language specifications do not usually go into this level of detail.

**Common Implementations**

This division of the object representation into three groups makes it a superset of the representations used by all known processors (many do not have padding bits). The Harris/6 computer represented the type **long** using two consecutive **int** types. This meant that the sign bit of one of the ints had to be ignored; it was treated as a padding bit. The value representation of the type **int** is 24 bits wide, and **long** had a value representation of 47 bits with one padding bit. The Unisys A Series uses a representation for its signed integer types that contain all three groups (see Figure ??). Also, the sign bit is not part of the value representation of its unsigned integer types.

598 There need not be any padding bits;

**Commentary**

The standard places no upper limit on the number of padding bits that may exist in an integer representation.

**Common Implementations**

Most implementations have no padding bits.

599 there shall be exactly one sign bit.

sign one bit

**Commentary**

Having specified that there is a sign bit, the standard needs to say something about it. The encoding of the sign bit limits the number of possible integer representations to three (when a binary representation is used).

610 sign bit representation

**C90**

This requirement was not explicitly specified in the C90 Standard.

**C++**

*[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types. ]*

3.9.1p7

These three representations all have exactly one sign bit.

**Other Languages**

Cobol provides several different arithmetic types. One of these types requires a representation based on using the Ascii character set values for the digits and the sign. This sign representation not only occupies more than one bit, but where it occurs in the object representation can vary. (It can occupy a single byte that leads or trails the digits, or it can be encoded as part of the value representation of the leading or trailing digit.)

value bits  
signed/unsigned

Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are  $M$  value bits in the signed type and  $N$  in the unsigned type, then  $M \leq N$ ).

### Commentary

This is a requirement on the implementation. The value bits in the signed integer representation must represent the same power of two as the corresponding bit in the unsigned representation. From this we can deduce that, provided there are no padding bits in the object representation of the unsigned type, the sign bit is the most significant bit.

The relational condition on the number of bits in each value representation is specified in terms of ranges of values elsewhere.

### C90

This requirement is new in C99.

### C++

positive  
signed in-  
teger type  
subrange of  
equivalent  
unsigned type

- 3.9.1p3 *The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.*

If the value representation is the same, the value bits will match up. What about the requirement on the number of bits?

- 3.9.1p3 *... each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type<sup>40)</sup>; that is, each signed integer type has the same object representation as its corresponding unsigned integer type.*

Combining these requirements with the representations listed in 3.9.1p7, we can deduce that C++ has the same restrictions on the relative number of value bits in signed and unsigned types.

### Other Languages

Most languages do not contain both signed and unsigned types. Those languages that do, do not get involved in this level of detail.

### Common Implementations

In most implementations there is one more bit in the value representation of an unsigned type than in its corresponding signed type (the sign bit). The Unisys A Series (see Figure ??) is the only processor known to your author where the number of value bits in both the signed and unsigned types are the same.

42) Thus, for example, structure assignment may be implemented element at a time or via memcpy. need not copy any padding bits.

### Commentary

The wording was changed by the response to DR #222, which also contained the following Committee discussion:

- DR #222 *It was observed that the point of the original footnote was primarily to illustrate one reason why padding bits might not be copied: because member-by-member assignment might be performed. But member-by-member assignment would imply that struct assignment could produce undefined behavior if a member of the struct had a value that was a trap representation. Instead of adding further text explaining that member values that were trap representations were not permitted to render assignment of a containing struct or union object undefined (e.g., if member-by-member copying were used), it was decided that the footnote should simply clarify the issue of padding bits directly.*

footnote  
42

**C90**

The C90 Standard did not explicitly specify that padding bits need not be copied.

**C++**

36) By using, for example, the library functions (17.4.1.2) `memcpy` or `memmove`.

Footnote 36

The C++ Standard does not discuss details of structure object assignment for those constructs that are supported by C. However, it does discuss this issue (12.8p8) for copy constructors, a C++ construct.

