WG14/N299
X3J11/93-045

ANSI/ISO C Defect report #rfg1:

There appears to be an inconsistancy between the constraints on "passing" values versus "returning" values.

The constraints for function calls clearly indicate that a diagnostic is required if any given actual argument is passed (to a prototyped function) into a corresponding formal parameter whose type is not assignment compatatable with respect to the type of the passed value.

In the case of values returned by a return statement however, there seems to be no such compatability constrain imposed upon the expression given in the return statement and the corresponding (declared) function return type.

A new constraint should be added to the standard like:

> If present, the expression given in a return statement shall have a type such that its value may be assigned to an object with the unqualified version of the return type of the containing function.

(This exactly mirrors the existing constraint on parameter matching imposed upon calls to prototyped functions.)

ANSI/ISO C Defect report #rfg2:

There is an ambiguity with respect to the constraints which (or may not) apply to initializations.

Section 3.5.7 of the ("classic") ANSI C standard says:

> "...the same type constraints and conversions as for simple assignment apply."

Note however that this rule itself appears within a semantics section, thus leading some implementors to feel that no diagnostics are required in cases where an attempt is made to provide an initializer for a give scalar and where the type of the initializer is NOT assignment compatible with the type of the scalar object being initialized.

This ambiguity should be removed by adding an explicit constraint to the section covering initializations, such as:

> Each scalar initializer expression given in an initializer shall have a type such that its value may be assigned to an object with the unqualified version of the corresponding scaral object to be initialized by the given scalar initializer expression.

(This roughly mirrors the existing constraint on parameter matching imposed upon calls to prototyped functions.)

ANSI/ISO C Defect report #rfg3:

Section 3.5.4.2 (Array Declarators) of the ("classic") ANSI C standard fails to contain any constraint which would prohibit the element type in an array declarator from being a type which is not an object type. (Note that section 3.1.2.5 of the "classic" ANSI C standard seems to suggest that such usage is prohibited by saying that "An array type describes a contiguously allocated nonempty set of objects...", but this still leaves the matter rather unclear.)

I believe that some new constraint prohibiting the element type in an array declarator from being a non-object type (at least in some obvious cases) is clearly needed.

Please consider the case of an array declarator, occuring at some point within a given translation unit, and indicating an element type 'T', where 'T' is one of the following:

1) A function type.

2) A void type.

3) An incomplete struct or union type which is NEVER completed within the given translation unit.

4) An incomplete struct or union type which IS completed later within the given translation unit.

5) An incomplete array type which is NEVER completed within the given translation unit.

6) An incomplete array type which IS completed later within the given translation unit.

I believe that it should be abundantly clear that the standard should contain a constraint prohibiting array declarators where the specified element type is either (1) or (2). Essentially all existing implementations already issue diagnostics for such usage.

Also, in cases where an array declarator uses either a (3) or a (5) as the element type, it seems emminently reasonable to require diagnostics... and indeed, many/most existing implementations already do issue diagnostics for such usage... but this is perhaps debatable.

Cases (4) and (6) from the above list are *entirely* debatable. Existing practice among so-called "conformant" C compilers varies with respect to these cases (in which an element type is completed at some point AFTER use of the type, as an element type, in an array declarator). Here are two examples:

```
struct S array[10];     /* ok? */
struct S { int member; };   /* type completed now */

int array_of_array[][];     /* ok? */
int array_of_array[5][5];   /* type completed now */
```

As I say, I believe that the very least the committee should do is to add

a constraint requiring diagnostics for array declarators whose element types fall into categories (1) or (2). The committee may wish to provide an even more stringent interpretation of 3.1.2.5 and also require diagnostics for element types falling into categories (3) and/or (5). The committee may even wish to take *the* simplest approach to this entire problem, and simply require diagnostics for ANY case in which an array declarator specifies an element type which is not (already) an object type.

Regardless of which choice is made, I feel strongly that it is important for section 3.5.4.2 (Array Declarators) to be revised to fully reflect both common sense and (to the extent possible) the intent of section 3.1.2.5.

