N514/N533
X3J11/94-008

Record of Responses #1

This document presents all the responses to date of ISO committee JTC1/SC22/WG14 (Programming language C) to Defect Reports #001 through #059 for International Standard ISO/IEC 9899:1990. As a Record of Responses, this document is *not* normative; rather, it provides guidance on how to interpret the ISO C Standard.

That guidance was crafted by technical experts from a number of ISO member nations. In particular, WG14 solicited, and received, extensive assistance from the ANSI authorized committee X3J11, which developed the ANSI C Standard that became the ISO C Standard.

To form a coherent history of Defect Reports and their disposition, this Record of Responses also records a number of responses that call for normative changes to the ISO C Standard. A separate document, called a Technical Corrigendum, makes those normative changes. In this Record of Responses, such an item from the Technical Corrigendum is identified as a **Correction**.

WG14 has developed no response yet for Defect Report #056. At the request of X3J11, and others, WG14 has elected to allow additional time for technical experts to consider the issues. Nevertheless, that Defect Report is also included in this document, again in the interest of completeness. It is labeled as an **Open Issue**.

Every other committee interpretation is labeled as a **Response** in this document.

This Record of Responses includes a **Summary of Issues**, to assist the reader in locating areas of particular interest.

Defect Report #001

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-009 (Paul Eggert)

Question 1

Do functions return values by copying?

The C Standard is clear (in subclause 6.3.2.2) that function arguments are copied, but is not clear (in subclause 6.6.6.4) whether a function's returned value is also copied. This question becomes an issue in the assignment statement `s=f();` where `f` yields a structure: is the result defined when the structure `s` overlaps the structure that `f` obtained the returned value from?

I ask this question because the GNU C compiler does not copy the structure in this case. When I filed the enclosed bug report [omitted from this document], Richard Stallman, the author of GNU C, replied that he didn't think that Standard C required the extra copy. I sympathize with Stallman's desire for efficient code, and I also would prefer that the C Standard did not require the extra copy here, but the point should be made clear in the C Standard.

Correction

In subclause 6.6.6.4, page 80, 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, page 80:

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.

Defect Report #002

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-010 (Terence David Carroll)

Question 1

Subclause 6.8.3.2: Semantics of #

A minor detail in the semantics is that it does not explicitly state that a \ character will be inserted before a \ character that occurs within a macro actual parameter, only when the \ character occurs within a string literal or character constant within the actual parameter.

I can see that there is an argument concerning the systems where \ is a valid part of a path name and where `#include` path names are desired to be built dynamically and then #ed.

Would it not be better, however, to escape all \ within actual parameters and require either

a) that systems using \ in path names escape them within `#include` strings, or perhaps

b) that during macro processing of `#include` parameters the # operator will not cause \ characters to be escaped at all or will be implementation defined?

This second case (b) is better, as strings for `#include` directives are already a special case (no escape processing, etc.), so should not the # operator also be special for `#include` directives?

Response

The Committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous response to this item from David F. Prosser (Editor of the standard). The Committee endorses the substance of this response, which follows:

The rules in subclause 6.8.3.2 regarding \ were discussed in the Committee and the result is as intended. The Committee's charter was to standardize prior art where such was clear, and the behavior of those C preprocessing phases that allowed tokens such as \ left them alone, even when inserting them into strings. However, the Committee also had license to fix or add to the language where it was judged to be deficient. Since none of the existing stringizing preprocessing phases correctly handled string literals and certain character constants, the special rules for these were chosen.

Subclause 6.8.3's examples (page 92, lines 4-33) include a \ that is outside of a string literal or character constant. If the rules were to be modified along the lines of your proposal, the intended effect would not happen.

One of the main points in your argument is that uncontained \s are only critical in path names that use \ as a special character, and that this is only needed when `#include` filenames are constructed via macro replacement. I agree that the current rules do allow this sort of use without too much trouble, but I don't see this as being a main motivation for this feature. By default, the rules for stringizing were that the original spelling of the invocation argument is placed into a string literal. The only exceptions made to this were those that were valid C tokens that could not be simply inserted between a pair of "s. The rules for \ and " within string literals and character constants were derived from that need only. Furthermore, a lone \ is a preprocessing token due to the "some other character" rule of the syntax from subclause 6.1. This would be the only place where special constraints were placed on one of this type of preprocessing token.

