



INTERNATIONAL STANDARD ISO/IEC 9899:1990
TECHNICAL CORRIGENDUM 1

Published 1994-09-15
Corrected and reprinted 1995-09-15

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

Programming languages — C

TECHNICAL CORRIGENDUM 1

Langages de programmation — C

RECTIFICATIF TECHNIQUE 1

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

Page 6

In subclause 5.1.1.3, lines 15-17, change:

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint.

to:

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined.

Add to subclause 5.1.1.3:

Example

An implementation shall issue a diagnostic for the translation unit:

```
char i;  
int i;
```

because in those cases where wording in this International Standard describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error shall be diagnosed.

Page 13

In subclause 5.2.4.1, lines 1-2, change:

— 15 nested levels of compound statements, iteration control structures, and selection control structures

to:

— 15 nested levels of compound statements, iteration statements, and selection statements

ICS 35.060

Ref. No. ISO/IEC 9899:1990/Cor.1:1994(E)

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

© ISO/IEC 1994

Printed in Switzerland

Page 18

Add to subclause 6.1, (Semantics):

A header name preprocessing token is only recognized within a `#include` preprocessing directive, and within such a directive, a sequence of characters that could be either a header name or a string literal is recognized as the former.

Page 20

Add to subclause 6.1.2, (Semantics):

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword.

Page 21

In subclause 6.1.2.2, change:

If the declaration of an identifier for an object or a function contains the storage-class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

to:

For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible*, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the latter declaration becomes the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage. [Footnote *: As specified in 6.1.2.1, the latter declaration might hide the prior declaration.]

Page 25

In subclause 6.1.2.6, lines 19-20, change:

For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

to:

For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible*, if the prior declaration specifies internal or external linkage, the type of the identifier at the latter declaration becomes the composite type. [Footnote *: As specified in 6.1.2.1, the latter declaration might hide the prior declaration.]

Page 32

In subclause 6.1.7, lines 32-34, delete:

Constraint

Header name preprocessing tokens shall only appear within a `#include` preprocessing directive.

Add to subclause 6.1.7, (Semantics):

A header name preprocessing token is recognized only within a `#include` preprocessing directive.

Page 38

In subclause 6.3, lines 18-21, change:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:³⁶

- the declared type of the object,
- a qualified version of the declared type of the object,

to:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:³⁶

- a type compatible with the declared type of the object,
- a qualified version of a type compatible with the declared type of the object,

Page 40

In subclause 6.3.2.2, line 35, change:

The value of the function call expression is specified in 6.6.6.4.

to:

If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.6.6.4. Otherwise, the function call has type `void`.

Page 54

Add to subclause 6.3.16.1, another Example:

In the fragment:

```
char c;
int i;
long l;

l = ( c = i );
```

the value of `i` is converted to the type of the assignment-expression `c = i`, that is, `char` type. The value of the expression enclosed in parenthesis is then converted to the type of the outer assignment-expression, that is, `long` type.

Page 58

Add to subclause 6.5.1, (Semantics):

If an aggregate or union object is declared with a storage-class specifier other than `typedef`, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

Page 62

In subclause 6.5.2.3, line 27, change:

occurs prior to the declaration that defines the content

to:

occurs prior to the `}` following the *struct-declaration-list* that defines the content

Page 63

Add to subclause 6.5.2.3, another Example:

An enumeration type is compatible with some integral type. An implementation may delay the choice of which integral type until all enumeration constants have been seen. Thus in:

```
enum f { c = sizeof(enum f) };
```

the behavior is undefined since the size of the respective enumeration type is not necessarily known when `sizeof` is encountered.

Page 68

In subclause 6.5.4.3, lines 2-4, replace:

In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

with:

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

In subclause 6.5.4.3, lines 22-25, change:

(For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 6.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

to:

(In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the type that results from conversion to a pointer type, as in 6.7.1, and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)