Footnote: Note that while is it ALWAYS possible for a given incomplete struct or union type to be completed somewhere later within the same scope and same translation unit where it is used, and while it is OFTEN possible to complete a given incomplete array type later within the same scope and same translation unit where it is used (as illustrated by the above examples) it can sometimes be IMPOSSIBLE to EVER complete a given array type later within its scope and translation unit. This will certainly be the case whenever the array type in question is NOT used to declare an entity having *some* linkage (either internal or external).

Examples:

```
    void example ()
    {
       void *vp = (int (*)[][]) 0;   /* abstract declarator declares no
                        object - type can't be completed */

       int array[][];        /* no linkage - type can't ever be
                        completed */
    }
```

I mention these cases only because they may potentially have some small bearing upon the committee's deliberations of the certral issues of this Defect Report.

ANSI/ISO C Defect report #rfg4:

Section 4.1.5 of the ("classic") ANSI C standard fails to contain any constraint which would prohibit the type argument given in an invocation of the offsetof() macro from being an incomplete type.

This situation can arise in examples such as the following:

```
   #include <stddef.h>

   struct S
   {
       int member1;
       int member2[1+offsetof(struct S,member1)];
   };
```

I believe that a constraint prohibiting the type argument to offsttof()
from being an incomplete type is clearly needed.

This problem could be solved by adding an explicit constraint to ("classic")
section 4.1.5, such as:

> The type argument given in an invocation of the offsetof() macro
> shall be the name of a complete struct type or a complete union type.

(Note that this way of expressing the constraint also makes it completely
clear that diagnostics are required for cases where the type given in the
invocation is, for instance, a function type, an array type, an enum type,
a pointer type, or a built-in arithmentic type.)

ANSI/ISO C Defect report #rfg5:

Section 3.3.3.4 of the ("classic") ANSI C standard provides the following
constraint:

> "The sizeof operator shall not be applied to an expression that has
> function type or an incomplete type..."

The logical implication of this constraint is that neither function types
nor incomplete types have "sizes" per se... at least not as far as the
standard is concerned.

I have noted however that neither ("classic") section 3.3.2.4 (Posfix
increment and decrement operators) nor ("classic") section 3.3.3.1
(Prefix increment and decrement operators) contain any constraints
which would prohibit the incrementing or decrementing of pointers
to function types or pointers to incomplete types.

I believe that this logical inconsistancy needs to be addressed (and
rectified) in the standard. It seems that the most appropriate way to
do this is to add the following additional constraint to 3.3.2.4:

> The operand of the posfix increment or decrement operator
> shall not have a type which is a pointer to incomplete type
> or a pointer to function type.

Likewise, the following new constraint should be added to section 3.3.3.1:

> The operand of the prefix increment or decrement operator
> shall not have a type which is a pointer to incomplete type
> or a pointer to function type.

ANSI/ISO C Defect report #rfg6:

Section 3.2.1.5 of the ("classic") ANSI C standard explicitly allows an
implementation to evaluate a floating-point expression using some type
which has MORE precision that the apparent type of the expression itself:

> "The values of floating operands and the results of floating expressions may be represented in greater precision and range than that required by the type."

A footnote on this rule also says explicitly that:

> "Cast and assignment operators still must perform their specified conversions, as described in 3.2.1.3 and 3.2.1.4."

As noted in the first of these two quotes (above) some compilers (most notably for x86 and mx680x0 target systems) may perform floating-point expression evaluation using a type which has more precision and/or range that that of the "apparent type" of the expression being evaluated.

The clear implication of the above rules is that compilers must sometimes generate code to implement narrowing of floating-point expression results, when (a) those results were genereated using a format with more precision and/or range than the "apparent type" of the expression would seem to call for, and where (b) the expression result is the operand of a cast or is used as an operand of an "assignment operator".

My question is simply this: For the purposes of the above rules, does the term "assignment operator" mean exactly (and only) those operators listed in ("classic") section 3.3.16, or should implementors and users expect that other operations described within the standard as being similar to "assignment" will also producing floating-point narrowing effects (under the right conditions)?