Finally, solution b) of your discussion involves context-dependent rules for the stringizing operation. While there is a minor context dependency regarding macro replacement and the `defined` unary operator on `#if` and `#elif` directive lines, this is the only context dependency in the whole set of macro replacement rules. Moreover, this dependency is at the topmost level only. Solution b) would require a flag noting whether the result of the replacement was to be used within a `#include` directive. Therefore, the same macro invocation would produce different results at different invocations. (At the least, debugging and/or testing of a tricky macroized `#include` directive would be more difficult.)

In conclusion, to the best of my knowledge, the Committee wants to keep the behavior here as currently described, and made this choice intentionally.

Defect Report #003

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-011 (Terence David Carroll)

Question 1

Subclause 6.1.8: Preprocessing numbers

I note from the rationale document of November 1988, X3J11 Document Number 88-151, that the following problem has been observed. I am surprised at the Committee's decision to allow such a loose description.

Under the grammar given for a ***pp-number***

```
0xEE+23    0x7E+macro    0x100E+value-macro
```

are preprocessing numbers and as such a conforming C compiler would be required to generate an error when it failed to successfully convert them to actual C language number tokens.

The solution is simply to restrict the inclusion of [**eE**][**+-**] within a ***pp-number*** to situations where the **e** or **E** is the first ***non-digit*** in the character sequence composing the preprocessing number. This can be easily implemented in a variety of methods; the informal description above gives perhaps a better guide to efficient implementation than the following revised grammar:

```
pp-number:
      pp-float
      pp-number digit
      pp-number .
      pp-number non-digit /* A non-digit is a letter or underscore */

pp-float:
      pp-real
      pp-real E sign
      pp-real e sign

pp-real:
      digit
      .
      pp-real digit
      pp-real .
```

It is unbelievable that a standards committee could so lose sight of its objective that it would, in full awareness, make simple expressions illegal.

To illustrate the absurdity of the rationale document's claim that the faulty grammar was felt to be easier to implement, why not adopt the following grammar for a ***pp-number*** and really make life simple; after all, who wants to have their preprocessor slowed down by checking whether the + or - was preceded by an **e** or an **E**?

```
pp-number:
      digit
      .
      pp-number digit
      pp-number .
      pp-number non-digit
      pp-number sign
```

Response

The Committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous responses to this item from David F. Prosser. The Committee endorses the substance of these responees, which follow:

In response to your first suggested grammar: This grammar doesn't include all valid numeric constants and exclude other important tokens. For example, . is derivable. But let's assume that you intended something like

```
pp-number:
      pp-float
      digit
      pp-number digit
      pp-number non-digit

pp-float:
      pp-real
      pp-float E sign
      pp-float e sign
      pp-float digit
      pp-float .
      pp-float non-digit

pp-real:
      digit
      . digit
      pp-real digit
      pp-real .
```

This grammar is certainly more complicated than the one-level construction in the C Standard, and consequently harder to understand. That's a strike against it.

Another strike is that, while it does mimic the two major numeric categories, it still doesn't include all sequences covered by the existing grammar, save those that would otherwise be valid by the stricter tokenization rules. For example, `0b0101e+17` might be someone's future notion of a binary floating constant.

Finally, it suffers from a great deal of reduce/reduce conflicts, making the implementation and specification less likely to be understood and implemented as intended.

In response to your second suggested grammar: This could have been done. But the Committee chose a compromise at a different point — one that restricts the inappropriate gobbling of characters to + and - immediately after `E` or `e`. This was all that was necessary to cover all valid numeric constants in as simple a grammar as was possible.

For more background, you'd need to know the state of the proposed standard a few years before this grammar was voted in. The Committee had stated its intent that "garbage" character sequences that began like a numeric constant were to be tokenized as a single sequence. This was to prevent situations in which this "garbage" would be turned into valid C code through obscure macro replacements, among more minor reasons. This was, unfortunately, very poorly stated in the draft. As I recall, it was placed in the constraints for subclause 6.1. It was something like "Each pair of adjacent tokens that are both keywords, identifiers, and/or constants must be separated by white space." [As "improved" for the May 1, 1986 draft proposed standard, subclause 6.1 **Constraints** consisted of the single sentence: "Each keyword, identifier, or constant shall be separated by some white space from any otherwise adjacent keyword, identifier, or constant."]

As you can see, this constraint neither presented the intent of the Committee nor caused implementations to behave in any sort of consistent manner with respect to tokenization.