**Example**

```

1  #include <stdio.h>
2
3  struct T_1 {
4      int mem_1;
5      unsigned : 4;
6      int mem_2: 2;
7      double mem_3;
8  };
9  struct T_2 {
10     int mem_1;
11     unsigned int mem_name: 4;
12     int mem_2: 2;
13     double mem_3;
14 };
15 union T_3 {
16     struct T_1 su; /* First named member is initialized. */
17     struct T_2 sn; /* Both structure types have a common initial sequence. */
18 } gu;
19 struct T_1 s;
20
21 int main(void)
22 {
23     union T_3 lu;
24
25     lu.su = s;
26
27     if (lu.sn.mem_name != 0)
28         print("Whether this string is output is unspecified\n");
29     if (gu.sn.mem_name != 0)
30         print("Whether this string is output is unspecified\n");
31 }

```

602 43) It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts.

footnote  
43**Commentary**

For this situation to occur, the two objects must contain different bit patterns. But, how can two objects containing different sequences of bits ever be considered to contain the same value?

- When there are padding bits in the object representation. They are padding bits in the sense of not being part of the value representation of the object's effective type. When the object is treated as, for [effective type](#) instance, an array of **unsigned char**, the padding bits are included in its value.

`x == y`<sup>604</sup>  
 x not same as y  
 pointer  
 segmented  
 architecture

- When two different sequences of bits are interpreted using some type to have the same value; for instance, plus and minus zero in signed magnitude or IEC 60559 floating-point. Also, in some implementations, two different sequences of bits can represent the same address in storage.

## C90

This observation was not pointed out in the C90 Standard.

## C++

The C++ Standard does not make a general observation on this issue. However, it does suggest that such behavior might occur as a result of the `reinterpret_cast` operator (5.2.10p3).

## Other Languages

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

## Common Implementations

Very few implementations have padding bits in the object representation of arithmetic types. Most implementations use two's complement notation for integer types. Each value has a unique *value representation* in this notation.

In the past some processors have used 32 bits in the object representation of pointer types, but used fewer bits in their value representation. As the price of memory chips dropped and customers demanded support for greater amounts of storage capacity, vendors upgraded their processors to use the full 32 bits of representation. Processors using 64 bits in their pointer type object representation are now available, but it is expected that it will be some years before memory chip prices reach the point where this capability will be fully utilized. At the time of this writing some processors only use 48 bits effectively making the remaining 16 bits padding.

On segmented architectures, there is often more than one sequence of bits (pointer value representations) capable of denoting the same address.

## Coding Guidelines

This footnote highlights the dangers of developers becoming involved in the representation details of objects.

## Example

```

1  _Bool negative_zero_may_have_more_than_one_representation(int valu)
2  {
3  if (valu == 0)
4      if (value & 1) /* Representation used affects the result here. */
5          return 1;
6  return 0;
7  }
```

---

In particular, if `==` is defined for type `T`, then `x == y` does not imply that `memcmp(&x, &y, sizeof(T)) == 0`. 603

## Commentary

The equality operators are not defined for operands having structure or union types, so padding bytes are not an issue here. It is also true that

- `x != y` does not imply `memcmp(&x, &y, sizeof(T)) != 0` (this case occurs if both `x` and `y` have the same NaN value);
- `(x-y) == 0` does not imply `x == y` (it need not apply for nonzero values of `x` and `y` if an implementation does not support subnormals);
- on a segmented architecture, there may be more than one representation of the null pointer constant; this is discussed elsewhere.

equality  
 operators  
 constraints

subtraction  
 result of  
 subnormal  
 numbers

pointer  
 segmented  
 architecture

## Other Languages

Very few languages define a function that is the equivalent of `memcmp`, so this is rarely an issue. However, some implementations of these languages provide an equivalent function, although such implementations do not usually go into the implications of its use.

## Coding Guidelines

Calling `memcmp` is making use of representation information, even though developers do not see it in that light. The guideline recommendation dealing with the use of representation information is applicable. Given that C does not contain support for objects having structure or union types as operands of the equality operators, use of `memcmp` has some attractions. Used in conjunction with the `sizeof` operator, it does not need modification when new members are added to the types. However, padding bytes do need to be taken into account. ?? representation information using

604 Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; x == y  
x not same as y

## Commentary

The values `-0.0` and `+0.0` are different values (dividing by these values results in  $-\infty$  and  $+\infty$ , respectively), but they compare equal. If `x` and `y` have different integer types, there are several different cases where they can compare equal and have different values; for instance (assuming the type `int` is 16 bits and the type `long` is 32 bits), the values `-1` and `0xffff` compare equal. equality operators  
true or false

For pointer types, equality is defined in terms of them pointing to the same object. pointers  
compare equal

## Other Languages

The values that create this behavior in C are caused by the underlying representations used by the host processor. As such, the cases are applicable to all languages supporting the same C representations that are translated for execution on that processor.

