Comments on TC1

WG14/N504
X3J11/93-053
1 DEC 93
PARTNER

[Comments marked [Brader] were written by Mark Brader and included with permission.]

[Brader] DR01Q1

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", is nonsense 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 me 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.

I 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 my proposed wording the behavior is defined.

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

As an editorial matter, I 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 sementics:

| 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.

========

[Brader] DR13Q1

"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 ...)

========

DR13Q4

I've been having second and third thoughts about DR13Q4. I went back and looked at 6.5.2.3, and it seems to me that the original questioner is confused, or perhaps I am. Let's take this in two parts.

First, look at his example:

```
struct foo              /* #1 */
{
   struct foo *p;       /* #2 */
}
a [sizeof (struct foo)]; /* #3 */
```

Now, #1 indicates the start of the definition of struct foo, and this is IMHO adequately described by the first part of the semantics of 6.5.2.3; I'll come back to this later.

#2 is part of a pointer to an indirect type. The questioner was unsure that 6.5.2.3 made it clear that #2 (a) refers to #1, and (b) is indirect at this point. Now point (b) is made by 6.5.2.1:

The type is incomplete until after the } that terminates the list.

Point (a) is made in the first item of 6.5.2.3:

> If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared structure, union, or enumerated type.

if the declaration is visible at #2. Again, I'll come back to this later.

#3 is reference to a completed type. The fact that it is completed is made by the same quote from 6.5.2.1; again, whether it refers to the same type as #1 is the same question as it is for #2.

This brings us to the second part: what does the quoted wording mean, and is it correct ? If we read the 6.5.2.3 as a whole, it clearly falls into four separate parts:

(1) Use of:
struct-or-union identifier { struct-declaration-list }
enum identifier { enumerator-list }

This case defines the contents of the type. Uses of the tag with this visible refer to this type.

(2) Use of:
struct-or-union identifier

prior to case (1). This is the one quoted in the question. This declares the tag as an incomplete type.

(3) Use of:
struct-or-union identifier ;

This declares the tag (as an incomplete type, because this case also meets case (2)), and makes it distinct from any type in an enclosing scope.

(4) Use of:
struct-or-union { struct-declaration-list }
enum { enumerator-list }

Declares a type different from any other.

When this is all spelt out like this, it seems that we need a case (5):

(5) Use of:
struct-or-union identifier

not prior to the occurence of case (1).

But do we ? Now, when case (1) is visible, case (5) refers to the type defined in case (1), and case (1) adequately says so. So, the question is whether there is any point which falls into case (5), yet for which case (1) is not visible. To answer this, we look at the definition of "visible" in 6.1.2.1:

> An identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*.
> [....]
> Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag.

So there aren't any such points, and my case (5) is adequately handled by case (1). Furthermore, #1 *is* visible to #2 and #3.

Therefore I conclude that the Standard correctly answers the question, and no Technical Corrigendum is required for it. The category should be changed to Record of Response, and the response changed to reflect the discussion above.

========

DR14Q2:

I thought that we decided to change the definition of %n to make it clear that it can't have an input failure.

After further thought, I believe that we need to add the following statements:
- though no argument is converted, one is consumed;
- if the total specification is not one of "%n", "%ln", or "%hn", the effect is undefined.

========

DR16Q2:

For consistency with the rest of the Standard, the bullet points should use dashes, not numbers. Furthermore, points 1 and 2 don't need the word "implicitly" repeated in them.

[Brader]

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".

========

DR17Q1:
In reviewing this, I found myself unable to locate *any* syntax rule or constraint violated by:

```
#if macro(1
```

========

[Brader] DR17Q3

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.

========

DR17Q6:

In reviewing this, I found myself unable to locate *any* syntax rule or constraint violated by:

```
register struct s { int c; };
```

Should we add the word "object" after "union" to prevent people from thinking we're talking about this case. Do we need to add words to outlaw this case ?

[Brader]

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;
```

I 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*.)

========

[Brader] DR17Q37

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.

========

DR27Q1:

Replace "90-odd" with the exact number. I think it is 95, but PJP disagrees.

[Brader]

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

A footnote might also be added to explain the reason:

[*] This allows an implementation to choose to interpret the line

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

as defining a function-like macro THISANDTHAT, rather than an object-like macro THIS. Whether it does so or not, a diagnostic is required.

========

[Brader] DR43Q1

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.

========

[Brader] DR54Q1

In the proposed added wording, the reference to subclause 7.1.7 must

be to one or both of the following sentences there:

> If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined. If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

But the second sentence is irrelevant, since there are no zero-length arrays and therefore no address computations or accesses could be valid for them. It appears, then, that the intent is merely to say that the pointer must point to an object except where the function is described as accepting a null pointer. Why not say so?

Also, it seems more exact to say that n specifies, rather than determines, the length of the array.

Finally, memchr() is not properly covered; the words "and a function that searches for a character shall not find it" could be added.