ANSI/ISO C Defect Report #rfg24: WG14/N303
---------- X3J11/93-050

Subject: Abstract semantics, sequence points, and expression evaluation.

Question: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
int example ()
{
 int x1 = 2, x2 = 1, x_temp;

 return (x_temp = x1, x_temp) + (x_temp = x2, x_temp);
}
```

Background:

Section 5.1.2.3; ISO C standard:

> "The semantic descriptions in this Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant."

Section 6.3; ISO C standard:

> "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression."

Although it is quite clear that the above quoted "modified at most once" rule was intended to render certain programs "not strictly conforming", there is an unfortunate amount of ambiguity built-in to the current wording of that rule.

Quite simply, while the "modified at most once" rule is obviously telling us what a "strictly conforming program" must not do between two particular points IN TIME, it is altogether less than clear what events and/or actions (exactly) are associated with these two points in time. Additionally, it is also less than clear (from reading the remainder of the standard) what actions and/or events are allowed (or required) to take place between some pair of sequence points in cases where both members of the pair are part of some large single expression whose evaluation order is not completely dictated by the standard.

Note that despite the assertion given in ISO 5.1.2.3 (and quoted above) the standard does not *fully* specify the behavior of the "abstract machine", especially when it comes to the issue of the ordering of sub-expression evaluation used by the "abstract machine" model.

This fact makes it inheriently impossible to precisely determine even just the *relative* timings of various events (including the "occurance" of or

the "execution" of or the "evaluation" of sequence points) which may (or must) occur sometime during the evaluation of a larger containing expression (except in a few cases involving `||' or `&&' or `?:' or `,' operators).

To put it more plainly, if some pair of sequence points will be "reached" (or "evaluated" or "executed") during the evaluation of any pair of sub-expressions which are themselves operands for some binary operator (other than the operators `||' or `&&' or `?:' or `,') then the standard's description of the "abstract machine" semantics are inadequate to enable us to know either which *order* these two sequence points will occur in, or even which other aspects of the evaluation of the overall expression may (or must) occur "between" the two sequence points.

Thus, it seems that it may also be inheriently impossible to know whether or not the prohibition against multiple modifications of a given variable "between" two consecutive sequence points is (or may be) violated in such contexts.

Here is a simple example of an expression which illustrates these points:

```
(x = i, x) + (x = j, x)
```

In this expression there are two "comma" sequence points, however nothing in the standard gives any indication as two which of these two may be (or must be) "evaluated" or "reached" first. (Indeed, it would seem that on a parallel machine of some sort, BOTH points could perhaps be reached simultaneously.) It is fairly clear however that each of the references to the stored values of `x' must not be evaluated until their respective preceeding "comma sequence points" have been "reached" or "evaluated". Thus, a partial (but very incomplete) ordering is imposed upon the sequence of events which must occur during the evaluation of this expression.

For the sake of this example, let us call the leftmost comma in the above expression "lcomma" and call the rightmost comma "rcomma". Given this terminology, it would appear the the standard permits the following sequence of events during evaluation of the above expression:

```
eval(i)
x=        (leftmost assignment to x)
lcomma     <==== sequence point
eval(x)   (leftmost reference to stored value of x)
eval(j)
x=        (rightmost assignment to x)
rcomma     <==== sequence point
eval(x)   (rightmost reference to stored value of x)
+
```

Note that in this (very realistic) example, the stored value of `x' is NEVER modified more than once between any pair of sequence points. Given that the ordering described above is both a perfectly PLAUSIBLE and also a per-

fectly PERMISSIBLE ordering for the evaluation of the expression in question, and given that this particular permissible ordering of events does not violate the "modified at most once" rule (quoted earlier) it therefore appears that the expression in question may in fact be interpreted as being "strictly conformant", and that such expressions may appear within "strictly conformant" programs.

I would like the committee to either confirm or reject this view, and to provide some commentary explaining that confirmation or rejection.

ANSI/ISO C Defect Report #rfg25:

Subject: Completion point for enum types.

Question: Are diagnostics required for the following code examples?

```
enum E1 { enumerator1 = sizeof (enum E1) };
enum E2 { enumerator2 = sizeof (enum E2 *) };
```

(Just read on! This *isn't* just the same old question again!)

Background:

Section 6.3.3.4; ISO C standard (constraints):

> "The sizeof operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type..."

Section 6.5.2.1; ISO C standard (semantics):

> "The [struct or union] type is incomplete until after the } that terminates the list [of member declarations]."

(Bracketed portions added for clarity.)

CIB #1, RFI #13, response to question #5:

> "For the example:
>
> ```
> enum e { a = sizeof(enum e) };
> ```
>
> the relevant citations are {ANSI} 3.1.2.1 starting on page 21, line 39, indicating that the scope of the first `e' begins at the `{', and {ANSI} 3.5.2.2, page 62, line 20, which attributes meaning to a later `enum e' ONLY IF this use appears in a SUBSEQUENT declaration. By subsequent, we mean "after the `}'". Because in this case, the

second `enum e' is not in a subsequent declaration, and no other wording in the standard addresses the meaning, the standard has left this example in the category of undefined behavior."

Please note that the above response to RFI #13, question #5 has totally failed to solve the *real* problem with the current wording of the standard.

The *real* problem is that (unlike the case for struct and union type definitions) nothing in the standard presently indicates where (or whether) an enum type becomes "completed".

This is a very serious flaw in the current standard. Given that the standard currently contains no statement(s) which specify where (or whether) an enum type becomes a "completed" type, any and all programs which use ANY enum type in ANY context requiring a completed type are, by definition, NOT "strictly conforming". (This will come as quite a shock to a number of C programmers!)

I feel that the committee must resolve this serious problem as soon as possible. The only plausible way to do that is to add a statement to section 6.5.2.2 of the ISO C standard which will specify the point at which an enum type become a "completed" type.

Using the statement currently given in section 6.5.2.1 of the ISO C standard (relating to struct and union types) as a guide, it would appear that section 6.5.2.2 should be ammended to include the following new semantic rule:

> "The enum type is incomplete until after the } that terminates the list of enumerators."

Some such addition is obviously necessary in order to render enum types usable as complete types within "strictly conforming" programs.

Note however that such a clarification would have the additional (beneficial?) side effect of rendering the following declaration subject to a mandatory diagnostic (due to the violation of the constraints for the operand of the sizeof operator):

```
enum E1 { enumerator1 = sizeof (enum E1) };
```

Even after such a clarification however, the status of:

```
enum E2 { enumerator2 = sizeof (enum E2 *) };
```

is still questionable at best, and the proper interpretation for such a case should, I believe, still be drawn from the response given to RFI #13, question #5... i.e. such examples should be viewed as involving undefined behavior.

-- Ronald F. Guilmette, Sunnyvale, California ----------