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*.

ICS 35.060 **Ref. No. ISO/IEC 9899:1999/Cor.2:2004(E)**

Published in Switzerland

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
> ```

The initialization of `t2` is invalid (and violates a constraint) because `struct s` is treated as if it did not contain member `d`. The assignment to `t1.d[0]` is probably undefined behavior, but it is possible that

```
sizeof (struct s) >= offsetof (struct s, d) + sizeof (double)
```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

24. *Page 104 6.7.2.1*
Replace paragraph 19 with:

After the further declaration:

```
struct ss { int n; };
```

the expressions:

```
sizeof (struct s) >= sizeof (struct ss)
sizeof (struct s) >= offsetof (struct s, d)
```

are always equal to 1.

25. *Page 104 6.7.2.1*
Replace paragraph 20 with:

If `sizeof (double)` is 8, then after the following code is executed:

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

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

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

26. *Page 104 6.7.2.1*
Add a new paragraph 21:

Following the further successful assignments:

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


they then behave as if the declarations were:

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

and:

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

27. *Page 104 6.7.2.1*
Add a new paragraph 22:

The assignment:

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

only copies the member **n**; if any of the array elements are within the first **sizeof (struct s)** bytes of the structure, they might be copied or simply overwritten with indeterminate values.

28. *Page 119, 6.7.5.3*
Change paragraph 11 to:

If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

29. *Page 146, 6.10*
In paragraph 2, italicize the first use of the term "preprocessing directive".

30. *Page 148, 6.10.1*
Change paragraph 3, starting mid third sentence, to:

… and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they had the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header **<stdint.h>**.

31. *Page 148, 6.10.1*
Add a footnote to the end of paragraph 3:

Thus, on an implementation where **INT_MAX** is **0x7FFF** and **UINT_MAX** is **0xFFFF**, the constant **0x8000** is signed and positive within a **#if** expression even though it would be unsigned in translation phase 7.

32. *Page 152, 6.10.3*
In paragraph 11, last sentence, after "preprocessing directives", add a new footnote:

> Despite the name, a non-directive is a preprocessing directive.

33. *Page 161, 6.10.8*
Change paragraph 2, item 3, to:

> **__STDC_ISO_10646__** An integer constant of the form **yyyymmL** (for example, **199712L**). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **wchar_t**, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month.

34. *Page 170, 7.3.1*
Replace paragraph 3 with:

> The macro
>
> **I**
>
> expands to **_Complex_I**.

35. *Page 170, 7.3.1*
Replace paragraph 4 with:

> Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros **complex** and **I**.

36. *Page 170, 7.3.1*
Remove paragraph 5.

37. *Page 188, 7.6*
Append to paragraph 6:

> If no such macros are defined, **FE_ALL_EXCEPT** shall be defined as **0**.

38. *Page 209, 7.11.2.1*
In paragraph 9, the first sentence, change "the rules" to "rules".

39. *Page 209, 7.11.2.1*
In paragraph 9, replace the names of the countries Finland, Italy, Netherlands, and Switzerland with Country1, Country2, Country3, and Country4 respectively.

40. *Page 209, 7.11.2.1*
In paragraph 10, the first sentence, change "are" to "could be".

41. *Page 209, 7.11.2.1*
In paragraph 10, replace the names of the countries Finland, Italy, Netherlands, and Switzerland with Country1, Country2, Country3, and Country4 respectively.

42. *Page 209, 7.11.2.1*
In paragraph 10, change the **`n_sep_by_space`** table entry for Country3 from **`1`** to **`2`.**

43. *Page 209, 7.11.2.1*
In paragraph 10, change all the **`int_p_sep_by_space`** table entries from **`0`** to **`1`.**

44. *Page 209, 7.11.2.1*
In paragraph 10, change the **`int_n_sep_by_space`** table entries for Country1 and Country3 from **`0`** to **`2`**.

45. *Page 209, 7.11.2.1*
In paragraph 10, change the **`int_n_sep_by_space`** table entries for Country2 and Country4 from **`0`** to **`1`**.

46. *Page 212, 7.12*
Change the first sentence of paragraph 6 to:

> The *number classification macros* are:
>
> ```
> FP_INFINITE
> FP_NAN
> FP_NORMAL
> FP_SUBNORMAL
> FP_ZERO
> ```

47. *Page 212, 7.12*
Append to paragraph 7:

> If defined, these macros expand to the integer constant 1.

48. *Page 223, 7.12.6.5*
In paragraph 2, change:

> A range error may occur if **`x`** is **`0`**.

to:

> A domain error or range error may occur if **`x`** is zero, infinite, or NaN.

49. *Page 223, 7.12.6.5*
Append to paragraph 2:

> If the correct value is outside the range of the return type, the numeric result is unspecified.

50. *Page 226, 7.12.6.11*
In paragraph 2, change:

> A domain error may occur if the argument is zero.

to:

> A domain error or range error may occur if the argument is zero.

51. *Page 228, 7.12.7.4*
In paragraph 2, change:

> A domain error may occur if `x` is zero and `y` is less than or equal to zero.

to:

> A domain error may occur if `x` is zero and `y` is zero. A domain error or range error may occur if `x` is zero and `y` is less than zero.

52. *Page 230, 7.12.8.4*
In paragraph 2, change:

> A domain error occurs if `x` is a negative integer or if the result cannot be represented when `x` is zero.

to:

> A domain error or range error may occur if `x` is a negative integer or zero.