Specifically, may (or must) implicit floating-point narrowing occur as a result of parameter passing if the actual argument expression is evaluated in a format which is wider than its "apparent type"? May (or must) implicit floating-point narrowing occur as a result of a return statement if the return statement contains a floating-point expression which is evaluated in some format which is wider than its "apparent type"?

Here are two examples illustrating these two questions. Imagine that these examples will be compiled for a type of target system which is capable of performing floating-point addition ONLY on floating-point operands which are represented in the same FP format normally used to hold type 'long double' operands in C:

```
=====================================================================
extern void callee ();  /* non-prototyped */

double a, b;

void caller ()
{
    callee(a+b);  /* evaluated in long double format then narrowed? */
}
=====================================================================

=====================================================================
double a, b;
```

```
double returner ()
{
    return a+b;  /* evaluated in long double format then narrowed? */
}
```

==========

ANSI/ISO C Defect report #rfg7:

In section 3.6.6.4 (The Return Statement) of the ("classic") ANSI C standardxi it says:

> "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."

This is nonsensical. The type of the containing function is a function type... and that's different from an object type.

I believe that should be changed to read:

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

ANSI/ISO C Defect report #rfg8:

Section 3.3.2.2 (Function Calls) of the ("classic") ANSI C standard says:

> "If the expression that denotes the called function has a type which includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters."

The problem with this statement is the phrase "as if by assignment". The above rule fails to yield an unambiguous meaning in cases where an assignment of the actual to the formal would be prohibited by other rules of the language, as in:

```
void callee (const int formal);
int actual;
void caller () { callee(actual); }
```

(Here, the name of the formal parameter 'formal' may be initialized but not assigned to... because it is a non-modifiable lvalue.)

A similar problem exists within section 3.6.6.4 (The Return Statement) of the ("classic") ANSI C standard. It says:

> "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."

This statement leaves the validity of the following code open to question:

```
const int returner () { return 99; }
```

Last but not least, section 3.5.7 (Initialization) of the ("classic") ANSI C standards says:

> "The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression; the same type constraints and conversions as for simple assignment apply."

This statement leaves the validity of the following code open to question:

```
const int i = 99;
```

(Note that *assignment* to the data object 'i' is not normally permitted, as its name does not represent a modifiable lvalue.)

ANSI/ISO C Defect report #rfg9:

Section 3.5 of the ("classic") ANSI C standard (constraints subsection) says:

> "If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 3.5.2.3."

Section 3.5.2.3 of the ("classic") ANSI C standard (semantics subsection) says:

> "Subsequent declarations {of a tag} in the same scope shall omit the bracketed list."

Given that one of the above two rules appears in a constraints subsection, while the other appears in a semantics subsection, it is ambiguous whether or not diagnostics are strictly required in the following cases (in which more than one defining declaration of each tag appears within a single scope):

```
void example ()
{
    struct S { int member; };
    struct S { int member; };  /* diagnostic required? */

    union U { int member; };
    union U { int member; };   /* diagnostic required? */

    enum E { member };
    enum E { member };         /* diagnostic required? */
}
```

ANSI/ISO C Defect report #rfg10:

According to section 3.5 of the ("classic") ANSI C standard:

> "If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer."

Note that this rule appears in a semantics section, so it would seem that comformant implementations are permitted but not strictly required to produce diagnostics for violations of this rule.

Anyway, my interpretation of the above rule is that conformant implementations are permitted (and even encouraged it would seem) to issue diagnostics for code such as the following, in which formal parameters for functions (which, by definition, have no linkage) are declared to have incomplete types:

```
typedef int AT[];

void example1 (int arg[]);  // diagnostic permitted/encouraged?
void example2 (AT arg);     // diagnostic permitted/encouraged?
```

I believe that section 3.5 needs to be reworded so as to clarify that code such as that shown above is perfectly valid ANSI/ISO C code, and that conformant implementations should not reject such code out of hand.

