WG14/N329

X3J11/94-007

ISO / IEC JTC1 / SC22
Programming languages, their environments and system software interfaces
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC22
N 1532

DECEMBER 1993

TITLE: Summary of Voting on Draft Technical Corrigendum 1 to ISO/IEC 9899:1990 Programming language C

SOURCE: Secretariat ISO/IEC JTC1/SC22

WORK ITEM: JTC1.22.20.01

STATUS: New

CROSS REFERENCE: N1445

DOCUMENT TYPE: Summary of Voting

ACTION: For information to SC22 Member Bodies.
See attached.

Address reply to: ISO/IEC JTC1/SC22 Secretariat
J.L. Côté
Treasury Board Secretariat
300 Laurier Ave. West, 10th Floor, Ottawa, Ontario, Canada K1A 0R5
Tel.: (613)957-2496 Telex: 053-3336 Fax: (613)957-8700

455

SUMMARY OF VOTING ON:

Letter Ballot Reference No: SC22 N1445
Circulated by :JTC1/SC22
Circulation Date :1993-09-01
Closing Date :1993-12-10

SUBJECT: Draft Technical Corrigendum 1 to ISO/IEC 9899:1990
Programming language C

The following responses have been received:

'P' Members supporting proposal,
without comments : 13

'P' Members supporting proposal,
with comments : 01

'P' Members not supporting proposal, : 00

'P' Members abstaining : 01

'P' Members not voting : 07

Secretariat Action:

The comments received will be forwarded to WG14 - C for recommendation on further processing of Technical Corrigendum 1.

ISO/IEC JTC1/SC22 LETTER BALLOT SUMMARY

PROJECT NO: JTC1.22.20.01

SUBJECT: Draft Technical Corr 1 to ISO/IEC 9899:1990 Programming Language C

Reference Document No: N1445
Circulation Date: 1993-09-01

Ballot Document No: N1445
Closing Date:1993-12-10

Circulated To: SC22 P, L

Circulated By: Secretariat

SUMMARY OF VOTING AND COMMENTS RECEIVED

	Approve	Disapprove	Abstain	Comments	Not Voting
'P' Members					
Australia	( )	( )	(x)	( )	( )
Austria	(x)	( )	( )	( )	( )
Belgium	(x)	( )	( )	( )	( )
Brazil	( )	( )	( )	( )	(✓)
Bulgaria	( )	( )	( )	( )	(✓)
Canada	(x)	( )	( )	(x)	( )
China	( )	( )	( )	( )	(✓)
Denmark	(x)	( )	( )	( )	( )
Finland	(x)	( )	( )	( )	( )
France	( )	( )	( )	( )	(✓)
Germany	(x)	( )	( )	( )	( )
Greece	( )	( )	( )	( )	(✓)
Italy	(x)	( )	( )	( )	( )
Japan	(x)	( )	( )	( )	( )
Netherlands	(x)	( )	( )	( )	( )
New Zealand	(x)	( )	( )	( )	( )
Slovenia	( )	( )	( )	( )	(✓)
Sweden	(x)	( )	( )	( )	( )
Switzerland	( )	( )	( )	( )	(✓)
UK	(x)	( )	( )	( )	( )
Ukraine	(x)	( )	( )	( )	( )
USA	(x)	( )	( )	( )	( )
'O' Members					
Argentina	( )	( )	( )	( )	( )
Cuba	( )	( )	( )	( )	( )
Czech/Slovak Re	(x)	( )	( )	( )	( )
Hungary	( )	( )	( )	( )	( )
Iceland	( )	( )	( )	( )	( )
India	( )	( )	( )	( )	( )
Poland	(x)	( )	( )	( )	( )
Portugal	( )	( )	( )	( )	( )
Romania	( )	( )	( )	( )	( )
Singapore	( )	( )	( )	( )	( )
Turkey	( )	( )	( )	( )	( )
Thailand	( )	( )	( )	( )	( )
Yugoslavia	( )	( )	( )	( )	( )

Canadian comments on JTC1/SC22N1445

Proposed technical corrigendum 1 to ISO/IEC 9899:1990

Canadian response is to approve for circulation as DIS with the following comments:

In what follows, the notation "[DR17Q37, p16]" means the response to Defect Report 17, question 37, beginning on page 16 of N1445.

TECHNICAL PROBLEMS

1. Regarding [DR01Q1, p2]:

The proposed addition to subclause 6.6.6.4, "The overlap restriction in subclause 6.3.16.1 does not apply to the case of function return",
does not make sense as written. Subclause 6.3.16.1 is about simple assignment and has nothing to do with function return; mentioning it in 6.6.6.4 does not change this fact.

It seems to us that the ambiguity that's being cleared up here arose, not because of any real problem in 6.6.6.4, but because the overlap restriction 6.3.16.1 is unclear as to the time period in which it is restricting accesses. Taken literally, 6.3.16.1 prohibits a program that assigns to a struct member and then, in a separate statement, copies that struct to another struct.

We think that the following might have been the intention:

> If a value is stored in an object by simple assignment, and a value is accessed from an object that overlaps that object, and no sequence point occurs between the storage and the access, then the behavior is undefined, unless the overlap is exact and the two objects have qualified or unqualified versions of a compatible type.

