



INTERNATIONAL STANDARD ISO/IEC 9899:1999
TECHNICAL CORRIGENDUM 2

Published 2004-11-15

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Programming languages — C

TECHNICAL CORRIGENDUM 2

Langages de programmation — C

RECTIFICATIF TECHNIQUE 2

Technical Corrigendum 2 to ISO/IEC 9899:1999 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

1. *Page 4, 3.4.4*

Prepend to paragraph 1:

use of an unspecified value, or other

before:

behavior where

2. *Page 15, 5.1.2.3*

In paragraph 12, last line in the code fragment, change:

expressions

to:

expression

3. *Page 19, 5.2.1.2*

In paragraph 1, replace items 4 and 5 with:

- A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.

4. *Page 23, 5.2.4.2.2*

Add a new paragraph after paragraph 3:

An implementation may give zero and non-numeric values (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

5. *Page 24, 5.2.4.2.2*

Change paragraph 4 to:

The accuracy of the floating-point operations (+, -, *, /) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library routines in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown.

6. *Page 38, 6.2.6.1*

Change paragraph 6 to:

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⁾ The value of a struct or union object is never a trap representation, even though the value of a member of a struct or union object may be a trap representation.

7. *Page 38, 6.2.6.1*

Change footnote 42 to:

Thus, for example, structure assignment need not copy any padding bits.

8. *Page 38, 6.2.6.1*

Change paragraph 7 to:

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.

9. *Page 39, 6.2.6.2*

Append to paragraph 5:

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

10. *Page 43, 6.3.1.1*

Change paragraph 2, the first item, to:

An object or expression with an integer type whose integer conversion rank is less than or equal to the rank of `int` and `unsigned int`.

11. *Page 50, 6.4.1*

Append to paragraph 2:

The keyword `_Imaginary` is reserved for specifying imaginary types.

12. *Page 50, 6.4.1*

Add a footnote to the last sentence of paragraph 2:

One possible specification for imaginary types appears in Annex G.

13. Page 87, 6.5.9

Add the following text to a new paragraph at the end of the *Semantics* section:

For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

14. Page 92, 6.5.16.1

Change footnote 93 to:

The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **int volatile * const**).

15. Page 99, 6.7.2

Delete "**_Imaginary**" from paragraph 1.

16. Page 100, 6.7.2

Delete from paragraph 2:

- **float _Imaginary**
- **double _Imaginary**
- **long double _Imaginary**

17. Page 100, 6.7.2

Replace paragraph 3 with:

The type specifier **_Complex** shall not be used if the implementation does not provide complex types.¹⁰¹⁾

18. Page 100, 6.7.2

Replace footnote 101 with:

Freestanding implementations are not required to provide complex types.

19. Page 101, 6.7.2.1

Change the first sentence of paragraph 3 to:

The expression that specifies the width of a bit-field shall be an integer constant expression that has nonnegative value that shall not exceed the width of an object of the type that would be specified were the colon and expression omitted.

20. *Page 103 6.7.2.1*

In paragraph 16, replace the second, third and forth sentences with:

In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array.

21. *Page 103 6.7.2.1*

In paragraph 16, remove footnote 106.

22. *Page 103 6.7.2.1*

Replace paragraph 17 with:

EXAMPLE After the declaration:

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

the structure `struct s` has a flexible array member `d`. A typical way to use this is:

```
int m = /* some value */;
struct s *p = malloc(sizeof(struct s) + sizeof (double [m]));
```

and assuming that the call to `malloc` succeeds, the object pointed to by `p` behaves, for most purposes, as if `p` had been declared as:

```
struct { int n; double d[m]; } *s1;
```

(there are circumstances in which this equivalence is broken; in particular, the offsets of member `d` might not be the same).

23. *Page 104 6.7.2.1*

Replace paragraph 18 with:

Following the above declaration:

```
struct s t1 = { 0 };           // valid
struct s t2 = { 1, { 4.2 } }; // invalid
t1.n = 4;                     // valid
t1.d[0] = 4.2;                 // might be undefined behavior
```