ANSI/ISO C Defect report #rfg11:

According to section 3.5 of the ("classic") ANSI C standard:

> "If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer."

It would appear that the above rule effectly renders the following code "not strictly conforming" (because this code violates the above rule):

```
typedef struct incomplete_S ST;
typedef union  incomplete_U UT;

void example1 (ST arg);     // diagnostic permitted/encouraged?
void example2 (UT arg);     // diagnostic permitted/encouraged?
```

I have noted however that many/most/all "conforming" implementations do in fact accept code such as that shown above (without producing any diagnostics).

Is it the intention of X3J11 that code such as that shown above should be considered to be "strictly conforming"? If so, then some change to the wording now present in section 3.5 is in order (to allow for such cases).

ANSI/ISO C Defect report #rfg12:

Section 3.5 of the ("classic") ANSI C standard says (in its constraints section):

> "All declarations in the same scope that refer to the same object or function shall specify compatible types."

However in section 3.1.2.6 we have the following rule:

> "All declarations that refer to the same object or function shall have compatible type; otherwise the behavior is undefined."

There is a conflict between the meaning of these two rules. The former rule indicates declaring something in two or more incompatible ways (in a given scope) *must* cause a diagnostic, while the latter rule indicates that doing the exact same thign may result in undefined behavior (i.e. possibly silent acceptance of the code by the implementation).

+ (Note that this same issue was raised previously in the C Information
+ Bulletin #1, RFI #17, question #3. While the response to that question
+ indicated that no change was needed, a change IS clearly need in order to
+ resolve this ambiguity.)
+

Furthermore, the use of the term "refer to" in both of these rules seems both unnecessary and potentially confusing. Why not just talk instead about declarations "declaring" things, rather than "referring to" those things?

To eliminate the first problem I would suggest that the rules quoted above from section 3.1.2.6 should be clarified as follows:

> "If any pair of declarations of the same object or function which appear in different scopes declare the object or function in question to have two different incompatible types, the behavior is undefined."

(Actually the rule regarding declaration compatability which now appears in 3.1.2.6 seems entirely misplaced anyway. Shouldn't it just be taken out of 3.1.2.6 and moved to the section on declarations, i.e. 3.5?)

ANSI/ISO C Defect report #rfg13:

Section 3.2.2.2 of the ("classic") ANSI C standard says:

> "The (nonexistant) value of a void expression (an expression that has type void) shall not be used in any way..."

There are two separate (but related) problems with this rule.

First, it is not entirely clear what constitutes a "use" of a value (or of an expression). In which lines of the following code is a type 'void' value actually "used"?

```
void example (void *pv, int i)
{
    &*pv;           /* ? */
    *pv;            /* ? */
    i ? *pv : *pv;  /* ? */
    *pv, *pv;       /* ? */
}
```

(The answer to this question will determine which of the above lines cause undefined behavior, and which cause well defined behavior.)

If one or more of the (questionable) lines from the above example are judged by the committee to result in well defined behavior, then a second (separate) issue arises. This second issue requires some explaining...

Section 3.2.2.1 of the ("classic") ANSI C standard contains the following rules:

> "An lvalue is an expression (with an object type or an incomplete type other than void)..."
>
> "Except when it is the operand of the 'sizeof' operator, the unary & operator. the ++ operator, the -- operator, or the left operand of the . operator or and assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)... If the lvalue has an incomplete type and does not have array type, the behavior is undefined."

Note that that final rule (specifying a condition under which undefined behavior arises) seems, based upon the context, to only apply to those cases in which "...an lvalue that does not have an array type is converted to the value...". More specifically, it appears that undefined behavior is NOT necessarily produced for non-lvalue expressions (appearing in the indicated contexts).

Furthermore, it should be noted that the definition of an lvalue (quoted above) DOES NOT INCLUDE all void types. Rather, it only includes THE void type.