Some languages provide more than one mechanism for comparing for equality. This issue is discussed elsewhere. equality operators  
syntax

## Coding Guidelines

Modern processors have a single representation for integer zero. If the guideline recommendation dealing with comparing objects having floating-point types for equality is followed, there will not be any issues for those types. ?? equality operators  
not floating-point operands

## Example

```

1  #include <stdio.h>
2
3  extern double x, y;
4
5  void f(void)
6  {
7  if (x == y)
8      if ((1.0 / x) != (1.0 / y))
9          printf("x and y are differently signed zeros\n");
10 }
11
12 void g(void)
13 {
14 unsigned int x = 0xffff;
15 signed int y = -1;
16 unsigned long z = 0xffff;
17
18 if (x == y)
19     if ((x == z) == (y == z))
20         printf("int and long probably have the same representation\n");
21     else

```

```

22     printf("int and long probably have a different representation\n");
23 }

```

---

other operations on values of type **T** may distinguish between them.

605

### Commentary

In the case of integer types represented using signed magnitude or one's complement notation, there are two representations for zero,  $-0$  and  $+0$ . Arithmetic operations always deliver the same result. In the case of bitwise operations, their behavior on signed operands is undefined, so the issue of two representations is moot.

bitwise operations  
signed types

There are also two representations of zero,  $-0.0$  and  $+0.0$ , in IEC 60559. A great deal of thought lay behind the decision to have a signed zero and the results of arithmetic operations on them (in some cases the result depends on the sign, while in others it does not). For instance, dividing by  $-0$  returns  $-\infty$  and dividing by  $+0$  returns  $+\infty$  (they may produce the same results using other floating-point models), but adding a nonzero value always delivers the same result. The `copysign` library function will also return different results.

floating types  
can represent

signed  
of non-numeric values

### Coding Guidelines

This situation usually arises when there is more than one representation of zero. This occurs for integer types represented in one's complement and signed magnitude format, or the floating-point representation in IEC 60559. The number of defined operations that can be performed using pointer types is sufficiently limited that multiple representations are not an issue.

In the case of IEC 60559 the decision to have signed zeros was intended to be an aid to the developer writing numerical software. If unexpected behavior occurs in this case, the cause is likely to be developer misunderstanding of the mathematics involved, which is outside the scope of these coding guidelines.

---

44) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit.

606

### Commentary

The trap representations that occur because of parity errors are usually caused by hardware rather than software faults, while parity bits are not limited to those supported by hardware. It is rare for an implementation to choose to use one or more bits for this purpose.

This footnote is identical to footnote 45.

### C90

This footnote is new in C99.

### C++

This wording was added in C99 and is not explicitly specified in the C++ Standard. Trap representations in C++ only apply to floating-point types.

### Common Implementations

Processors that contain a trap representation for integer types (apart from parity bits, which are not usually visible at the program level) are rare. At least one processor includes a parity bit in its floating-point representation.

WE DSP32

### Coding Guidelines

Any program that accesses a subpart of an object to manipulate padding bits is making use of representation information and is covered by a guideline recommendation.

representation information  
using

---

Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types.

607

arithmetic operation  
exceptional condition

**Commentary**

This statement excludes arithmetic operations, not bitwise operations. While divide by zero (or the remainder operation) often raises an exception, the standard specifies the result of such an operation as being undefined. divide by zero

**C++**

This discussion on trap representations was added in C99 and is not explicitly specified in the C++ Standard.

**Common Implementations**

While various processor status flags might be set as a result of an arithmetic operation, they are not part of the representation of a value.

608 All other combinations of padding bits are alternative object representations of the value specified by the value bits.

**Commentary**

The object representation includes all bits in the storage allocated to the object, while the value representation may be a subset of these bits. The standard treats the combination of the value bits and the different values of the padding bits as alternative object representations.

**C++**

This discussion of padding bits was added in C99 and is not explicitly specified in the C++ Standard.

609 If the sign bit is zero, it shall not affect the resulting value.

**Commentary**

This is a requirement on the implementation. It reflects existing processor integer representation practice.

**C90**

This requirement was not explicitly specified in the C90 Standard.

**C++**

*[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types. ]*

3.9.1p7

In these three representations a sign bit of zero does not affect the resulting value.

