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 JTC1, *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

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.)

Page 71

In subclause 6.5.7, line 39, change:

All unnamed structure or union members are ignored during initialization.

to:

Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization. A union object containing only unnamed members has indeterminate value even after initialization.

Pages 71 and 72

In subclause 6.5.7, page 71, line 41 through page 72, line 2, change:

If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

to:

If an object that has static storage duration is not initialized explicitly, then:

— if it has pointer type, it is initialized to a null pointer;

— if it has arithmetic type, it is initialized to zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules;

— if it is a union, the first named member is initialized (recursively) according to these rules.

Page 72

In subclause 6.5.7, line 11, change:

The initial value of the object is that of the expression.

to:

The initial value of the object, including unnamed members, is that of the expression.

Page 80

In subclause 6.6.6.4, lines 30-32, replace:

If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type.

with:

If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment toan object having the return type of the function.*

[Footnote *: The **return** statement is not an assignment. The overlap restriction in subclause 6.3.16.1 does not apply to the case of function return.]

Add to subclause 6.6.6.4:

Example

In:

```
struct s {double i;} f(void);
union {struct {int f1;
                struct s f2;} u1;
       struct {struct s f3;
                int f4;} u2;
      } g;
struct s f(void)
     {
     return g.u1.f2;
     }
/* ... */
g.u2.f3 = f();
```

the behavior is defined.

Page 84

Add to subclause 6.7.2, a second Example:

If at the end of the translation unit containing

```
int i[];
```

the array **i** still has incomplete type, the array is assumed to have one element. This element is initialized to zero on program startup.

Page 86

***Add to subclause 6.8, line 5, (*Description**)**:

A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

Add to subclause 6.8, (*Constraints*):

In the definition of an object-like macro, if the first character of a replacement list is not a character required by subclause 5.2.1, then there shall be white-space separation between the identifier and the replacement list.*

[Footnote *: This allows an implementation to choose to interpret the directive:

```
#define THIS$AND$THAT(a, b)    ((a) + (b))
```

as defining a function-like macro **THISANDTHAT**, rather than an object-like macro **THIS**. Whichever choice it makes, it must also issue a diagnostic.]

Page 90

Add to subclause 6.8.3.3:

Example

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y); /* equivalent to char p[] = "x ## y"; */
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding **hash_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the catenation operator.

Page 96

Add to subclause 7.1.2, (before* Forward references*):

Any definition of an object-like macro described in this clause shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

In subclause 7.1.2, lines 32-33, change:

However, if the identifier is declared or defined in more than one header,

to:

However, if an identifier is declared or defined in more than one header,

Page 120

In subclause 7.7, lines 14-16, change:

and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

to:

and the following, which expand to positive integral constant expressions with distinct values that are the signal numbers, each corresponding to the specified condition:

Page 132

In subclause 7.9.6.1, lines 37-38, change:

For **o** conversion, it increases the precision to force the first digit of the result to be a zero.

to:

For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero.

Page 135

In subclause 7.9.6.2, lines 31-33, change:

An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence.

to:

An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.

Page 137

In subclause 7.9.6.2, delete:

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.

Add to subclause 7.9.6.2, line 4 (the* n *conversion specifier):

No argument is converted, but one is consumed. If the conversion specification with this conversion specifier is not one of **%n**, **%ln**, or **%hn**, the behavior is undefined.

Add to subclause 7.9.6.2:

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.* [Footnote *: **fscanf** pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to **strtod**, **strtol**, or **strtoul** are unacceptable to **fscanf**.]

Page 138

Add to subclause 7.9.6.2, another Example:

In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to **d1** and the value 3 to **n1**. Because **%n** can never get an input failure the value of 3 is also assigned to **n2**. The value of **d2** is not affected. The value 3 is assigned to **i**.

Page 145

In subclause 7.9.9.2, lines 39-40, change:

a value returned by an earlier call to the **ftell** function

to:

a value returned by an earlier successful call to the **ftell** function

Page 146

In subclause 7.9.9.3, lines 10-11, change:

a value obtained from an earlier call to the **fgetpos** function

to:

a value obtained from an earlier successful call to the **fgetpos** function

Page 162

Add to subclause 7.11.1:

Where an argument declared as **`size_t n`** specifies the length of the array for a function, **`n`** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call must still have valid values, as described in subclause 7.1.7. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

Page 172

In subclause 7.12.2.3, line 16, change:

```
if (mktime(&time_str) == -1)
```

to:

```
if (mktime(&time_str) == (time_t)-1)
```

Page 200

Add to subclause G.2:

— A program contains no function called **`main`** (5.1.2.2.1).

Page 201

Add to subclause G.2:

— A storage-class specifier or type qualifier modifies the keyword **`void`** as a function parameter type list (6.5.4.3).

Add to subclause G.2:

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression **`a[1][7]`** given the declaration **`int a[4][5]`**) (6.3.6).

Page 202

Add to subclause G.2:

— A fully expanded macro replacement list contains a function-like macro name as its last preprocessing token (6.8.3).

Page 203

Add to subclause G.2:

— A call to a library function exceeds an **environmental limit** (7.9.2, 7.9.3, 7.9.4.4, 7.9.6.1, 7.10.2.1).

Page 217

In the index, change:

`static` storage-class specifier, 3.1.2.2, 6.1.2.4, **6.5.1**, 6.7

to:

`static` storage-class specifier, 6.1.2.2, 6.1.2.4, **6.5.1**, 6.7