The result is that the indicated lines in following example would seem to yield well defined behavior (or at least they WILL yield well defined behavior if the committee decides that their unqualified counterparts do), however I suspect that this may not have been what the committee intended.

```
void example (const void *pcv, volatile void *pvv, int i)
{
    &*pcv;              /* ? */
    *pcv;               /* ? */
    i ? *pcv : *pcv;    /* ? */
    *pcv, *pcv;         /* ? */

    &*pvv;              /* ? */
    *pvv;               /* ? */
    i ? *pvv : *pvv;    /* ? */
    *pvv, *pvv;         /* ? */
}
```

In summary, I would ask that the committee comment upon and/or clarify the behavior produced by each of the examples shown herein. Separately, I would request that the committee make changes in the existing standard in order to make the rules applicable to such cases more readily apparent.

ANSI/ISO C Defect report #rfg14:

Section 4.2.1.1 of the ("classic") ANSI C standard says:

> Synopsis
>
> ```
> #include <assert.h>
> void assert (int expression);
> ```

This synopsis raises several related questions.

Question A: May a strictly conforming program contain code which includes an invocation of the assert macro for an expression whose type is not directly convertable to type 'int'? (See examples below.)

Question B: Must a conforming implementation issue diagnostics for any and all attempts to invoke the assert macro for an expression having some type which is not directly convertable to type 'int'?

Examples:

```
#include <assert.h>

char *cp;
void (*fp) ();
struct S { int member; } obj;

void example ()
{
    assert (cp);     /* conforming code?  diagnostic required? */
    assert (fp);     /* conforming code?  diagnostic required? */
    assert (obj);    /* conforming code?  diagnostic required? */
}
```

Question C: Must a conforming implementation convert the value yielded by the expression given in an invocation of the assert macro to type 'int' before checking to see if it compares equal to zero?

Example:

```
#include <assert.h>

void example ()
{
    assert (0.1);    /* must this casue an abort?  must it NOT? */
}
```

ANSI/ISO C Defect Report #rfg15:

Section 4.1.2.1 of the ("classic") ANSI C lists the set of reserved identifiers, but this list does not include keywords (3.1.1).

Section 3.1.1 of the ("classic") ANSI C standard says (in a semantics subsection):

> "The above tokens (entirely in lower-case) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise."

Based upon the above named sections of the standard, I am forced to conclude that the following code is "strictly conforming". Is this a correct conclusion?

```
#define double void
#include <math.h>

void example (double d1, double d2)
{
    d1 = acos (d2);
}
```

My impression is that few (if any) existing implementations now accept such code. I am therefore inclined to believe that the committee's true intentions were that ALL keywords (as listed in 3.1.1) should be considered to be reserved identifiers, at least during translation phase 4, and at least while processing #include directives which name standard include files provided by the implementation (as listed in 4.1.2).

I believe that the proper way to address this problem would be to add another stipulation (regarding reserved identifiers) to section 4.1.2.1. This additional stipulation might read as follows:

> If, during inclusion of any one of the standard headers listed in the preceeding section (during translation phase 4) any one of the keywords listed in section 3.1.1 is defined as a preprocessor macro, the behavior is undefined.

ANSI/ISO C Defect Report #rfg16:

Does the standard draw any significant distinction between "undefined values" and "undefined behavior"?

(It appears that it does, but if it does, that fact is not always apparent.)

Just to give two examples which, it would appear, involve the generation (in a running program) of undefined values (as opposed to totally undefined behavior at either compile-time or link-time or run-time) I provide the following two citations...

ISO C standard; 6.3.8 (Relational Operators):

> "If the objects pointed to are not members of the same aggregate or union object, the RESULT is undefined,..."

(Emphasis added.)

ISO C standard; 7.5.2.1 (The ACOS Function):

> "A domain error occurs for arguments not in the range [-1,+1]."

The issue of "undefined values" versus "undefined behavior" has great significance and importance to people doing compiler testing. It is generally accepted that the standard's use of the term "undefined behavior" is meant to imply that absolutely *anything* can happen at any time, e.g. at compile-time, at link-time, or at run-time. Thus, people testing compilers must either totally avoid writing test cases which involve any kind of "undefined behavior" or else they must treat any such test cases which they DO write as strictly "quality of implementation" tests which may validly cause errors at compile-time, at link-time, or at run-time.