Finally a letter writer understood the issue well enough to suggest a grammar along the lines of the current subclause 6.1.8. It, contrary to your opening remarks on this topic, is *not* a "loose description," and it finally stated in a precise way the intent of the tokenization rules.

The benefits of this construction were that all tokenization for all implementations would now be the same, no "garbage" character sequences would be able to be converted to valid C code, skipped blocks of code could silently be scanned withou generating needless and unnecessary tokenization errors, the preprocessing tokenization of numeric tokens would be greatly simplified, and room for future expansion of C's numeric tokens was reserved.

That's a lot of good. The down side was that certain sequences now would require some white space to cause them to be tokenized as the programmer intended. As noted in the rationale document, there are other parts in C that require white space for tokenization to be controlled, and this was found to be one more.

Since the "mistokenization" of such sequences must result in some diagnostic noise from the compiler, and since the fix is so mild, the Committee agreed that the proposed standard is still much better with this grammar than with any of the other suggestions.

Personally, I think that the biggest surprise "win" was the reservation of future numeric token "name space." I would not be at all surprised to find binary constants (that begin with 0b) in newer C implementations.

Question 2

Subclause 6.8.3: Macro substitutions, tokenization, and white space

In general I think it is a good guiding principle that a C implementation should be able to be based around completely disjoint preprocessing and lexical scanning parses of the compiler. As such the rules on tokenizing need to be emphasized with the following paragraphs (possibly placed after paragraph 1 of subclause 6.8.3.1):

> All macro substitutions and expanded macro argument substitutions will result in an additional space token being inserted before and after the replacement token sequence where such a space token is not already present and there is a corresponding preceding or subsequent token in the target token sequence.
>
> The last token of every macro argument has no subsequent token at the time of its initial macro argument expansion, and similarly a macro parameter that is the last token of a replacement token list has no subsequent token at the time of that parameter's substitution. Similarly for first tokens and preceding tokens.

Naturally such a step can be treated as purely conceptual by a tokenized implementation with combined preprocessing and lexical analysis, except for the purposes of argument stringizing where the added spacing may be essential for unambiguous identification of the preprocessing tokens involved.

Such a statement is not a substantive change, as it is merely clarifying the tokenization rules, and given that Standard C has changed the definition of the preprocessor substantially from K&R already (re macro argument expansion before substitution) such an additional explicit change from K&R C should cause comparatively little difficulty except to those who had not appreciated just how different the preprocessing rules are already.

Examples which are clarified by this change are:

```
#define     macro     +
            +macro
            macro+
#define     mac()     +
#define     ro        +
            mac()ro
```

all of which unambiguously result in lines with two + operator tokens, in strict accordance with the draft standard's tokenization rules, and not, as was formerly the case with traditional text oriented preprocessors, in single ++ operators.

Examples which are changed by this statement are:

```
#define     mac()             +
#define     ro                +
#define     str(s)            # s
#define     eval(m,e)         m(e)
            eval( str, mac()ro )
```

which produces the string "+ +" and not the string "++" as it would do with the draft's current wording.

Response

The Committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous responses to this item from David F. Prosser. The Committee endorses the substance of these responses, which follow:

K&R never specified the macro replacement algorithm to the extent that any such conclusion is possible. The widest range of implementation choices were present in this area of the language. The eventual choice of a macro replacement algorithm was one that did not match any existing implementation, but one that tried to include the behavior of all major variants.

You agree that the C Standard is clear that once a token is recognized, it is never retokenized unless subjected to a # or ## operation. The behavior described is that which was chosen by the Committee. Your proposal would cause, as you note, certain created string literals to include white space not present in the original text. This runs counter to the # operator's goal of producing a string version of the spelling of the invocation arguments.

The C Standard allows an implementation that uses a text-to-text separate preprocessing stage the option to use white space as necessary to separate tokens when it produces its output. However, this insertion of white space must not be visible to the program. The proposed extra white space would probably be a surprise to the programmer as well. Finally, this proposal would require those implementations that have a built-in preprocessing stage to add extra code to insert white space in special circumstances. This is counter to the goal of having both built-in and separate implementations be purely an implementation choice.

Question 3

Subclause 6.8.3: Empty arguments to function-like macros

I would like to make a request for clarification and a request for a stronger statement of standardization. Given

```
#define macro( xx ) xx
        macro()
```

is this a constraint violation of subclause 6.8.3 **Constraints** paragraph 4:

> The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, ...

or is this an undefined, implementation-dependent program — subclause 6.8.3, **Semantics** paragraph 5:

> If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined.