**Other Languages**

Other languages do not usually go into the representation details of their supported integer types. However, implementations of these languages will target the same hosts as C implementations. Unless a nonbinary representation is used to represent integer types, these implementations will follow the C model.

610 If the sign bit is one, the value shall be modified in one of the following ways:

sign bit  
representation**Commentary**

This is a requirement on the implementation. These three ways correspond to the three representations of binary notation in common (well one of them is) usage.

**C90**

This requirement was not explicitly specified in the C90 Standard.

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

sign and  
magnitude



### Common Implementations

The CDC 6600, 7600, and Cyber 200 are the only known (to your author) commercially sold processors (no longer available) using one's complement representation that supported a C translator.<sup>[1]</sup>

614 Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for one's complement), is a trap representation or a normal value.

### Commentary

Implementations with such trap representations are thought to have existed in the past. Your author was unable to locate any documents describing such processors.

### C90

The choice of representation for signed integer types was not specified as implementation-defined in C90 (although annex G.3.5 claims otherwise). The C90 Standard said nothing about possible trap representations.

### C++

The following suggests that the behavior is unspecified.

*The representations of integral types shall define values by use of a pure binary numeration system<sup>44</sup>. [Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types. ]*

3.9.1p7

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

### Other Languages

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

### Common Implementations

The Unisys A Series<sup>[4]</sup> uses two's complement for character types and a sign and magnitude representation for the other integer types.

### Coding Guidelines

The use of two's complement, by commercially available processors, is almost universal. Is there any worthwhile benefit in writing programs to execute correctly using all three representations?

Calculating the cost/benefit requires estimating the probability of having to port to a non-two's complement processor and the costs of writing code that has the same behavior for all integer representations likely to be used. There is only one commercially available processor (known to your author) that does not use two's complement, the Unisys A Series.<sup>[3]</sup> The probability of having to port code to this platform can only be calculated by individual development groups. Because of the likelihood that the overwhelming volume of existing code contains some representation dependencies, there is every incentive for hardware vendors to continue to use two's complement in new processors.

There are two ways a program can depend on details of the representation— by explicitly manipulating the bits of the value representation using bitwise operators, and by generating values that are not guaranteed to be representable in all implementations of an integer type. Manipulating bits rather than numeric values is generally recommended against for reasons other than portability to other representations.

Is simply specifying use of a two's complement representation sufficient? The minimum integer limits specified by the C Standard are symmetrical. However, all two's complement implementations (known to your author) are not symmetrical. They support one more negative value than positive value. Writing programs that only make use of a symmetrical set of positive and negative values could involve considerable cost. In this situation a program is not manipulating representations details, but having to deal with the consequences of a particular representation. Given the rarity of two of the possible integer representations being encountered, and overwhelming support for an asymmetric implementation of two's complement notation, the following is suggested:

types  
representation

integer types  
sizes

- One's complement or sign and magnitude issues should only be dealt with when writing for such implementations.
- The asymmetric nature of existing two's complement implementations should be accepted and no attempts made to handle the *extra* negative value any differently than the other values.

negative zero

---

In the case of sign and magnitude and one's complement, if this representation is a normal value it is called a *negative zero*. 616

### Commentary

This defines the term *negative zero*. The term *zero* is commonly used to imply the zero value with the sign bit also set to zero. The term *positive zero* is sometimes used to help distinguish this case when discussing both representations of zero.

### C90

The C90 Standard supported hosts that included a representation for negative zero, however, the term *negative zero* was not explicitly defined.

### C++

The term *negative zero* does not appear in the C++ Standard.

### Other Languages

This terminology is also used in Cobol, where negative zero is always supported irrespective of the underlying host representation of binary integers.

negative zero  
only generated by

---

If the implementation supports negative zeros, they shall be generated only by: 616

### Commentary

This is a requirement on the implementation. In the case of arithmetic operations, unless one of the operands has a value of negative zero, the operation does not deliver a negative zero result. The principle of *least surprise* (sometimes known as the *Principle of Least Astonishment*) could be said to be the driving principle here.

This requirement only applies to integer types and the list excludes all operators whose result is not derived by manipulating the operands (e.g., relational operators, comparison operators, or logical operators, which are all defined to return a value of 0 or 1). For these operators, an implementation cannot choose to use a negative zero value.

Negative zero can also be created as the result of a conversion operation.