In the example with the union object named g, the sequence point at the end of the full expression g.u1.f2 (in the return statement) must occur before the function call is complete and therefore before the value read in that expression is stored in the possibly overlapping object g.u2.f3. Therefore in our proposed wording the behavior is defined.

Of course, our proposal also makes legal some cases other than those mentioned in the Defect Report. But as our remarks above suggest, we think these should be unobjectionable.

As an editorial matter, we note that subclause 6.6.6.4 does refer in a somewhat awkward manner to assignment. It could be clarified by adding a constraint:

> The expression shall have type suitable for the right operand of a simple assignment whose left operand has the same type as the return type of the function.

and altering the semantics:

| If the expression has a type different from THE RETURN TYPE of the function in which it appears,
| THE VALUE is converted TO THE RETURN TYPE OF THE FUNCTION.

And for still greater clarity there could be a footnote:

| The return statement is not an assignment and does not modify an lvalue (except where operators
| in the expression itself modify lvalues). Only the expression's type is constrained as though for
| assignment. The return value of a function is not an lvalue.

2. Regarding [DR17Q3, p11]:

The constraint can hardly "take precedence" when it has already been violated. If the intent is that at least one diagnostic shall be issued for a program containing such a situation, the amendment should simply say so. However, it might be clearer to instead append to the first sentence of 5.1.1.3:

> , even if the behavior is also explicitly specified as undefined or implementation-defined.

3. Regarding [DR17Q6, p12]:

The proposed change to 6.5.1 is so worded as to require the declaration

```
extern struct s { int i; } x;
```

to be interpreted as

```
extern struct s { extern int i; } x;
```

We believe this is a syntax error, but whether it is or not, if it is given its obvious meaning then subclause 6.1.2.2 requires it to declare the identifier i with external linkage.

The intent is to assign to contained objects, not the storage-class specifier, and not even all of the properties of the storage class, but only the properties relating to storage duration and "registerness". The wording should say so.

It also needs to be made explicit that this behavior continues recursively to aggregate or union objects within the aggregate or union objects.

Here's one possibility:

> 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 members that are themselves aggregate or union objects.

(The recursion has to be explicit because only the outermost level can have a storage-class *specifier*.)

4. Regarding [DR17Q37, p16]:

There are two technical problems here:

(a) The entity whose type is being described is the function call expression, not "the result of the function call".

(b) It is possible for a function to return without a return statement being executed, even if it does not have type void. Consider the call to hi() in the program:

```
#include <stdio.h>
main() { hi(); return 0; }
hi() { printf("Hello, world\n"); }"
```

The proposed wording change seems to pretend that this can't happen.

There are also two editorial problems:

(c) Statements do not "execute", they "are executed".

(d) The word "otherwise" awkwardly refers back two sentences.

These defects might all be corrected by changing the wording inserted in subclause 6.3.2.2 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 the value determined by the return statement (if any) that is executed within the called function, as specified in 6.6.6.4. Otherwise the function call has type void.

But a simpler alternative would be to take advantage of the reference to 6.6.6.4 and just write:

> 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 the value determined as specified in 6.6.6.4. Otherwise the function call has type void.

5. Regarding [DR43Q1, p22]:

The added wording in subclause 7.1.2 should refer to an "object-like macro" rather than any macro. A function-like macro should group like a function call, not like an identifier, and 7.1.7 already requires this.

EDITORIAL COMMENTS

6. Regarding [DR13Q1, p 7]:

"Compatibility comparison" is not a term defined in the standard. Some alternatives:

(a) ... its type for these purposes ...

(b) ... in the determination of type compatibility and of a composite type, its type is taken as ...

(c) Recast the whole parenthetic sentence to:

(In this determination of type compatibility and of a composite type, each function parameter declared with function or array type is taken as having the type resulting from conversion ...)

7. Regarding [DR16Q2, p10]:

The wording is unnecessarily repetitive. There is no reason to split out pointer types from arithmetic types, but I suppose it's harmless. At least, though, the words "it is initialized implicitly according to these rules" could be reduced to a simple "then".

8. Regarding [DR27Q1, p20]

There's no need for this "minimal basic source character set" wording, which requires a footnote to explain it. We suggest instead:

| If the first character of a replacement-list is not a character
| required by subclause 5.2.1 to be in the basic source character
| set, then there shall be white-space separation between the
| identifier and the replacement-list.

Here's one possibility:

> 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 members that are themselves aggregate or union objects.

(The recursion has to be explicit because only the outermost level can have a storage-class *specifier*.)

4. Regarding [DR17Q37, p16]:

There are two technical problems here:

(a) The entity whose type is being described is the function call expression, not "the result of the function call".

(b) It is possible for a function to return without a return statement being executed, even if it does not have type void. Consider the call to hi() in the program:

```
#include <stdio.h>
main() { hi(); return 0; }
hi() { printf("Hello, world\n"); }"
```

The proposed wording change seems to pretend that this can't happen.

There are also two editorial problems:

(c) Statements do not "execute", they "are executed".

(d) The word "otherwise" awkwardly refers back two sentences.

These defects might all be corrected by changing the wording inserted in subclause 6.3.2.2 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 the value determined by the return statement (if any) that is executed within the called function, as specified in 6.6.6.4. Otherwise the function call has type void.

But a simpler alternative would be to take advantage of the reference to 6.6.6.4 and just write:

> 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 the value determined as specified in 6.6.6.4. Otherwise the function call has type void.