In connection with the above I would request that the Committee make a much stronger statement as to whether empty arguments are to be treated as valid arguments or are to be treated as errors. They can have their uses, but if that is recognized then it should be standardized; if not, it should not be allowed.

Response

If one takes the general case, empty arguments in invocations of function-like macros are easy to recognize:

```
#define f(a,b,c) whatever

    f(,,)
```

These empty arguments all have "shadows" that cause the sentence you reference in subclause 6.8.3 (page 90, lines 4-5) to be clearly in effect.

The only uncertain case is one in which an empty argument in an invocation of a one-parameter function-like macro mimics a "no arguments" invocation. (It should also be noted that the definition of a macro argument from clause 3 does not preclude an empty sequence.)

Thus the standard is clear that the behavior is undefined in the example from your request. If an implementation does not choose to allow empty arguments, a diagnostic will probably be emitted; otherwise, the invocation will most likely be replaced by a preprocessing token sequence in which the parameter is replaced with no tokens. But because the standard does not define this, other than as a common extension, there are no guarantees.

Question 4

Subclause 6.8.3: Preprocessor directives within actual macro arguments

It is a guiding principle that a macro function and an actual function should be invokable in as similar fashion as possible. In the latter case, it is not uncommon to find code with variations of arguments subject to conditional compilation. This should also compile correctly if an appropriate macro definition is made for the function.

While conditional compilations within function arguments is not necessarily a programming style that I would condone, I feel that it is in the interests of the C programming community at large for such constructs to be well formed, even if forbidden, and as such make the following requests:

I would like the Committee to change subclause 6.8.3 to state that **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#elif`**, and **`#endif`** preprocessing directives are allowed within actual macro arguments (not necessarily cleanly nested).

Conversely, I would like **`#define`** and **`#undef`** to be formally forbidden within macro invocations, as these can result in effects that are dependent on the particular implementation of the macro expansions.

Response

The Committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous response to this item from David F. Prosser. The Committee endorses the substance of this response, which follows:

The equivalent of your proposal was rejected a couple of years ago. Certain Committee members felt that requiring all preprocessors to recognize these lines as directives was too much. Those that felt that these lines must be recognized were finally convinced that it was enough to allow implementations the right to behave in the more orthogonal manner. (Maybe they figure that the next version of the standard will have to require this sort of behavior, as all "reasonable" implementations already will have it by then.)

Defect Report #004

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-012 (Paul Eggert)

Question 1

Are multiple definitions of unused identifiers with external linkage permitted?

The wording in subclause 6.7 permits multiple definitions of identifiers with external linkage, so long as the identifiers are never used. For example, the following program is "strictly conforming" if you read the wording in subclause 6.7 literally:

```
int F() {return 0;}
int F() {return 1;}
int V = 0;
int V = 1;
int main() {return 0;}
```

This must be a bug in the wording of subclause 6.7. It cannot have been the Committee's intent, since it prohibits the most commonly encountered linker model. For example, most linkers will flatly refuse to link the following "strictly conforming" program

x.c:

```
int F() {return 0;}
int G(int i) {return i;}
```

y.c:

```
int F() {return 1;}
int G(int);
int main() {return G(0);}
```

because `F` is defined twice.

Response

This Defect Report referred to an earlier draft of the C Standard, and was corrected prior to the publication of the C Standard.

Defect Report #005

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-020 (Walter J. Murray)

Question 1

According to subclause 6.8.6, a pragma directive "causes the implementation to behave in an implementation-defined manner." May a conforming implementation define and recognize a pragma which would change the semantics of the language? For example, might a conforming implementation recognize and honor the directive

```
#pragma UNSIGNED_PRESERVING
```

as a way for a program to request non-standard integral promotions?

I also pose the corollary question. May a strictly conforming program contain a pragma directive? According to subclause 4, a strictly conforming program "shall use only those features of the language ... specified in this standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior..."

If there is no constraint on how a conforming implementation may behave when encountering a pragma directive, would it not follow that a strictly conforming program may not contain a pragma directive?

Response

The relevant citations are subclause 6.8.6:

> A ... `pragma` ... causes the implementation to behave in an implementation-defined manner.

and clause 4:

> A strictly conforming program ... shall not produce output dependent on any ... implementation-defined behavior ...

