INTERNATIONAL STANDARD ISO/IEC 9899:1990
TECHNICAL CORRIGENDUM 2

Published 1996-04-01

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 International Standard ISO/IEC 9899:1990 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology.*

Page 6

In subclause 5.1.2.1, page 6, delete:

There are otherwise no reserved external identifiers.

Page 7

In subclause 5.1.2.2.3, page 7, add at the end of the first sentence the footnote:

In accordance with subclause 6.1.2.4, objects with automatic storage duration declared in `main` will no longer have storage guaranteed to be reserved in the former case even where they would in the latter.

Page 11

In subclause 5.2.1.2, page 11, change the third bullet item:

wherein each sequence of multibyte characters begins in an *initial shift state* and enters other implementation-defined *shift states*

to:

wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states*

Page 22

In subclause 6.1.2.4, page 22, first paragraph, change:

There are two storage durations: static and automatic.

to:

There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.10.3.

ICS 35.060 **Ref. No. ISO/IEC 9899:1990/Cor.2:1996(E)**

Descriptors: data processing, computer software, artificial languages, programming languages, C (programming language).

Page 25

In subclause 6.1.2.6, page 25, first paragraph, change:

Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerated types, the members shall have the same values.

to:

Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if at least one is an incomplete type or if they have the same number of members, the same member names, and compatible member types; for two complete structure types, the members shall be in the same order; for two complete structure or union types, the bit-fields shall have the same widths; for two enumerated types, the members shall have the same values.

Page 31

In subclause 6.1.4, page 31, change the last paragraph of* Semantics *(before the* Example*) from:

Identical string literals of either form need not be distinct. If the program attempts to modify a string literal of either form, the behavior is undefined.

to:

These arrays need not be distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

Page 36

In subclause 6.2.2.1, page 36, change the parenthetic remark in the final sentence of the first paragraph:

(including, recursively, any member of all contained structures or unions)

to:

(including, recursively, any member or element of all contained aggregates or unions)

Page 41

In subclause 6.3.2.2, page 41, second paragraph, change:

> If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters.

to:

> If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type.

Page 45

In subclause 6.3.4, page 45, change the paragraph under* Constraints*:

Unless the type name specifies void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

to:

Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

Page 61

In subclause 6.5.2.2, page 61, second paragraph of* Semantics*, change:

Each enumerated type shall be compatible with an integer type; the choice of type is implementation-defined.

to:

Each enumerated type shall be compatible with an integer type. The choice of type is implementation-defined, but shall be capable of representing the values of all the members of the enumeration.

In subclause 6.5.2.2, page 61, append to* Semantics*:

The enumerated type is complete at the } that terminates the list of enumerator declarations.

Page 72

In subclause 6.5.7, page 72, the penultimate paragraph of* Semantics *(before* Examples*), add after the comma:

or fewer characters in a string literal or wide string literal used to initialize an array of known size, and elements of character or `wchar_t` type

Page 89

In subclause 6.8.3, page 89, change, in both paragraphs 2 and 3:

may be redefined by another `#define` preprocessing directive provided that

to:

shall not be redefined by another `#define` preprocessing directive unless

Page 99

In subclause 7.1.7, page 99, insert after the words in parentheses in the second sentence of the first paragraph:

or a type (after promotion) not expected by a function with variable number of arguments

Page 102

In subclause 7.3, page 102, second paragraph, change:

Those functions that have implementation-defined aspects only when not in the `"C"` locale are noted below.

The term *printing character* refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of an implementation-defined set of characters that are not printing characters.

to:

Those functions that have locale-specific aspects only when not in the `"C"` locale are noted below.

The term *printing character* refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.

In subclause 7.3.1.2, page 102, subclause 7.3.1.6, page 103, subclause 7.3.1.9, page 104, and subclause 7.3.1.10, page 104, change:

is one of an implementation-defined set of characters

to:

is one of a locale-specific set of characters

Page 107

In subclause 7.4.1.1, page 107, second paragraph of* Description*, change:

a value of `""` for `locale` specifies the implementation-defined native environment.

to:

a value of `""` for `locale` specifies the locale-specific native environment.

Page 122

In subclause 7.8.1, page 122, change the last sentence from:

The `va_start` and `va_end` macros shall be invoked in the function accepting a varying number of arguments, if access to the varying arguments is desired.

to:

The `va_start` and `va_end` macros shall be invoked in corresponding pairs in the function accepting a varying number of arguments, if access to the varying arguments is desired.

In subclause 7.8.1.1, page 122, add at the end of the second paragraph of the* Description*:

`va_start` shall not be invoked again for the same `ap` without an intervening invocation of `va_end` for the same `ap`.

Page 151

In subclause 7.10.1.4, page 151, subclause 7.10.1.5, page 152, and 7.10.1.6, page 152, change:

In other than the `"C"` locale, additional implementation-defined subject sequence forms may be accepted.