If however the standard recognizes the separate existance of "undefined values" (whose mere creation DOES NOT involve wholly "undefined behavior") then a person doing compiler testing could write a test case such as the following, and he/she could also expect (or possibly demand) that a "conformant" implementation should, at the very least, compile this code (and possibly also allow it to execute) without "failure".

```
int i;
double d;

int array1[5];
int array2[5];
int *p1 = &array1[0];
int *p2 = &array2[0];

int main ()
{
    i = (p1 > p2);  /* Must this be "successfully translated"? */
    d = acos (2.0); /* Must this be "successfully translated"? */
    return 0;
}
```

So the bottom line question is this: Must the above code be "successfully translated" (whatever that means)? (See the footnote attached to section 5.1.1.3 of the ISO C standard.)

ANSI/ISO C Defect report #rfg17:

Subject: Formal parameters having array-of-non-object types.

Question #1: For which (if any) of the following function declarations and definitions is a diagnostic required?

Question #2: Which (if any) of the following function declarations and definitions would, if present in a translation unit, render the translation unit "not strictly conforming"?

```
typedef void VT;
typedef struct incomplete_S ST;
typedef union  incomplete_U UT;
typedef int AT[];
typedef void (FT) ();

void declaration1 (VT arg[]);       /* ? */
void declaration2 (ST arg[]);       /* ? */
void declaration3 (UT arg[]);       /* ? */
void declaration3 (AT arg[]);       /* ? */
void declaration3 (FT arg[]);       /* ? */

void definition1 (VT arg[]) { }     /* ? */
void definition2 (ST arg[]) { }     /* ? */
void definition3 (UT arg[]) { }     /* ? */
void definition3 (AT arg[]) { }     /* ? */
void definition3 (FT arg[]) { }     /* ? */
```

Footnote: I have heard rumors that the issue of the exact timing of the decay of a formal parameter's array type into a pointer type (relative to the timing of the necessary check that the type of the formal parameter is in fact a valid type) was determined EXPLICITLY to be undefined by X3J11, but there is no record of this in the CIB #1 document I have.

References: CIB #1, RFI #13, question #1
CIB #1, RFI #17, question #14
CIB #1, RFI #17, question #15

ANSI/ISO C Defect report #rfg18:

Subject: Conversion of pointer-to-qualified type values to type (void*) values.

Question: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

```
const char *ccp;
void *vp;

void
test ()
{
    vp = ccp;   /* diagnostic required? */
}
```

With respect to this example, the following quotations are relevant.

Section 6.2.2.3; ISO C standard:

> "A pointer to void may be converted to or from a pointer to any incomplete or object type."

Section 6.3.16.1; ISO C standard (constraints):

> "One of the following shall hold:
>
> ...
>
> - both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;
>
> - one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void, and the type pointed to by the left has all the qualifiers of the type pointed to by the right..."

The rule specified in section 3.2.2.3 (and quoted above) makes it unclear whether a value of some pointer-to-qualified-object type may be FIRST implicitly converted to type (void*) and then assigned to a type (void*) variable, or whether such implicit conversion only takes place as an integral part of an otherwise valid assignment operation.

If the former interpretation of section 6.2.2.3 is correct, then the above code example is valid, and no diagnostic is required. If however the latter interpretation is the correct one, then the code example shown above fails to meet the constraints of section 6.3.16.1, and (thus) a diagnostic is required.

ANSI/ISO C Defect report #rfg19:

Subject: Null pointer constants and relational comparisons.