In response to each question:

1) Yes, a conforming implementation may define and recognize a pragma which would change the semantics of the language.
2) Yes, for example, it might honor `UNSIGNED_PRESERVING`.
3) No, a strictly conforming program may not contain a pragma directive.
4) We agree with your conclusion, reasserting answer number 3.

Defect Report #006

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-020 (Walter J. Murray)

Question 1

It is unclear how the `strtoul` function behaves when presented with a subject sequence that begins with a minus sign. The `strtoul` function is described in subclause 7.10.1.6, which contains the following statements.

> If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.
>
> If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is stored in `errno`.

Assume a typical 32-bit, two's-complement machine with the following limits.

```
LONG_MIN  -2147483648
LONG_MAX   2147483647
ULONG_MAX  4294967295
```

Assuming that the value of `base` is zero, how should `strtoul` behave (return value and possible setting of `errno`) when presented with the following sequences?

Case 1: `"-2147483647"`

Case 2: `"-2147483648"`

Case 3: `"-2147483649"`

Response

The relevant citations are the ones supplied by you from subclause 7.10.1.6:

> If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.
>
> If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is stored in `errno`.

The Committee believes that there is only one sensible interpretation of a subject sequence with a minus sign: If the subject sequence (neglecting the possible minus sign) is outside the range [0, `ULONG_MAX`], then the range error is reported. Otherwise, the value is negated (as an `unsigned long int`).

The answers to your numeric questions are:

Case 1: 2,147,483,649

Case 2: 2,147,483,648

Case 3: 2,147,483,647

Defect Report #007

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-043 (Paul Eggert)

Question 1

Are declarations of the form ***struct-or-union identifier ;*** permitted after the ***identifier*** tag has already been declared? Here are some examples of the problem:

```
/*1*/ struct s;
/*2*/ struct s;
/*3*/ struct s {int a;};
/*4*/ struct s;

/*5*/ struct t {int a;};
/*6*/ struct t;
```

Subclause 6.5 says "A declaration shall declare at least a declarator, a tag, or the members of an enumeration." In this sense, does `/*2*/` also declare the tag `s`? If so, then surely all of the above lines are conforming. But if not, then in what sense does `/*3*/` declare a tag and thus satisfy subclause 6.5's constraint?

The example at the end of subclause 6.5.2.3 says "To eliminate this context sensitivity, the otherwise vacuous declaration `struct s2;` may be inserted ..." This seems to imply that `/*2*/`, `/*4*/`, and `/*6*/` are not conforming, because they are vacuous. But how can this be reconciled with the above argument?

Response

The declaration

```
struct s;
```

declares the tag `s`. It need not be the first or only declaration of the tag `s` within a given scope to qualify as a declaration of `s`, just as

```
int i;
```

declares `i` however often it is repeated. The applicable constraint is in subclause 6.5: "A declaration shall declare at least a declarator, a tag, or the members of an enumeration." Clearly,

```
struct s;
```

declares the tag `s`.

Subclause 6.5.2.3, in the examples, characterizes a declaration of this form as "otherwise vacuous" in the draft you read. The words "otherwise vacuous" were an editorial comment that was omitted from the International Standard. These words were intended to mean "other than declaring `s2` to be an (incomplete) struct type," and should not be read as saying that the declaration fails to declare the tag.

We believe that this interpretation is consistent with the intent of the Committee, and that a reasonable reading of the standard supports this interpretation.

Defect Report #008

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-021 (Otto R. Newman)

Question 1

Could you tell me if it is legitimate for a conforming C compiler to perform what's commonly referred to as dead-store elimination for the first assignment in the following code fragment:

```
auto int flag; /* non-volatile */
...
flag = 1;
flag = f();
```

If it is valid to do so, then consider

```
auto int flag; /* non-volatile */
if (setjmp(buf))
   {
    if (flag == 1) ...
   }
flag = 1;
flag = f();
```

where function `f` invokes `longjmp`. Is the result of the relational expression defined? A solution might be to define `flag` as `volatile`, but `flag` is *not* really volatile, and the programmer may not wish to degrade all references to `flag` nor to locate all such possible `flag`s and lie about their volatility.

A related issue is that in many existing applications, users have coded `setjmp`-like mechanisms based on a particular operational environment. The functions do not have the name "`setjmp`," but essentially establish an externally accessible entry point within the containing function. Sometimes, pointers are set to reference such functions, even though the standard precludes this from being done with `setjmp` itself since it is allowable that it only be provided as a macro.