53. *Page 232, 7.12.9.5*
In paragraph 2, change:

> If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of `x` is too large.

to:

> If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

54. *Page 233, 7.12.9.7*
In paragraph 2, change:

> If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of **x** is too large.

to:

> If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

55. *Page 234, 7.12.10.2*
Add to paragraph 3:

> If **y** is zero, whether a domain error occurs or the functions return zero is implementation defined.

56. *Page 235, 7.12.10.3*
Add to paragraph 3:

> If **y** is zero, whether a domain error occurs or the functions return zero is implementation defined.

If **y** is zero, the value stored in the object pointed to by **quo** is unspecified.

57. *Page 238, 7.12.13.1*
Add to paragraph 2:

> A range error may occur.

58. *Page 253, 7.17*
Change last part of paragraph 2 to:

wchar_t

> which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero.

59. *Page 255, 7.18.1.1*
Change paragraph 3 to:

> These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, no padding bits, and (for the signed types) that have a two's complement representation, it shall define the corresponding typedef names.

60. *Page 257, 7.18.2.1*
Add missing first paragraph number.

61. *Page 257, 7.18.2.2*
Add missing first paragraph number.

62. *Page 257, 7.18.2.3*
Add missing first paragraph number.

63. *Page 257, 7.18.2.4*
Add missing first paragraph number.

64. *Page 258, 7.18.2.5*
Add missing first paragraph number.

65. *Page 258, 7.18.3*
Append to paragraph 2:

> An implementation shall define only the macros corresponding to those typedef names it actually provides.

66. *Page 258, 7.18.3*
Add a new footnote to the last sentence:

> A freestanding implementation need not provide all of these types.

67. *Page 263, 7.19.1*
In paragraph 5, item 4, insert **`perror`** after **`gets`**.

68. *Page 277, 7.19.6.1*
Change **`g,G`** specifier in paragraph 8 to:

> **`g,G`** A **`double`** argument representing a floating-point number is converted in style **`f`** or **`e`** (or in style **`F`** or **`E`** in the case of a **`G`** conversion specifier), depending on the value converted and the precision. Let P equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **`E`** would have an exponent of X:
>
> — if $P > X \geq -4$, the conversion is with style **`f`** (or **`F`**) and precision $P - (X + 1)$.
>
> — otherwise, the conversion is with style **`e`** (or **`E`**) and precision $P - 1$.
>
> Finally, unless the **`#`** flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

69. *Page 280, 7.19.6.1*
Change the last line of code in paragraph 18 to:

```
fprintf(stdout, "|%13lc|\n", (wint_t) wstr[5]);
```

70. *Page 307, 7.20.1.3*
In paragraph 3, item 3, remove the words "one of".

71. *Page 307, 7.20.1.3*
In paragraph 3, item 4, remove the words "one of".

72. *Page 321, 7.20.7.2*
Add a paragraph number to paragraph 4 and, increment the following paragraphs.

73. *Page 324, 7.21.1*
Add a new paragraph 3:

> For all functions in this subclause, each character shall be interpreted as if it had the type **`unsigned char`** (and therefore every possible object representation is valid and has a different value).

74. *Page 337 7.23.1*
In paragraph 2, remove the word "constant".

75. *Page 371, 7.24.4.1.1*
In paragraph 3, item 3, remove the words "one of".

76. *Page 371, 7.24.4.1.1*
In paragraph 3, item 4, remove the words "one of".

77. *Page 386, 7.24.6.1.1*
In paragraph 3, change:

> The **`btowc`** returns

to:

> The **`btowc`** function returns

78. *Page 386, 7.24.6.1.2*
In paragraph 3, change:

> The **`wctob`** returns

to:

> The **`wctob`** function returns

79. *Page 457, F.9.3.5*
Change paragraph 1 to:

> If the correct result is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised.

80. *Page 464, F.9.8.4*
Change paragraph 1 to:

> No additional requirements beyond those on **`nextafter`**.

81. *Page 464, F.9.8.4*
Add paragraph number to paragraph 1.

82. *Page 465, G.2*
Add a new paragraph to the start of G.2:

There is a new keyword **`_Imaginary`**, which is used to specify imaginary types. It is used as a type specifier within declaration specifiers in the same way as **`_Complex`** is (thus "**`_Imaginary float`**" is a valid type name).

83. *Page 465, G.3*
Add missing first paragraph number.

84. *Page 466, G.4.2*
Replace paragraph 1 with:

> When a value of imaginary type is converted to a real type other than **`_Bool`**, the result is a positive zero. See 6.3.1.2.

85. *Page 470, G.6*
Replace paragraph 1 with:

The macros

`imaginary`

and

`_Imaginary_I`

are defined, respectively, as **_Imaginary** and a constant expression of type **const float _Imaginary** with the value of the imaginary unit. The macro

```
I
```

is defined to be **_Imaginary_I** (not **_Complex_I** as stated in 7.3). Notwithstanding the provisions of 7.1.3, a program may undefine and then perhaps redefine the macro **imaginary**.

86. *Page 473, G.6.2.2*
Add missing first paragraph number.

87. *Page 507, J.3.12*
Add after item 10:

— Whether a domain error occurs or zero is returned when a **remainder** function has a second argument of zero (7.12.10.2).
— Whether a domain error occurs or zero is returned when a **remquo** function has a second argument of zero (7.12.10.3).