Question #1: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

Question #2: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
void test (void *vp)
{
  (vp > (void*)0);    /* ? */
}
```

Background:

Section 6.2.2.3; ISO C standard:

> "An integral constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.

> If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type."

This last paragraph of 6.2.2.3 seems to suggest that zero valued integral constant expressions which are cast to void* (and then called null pointer constants) can *only* be used in assignments and/or equality comparisons, but not in relational comparisons.

(It was probably the committee's intent to permit such expression to be used in all ways, and in all contexts where any other type void* non-lvalued expressions can be used, but the current wording of 6.2.2.3 does not make that fact altogether apparent and unambiguous.)

ANSI/ISO C Defect report #rfg20:

Subject: Return expressions in functions declared to return qualified void.

Question #1: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

Question #2: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
volatile void func0 (voiatile void *vvp)
{
        return *vvp;    /* ? */
}

const void func1 (const void *cvp)
{
        return *cvp;    /* ? */
}
```

Background:

Section 6.6.6.4; ISO C standard (constraints):

> "A return statement shall not appear in a function whose return type is void."

Note that this constraint doesn't say anything about functions declared to return some qualified version of the void type.

I believe that it was probably the committee's true intent to require a diagnostic for any attempt to specify an expression in a return statement within any function declared to return *any* qualified or unqualified version of the void type (and indeed, many existing implementations do already issue diagnostics for usage such as that shown in the example above). Thus, it would seem appropriate for the committee to ammend the above quoted constraint (from 6.6.6.4) to read:

> "A return statement shall not appear in a function whose return type is a void type."

ANSI/ISO C Defect Report #rfg21:

Subject: Initialization of multi-dimensional char array objects.

Question #1: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

Question #2: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
    char array2[2][5] = { "defghi" };   /* ? */
```

Background:

Section 6.5.7; ISO C standard (constraints):

> "There shall be no more initializers in an initializer list than there are objects to be initialized."

Section 6.5.7; ANSI C standard:

> "An array of character type may be initialized by a character string literal, optionally enclosed in braces."

Section 3.5.7; ANSI C standard (examples):

> "...It defines a three-dimensional array object..."

It appears that many existing compilers seem to feel the the code example shown above violates the "no more initializers" constraint (quoted above) which is given in section 6.5.7.

Note however that the ENTIRE two-dimensional array object being initialized consists of exactly 2*5 = 10 individual char objects, whereas the initializer itself only consists of 7 individual char values (if one counts the terminating null byte). Thus, it would appear that these existing implementations are in fact WRONG in rejecting the above code, and that such usage is in fact strictly conforming.

I ask the committee to unambiguously either confirm or refute that position.

ANSI/ISO C Defect Report #rfg22:

Subject: Member declarators as declarators.

Question #1: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

Question #2: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
   struct { int mbr; };      /* ? */
```

```
union { int mbr; };      /* ? */
```

Background:

Section 6.5; ISO C standard (constraints):

> "A declaration shall declare at least a declarator, a tag, or the members of an enumeration."

It is not entirely clear what it means to "declare" a declarator. Neither is it clear whether or not a declarator for a member should be considered to satisfy the constraint quoted above. (Many existing implementations behave as if member declarators DO NOT satisfy the constraint.)

ANSI/ISO C Defect Report #rfg23:

Subject: Implicit unary & applied to register arrays.

Question #1: Does the following code involve usage which requires a diagnostic from a standard conforming implementation?

Question #2: Does the following code involve usage which renders the code itself "not strictly conforming"?

```
void example ()
{
    register int array[5];

    array1;        /* ? */
    array2[3];     /* ? */
    array3+3;      /* ? */
}
```

Background:

Section 6.5.1; ISO C standard (footnotes):

> "The implementation may treat the register declaration simply as an auto declaration. However whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier 'register' may not be computed, either explicitly (by use of the unary & operator as discussed in 3.3.3.2) or implicitly (by converting an array name to a pointer as discussed in 3.2.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier 'register' if 'sizeof'."

This footnote, while offering guidance doesn't really answer the question of whether or not an implementation is required to issue a diagnostic for the case where the address of a register array is implicitly taken (as discussed in 6.2.2.1). Nor does it definitively answer the question of whether such code should be considerd to be strictly conforming or not.

(Reference: CIB #1, RFI #17, question #7.)

-- Ronald F. Guilmette, Sunnyvale, California ----------
------ domain address: rfg@netcom.com ----------
------ uucp address: ...!uunet!netcom.com!rfg ----------