### C90

The properties of negative zeros were not explicitly discussed in the C90 Standard.

### C++

This requirement was added in C99; negative zeros are not explicitly discussed in the C++ Standard.

### Other Languages

Cobol also supports a negative zero under some conditions.

### Example

```

1  _Bool is_negative_zero(int valu)
2  {
3  if (valu == 0)
4      if (value & ~0)
5          return 1;
6  return 0;
7  }
```

floating-point  
converted  
to integer

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

bitwise operators  
negative zero

### Commentary

The bitwise operators operate directly on the individual bits of the value representation of integer types. If negative zero is supported by an implementation, there is the possibility that any of them could generate the bit pattern representing this value (starting from a bit pattern that does not represent negative zero). The use of bitwise operators on values having a signed integer type and exhibit implementation-defined and undefined behaviors (Also the standard does not guarantee that a negative zero value can be stored into an object without a change to its representation.)

615 negative zero

bitwise op-  
erations  
signed types  
620 negative zero  
storing

### Other Languages

Cobol does not include these operators.

### Common Implementations

The Unisys A Series<sup>[4]</sup> uses signed magnitude representation. If the operands have unsigned types, the sign bit is not affected by these bitwise operators. If the operands have signed types, the sign bit does take part in bitwise operations.

### Example

The following will produce a negative zero on an implementation that uses a sign and magnitude representation; for implementations that use other representations the behavior is undefined.

```
1 int unsigned_zero(void)
2 {
3     return (-1) & (-2);
4 }
```

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

### Commentary

The `+` and `-` operators exist in unary and binary form. The phrase *where one argument* could be taken to imply two operands (i.e., only the binary operator are intended). In the case of unary minus this requirement means that `-0` does not represent negative zero. In fact only negating a negative zero can return a result of negative zero.

615 negative zero

619— compound assignment operators based on the above cases.

### Commentary

This is specified to ensure that all the cases are covered. Compound assignment is defined in terms of its equivalent binary operator.

compound  
assignment  
semantics

### Other Languages

Cobol does not contain compound assignment operators.

### Common Implementations

Compound assignment is commonly implemented using the associated operator followed by an assignment. Although processors having instructions capable of operating directly on the contents of storage are no longer common (e.g., the Motorola 680x0 family<sup>[2]</sup>), implementations are not required to operate directly on the contents of storage for this permission to apply.

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

negative zero  
storing

**Commentary**

It is unspecified because developers may not have any control over what happens and the combinations of circumstances may be too disparate for an implementation to document. For instance, in one case a value may be operated on in a register before being written to storage (so any conversion that may be carried out by the store will not have happened), while in another case the same value may be operated on directly in its storage location. It is possible that a store operation, treating the contents of a register as signed, may convert the value to another kind of zero.

**C90**

This unspecified behavior was not called out in the C90 Standard, which did not discuss negative integer zeros.

**C++**

Negative zero is not discussed in the C++ Standard.

**Other Languages**

Cobol specifies when a negative zero is generated. For the object types involved, there is never any question of a negative zero being converted to any other value when it is written to storage.

**Common Implementations**

The Unisys A Series<sup>[4]</sup> uses a sign and magnitude representation and does not copy the sign bit when storing a value having an **unsigned int** type.

**Coding Guidelines**

The surprising effects of negative zeros is something that will need to be brought to developers' attention if a processor with this characteristic is ever encountered. However, implementations that support more than one integer representation of zero are very rare. For this reason no recommendations are made.

---

If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with arguments that would produce such a value is undefined. 621

**Commentary**

sign and  
magnitude  
bitwise op-  
erators  
negative zero

On processors that use a sign and magnitude representation the behavior of the code `(-1) & (-2)` will either be acceptable to an implementation or result in undefined behavior.

**C90**

This undefined behavior was not explicitly specified in the C90 Standard.

**C++**

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

---

The values of any padding bits are unspecified.<sup>45)</sup> 622

**Commentary**

This duplicates a sentence given earlier. The earlier sentence could be read to apply to unsigned integer types only. This sentence applies to all integer types.

---

A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. 623

**Commentary**

This is a requirement on the implementation. Two corresponding integer types must use the same number of bytes in their object representation. The value bits of a signed integer type are required to be a subset of the corresponding unsigned type and these value bits are at the same relative bit positions. If a signed integer object contains a positive value it can be copied, using `memcpy`, into an object having the corresponding

unsigned integer type and the two value representations will compare equal. The signed integer type contains a sign bit, which may be a value bit in the corresponding unsigned integer type (in two's complement implementations), or it may not (in some sign and magnitude implementation— e.g., the Unisys A Series<sup>[4]</sup>). It is possible for the unsigned type to have more than one additional value bit. An earlier sentence deals with the value representation.

600 value bits  
signed/unsigned  
positive  
signed in-  
teger type  
subrange of equiv-  
alent unsigned  
type  
footnote  
31  
alignment

The interchangeability discussed in footnote 31 does not mention pointers to corresponding signed/unsigned integer types. The preceding requirement, along with the alignment requirements, is sufficient to deduce that pointers to corresponding signed/unsigned integer types can be used to access positive values held in the pointed-to objects.

### C90

There was no such requirement on the object representation in C90, although this did contain the C99 requirement on the value representation.

positive  
signed in-  
teger type  
subrange of equiv-  
alent unsigned  
type

### C++

*... ; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.*

3.9.1p3

### Example

This requirement guarantees that:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  signed short ss;
5  unsigned short us;
6
7  void f(void)
8  {
9  memcpy(&us, &ss, sizeof(ss));
10
11  if ((long)us == (long)ss)
12     printf("Value of ss is positive or (sizeof(short) == sizeof(long))\n");
13  }
```

is defined if the value of `ss` is positive.

---

624 For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

### Commentary

This sentence was added by the response to DR #263.

#### **Problem**

DR #263

Consider the code:

```

int v [10];
memset (v, 0, sizeof v);
```

*Most programmers would expect this code to set all the elements of `v` to zero. However, the code is actually undefined: it is possible for `int` to have a representation in which all-bits-zero is a trap representation (for example, if there is an odd-parity bit in the value).*

**C90**

The C90 Standard did not specify this requirement.

precision  
integer type

---

The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. 625

**Commentary**

This defines the term *precision* (when applied to integer types). This term is also used in the definition of the I/O conversion specifiers. The term *precision* has a common usage meaning in the sense of the accuracy of a measurement. It is also used in the context of floating types. Values or objects having a floating type are commonly referred to as being single- or double-precision.

precision  
floating-point

**C90**

The definition of this term is new in C99.

**C++**

The term *precision* was added in C99 and is only defined in C++ for floating-point types.

**Other Languages**

The term *precision* is used in several different languages to mean a variety of different things.

**Coding Guidelines**

The term *width* is often used by developers when discussing the number of bits used to represent integer types. Use of the term *precision*, in an integer type context is limited to only a handful of locations in the C Standard. There are no obvious advantages to training developers to use this term in the context of integer types.

width  
integer type 626

---

The *width* of an integer type is the same but including any sign bit; 626

**Commentary**

This defines the term *width* (when applied to integer types), *width* = *precision* + *one\_if\_there\_is\_a\_sign\_bit*. If the width of type A is less than the width of type B, then type A is said to be *narrower* than type B.

narrower type  
narrower  
example

**C90**

The definition of this term is new in C99.

**C++**

The term *width* was added in C99 and is not defined in the C++ Standard.

**Other Languages**

The term *width* is used in several different languages to mean a variety of different things.

**Coding Guidelines**

Many terms are commonly used by developers to describe the number of bits in an integer type, including the term *number of bits*. Because there is no commonly used term, guideline documents will need to fully define the terms used. The term *width* has the advantage of being defined by the C Standard and fitting with the usage of the term *narrow*.

---

thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision. 627

**Commentary**

The standard does not require the precision of integer types having the same rank (i.e., a signed integer type and its equivalent unsigned form) to be the same. However, in practice they are the same in the majority of implementations.

rank  
standard in-  
integer types

---

628 45) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit.

footnote  
45**Commentary**

This footnote duplicates footnote 44.

606 footnote  
44

---

629 Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow.

**Commentary**

This footnote duplicates footnote 44.

606 footnote  
44

---

630 All other combinations of padding bits are alternative object representations of the value specified by the value bits.

**Commentary**

This footnote duplicates footnote 44.

606 footnote  
44

## References

1. K.-C. Li and H. Schwetman. Implementing a scalar C compiler on the Cyber 205. *Software—Practice and Experience*, 14(9):867–888, 1984.
2. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
3. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.
4. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.