There are a number of additional optimizations which must be inhibited across the actual invocation of `setjmp`, or a `setjmp`-like function. Always avoiding these optimizations as well as the dead-store elimination shown in the example may make the program safe for non-local jumps, but unnecessarily penalizes programs that don't use `setjmp`. To circumvent this problem, some implementors have defined a pragma which is included in `setjmp.h` to identify "`setjmp`" as having the property of establishing an externally accessible entry, i.e., defining an otherwise non-obvious point of control flow. Other implementations have hard-coded tests for the name "`setjmp`."

... would you please respond to the question regarding the legitimacy of the optimization in the first example?

Response

The relevant citation is subclause 7.6.2.1:

> All accessible objects have values as of the time `longjmp` was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding `setjmp` macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call are indeterminate.

In response to your question about the effect on optimizations of `setjmp`: Yes, it is legitimate for a compiler to perform optimizations that eliminate dead stores to local, non-volatile, automatic variables when `setjmp` is used. Subclause 7.6.2.1 makes the values of all such variables indeterminate after the `longjmp` is called. This grants a compiler the liberty to perform dead-store elimination as well as several other optimizations.

Question 2

What is happening is that, since the standard has not provided a mechanism to describe a very recognizable and very important property of a function, such mechanisms are by necessity being provided in non-standard ways. My understanding is that a pragma should never be required for a program to execute correctly as defined by the standard.

The existing situation serves to reduce portability of C programs. We believe the Committee should address this problem and would like to offer a suggestion which seems rather attractive.

Currently, defining an object as `volatile` indicates to the compiler that its contents may be altered in ways not under control of the implementation. This is meaningless with function declarations since a function doesn't have alterable contents (i.e., is not an lvalue). Instead, it may be possible to utilize this otherwise syntactic no-op by defining a "volatile function" to be one whose return may not necessarily occur sequentially at the point of the invocation, but possibly at some other point where the state of the calling program is unknown. In other words, invocation of such a function results in the state of the program becoming volatile.

Now, I admit that this is not a perfectly "clean" extrapolation of the use of the type qualifier `volatile`, but it is rather compelling, having the following advantages:

1) It solves the described problem in a general way that can be used with functions not necessarily named "`setjmp`." Implementations defining `setjmp` as a function in `setjmp.h` would simply declare

```
int volatile setjmp(jmp_buf env);
```

2) It utilizes an existing keyword and gives meaning to its use in a context which would be otherwise meaningless.

3) It is consistent with the type specifier syntax to distinguish between volatile pointers and pointers to volatile objects. For example,

```
int volatile setjmp();
```

defines `setjmp` to be a volatile function (i.e., a function whose invocation must inhibit certain optimizations).

```
int volatile (*maybe_setjmp_ptr)();
```

defines a pointer to such a function, while

```
int (*mustnotbe_setjmp_ptr)();
```

defines a pointer to a normal function.

```
int (* volatile vol_mustnotbe_setjmp_ptr)();
```

defines a volatile pointer to a normal function.

```
int volatile (* volatile vol_maybe_setjmp_ptr)();
```

defines a volatile pointer to a volatile function, and so on ...

4) Type consistency rules are already in place and make sense. For example,

```
maybe_setjmp_ptr = mustnotbe_setjmp_ptr;
```

is okay with no type-checking violation, whereas

```
mustnotbe_setjmp_ptr = maybe_setjmp_ptr;
```

is diagnosed. It would require casting such as

```
mustnotbe_setjmp_ptr = (int (*)())maybe_setjmp_ptr;
```

5) Since no new syntax or keywords are required, the impact of this change is very small to both the document defining the standard and to compilers which support it.

If there is enough Committee interest in this sort of solution, I would be glad to draft a formal proposal.

Response

The Committee reasserts that the current semantics for type qualifiers as they appear in the standard are as intended.

Defect Report #009

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-023 (Bruce Blodgett)

Question 1

Use of typedef names in parameter declarations

A syntactic ambiguity exists in the draft proposed C standard for which there appears to be no semantic disambiguation. A sequence of examples should explain the ambiguity. This matter needs interpretation by the Committee.

For these examples, let `T` be declaration specifiers which contain at least one type specifier, to satisfy the semantics from subclause 6.5.6:

> If the identifier is redeclared in an inner scope ..., the type specifiers shall not be omitted in the inner declaration.

Let `U` be an identifier which is a typedef name at outer scope and which has not (yet) been redeclared at current scope. A caret indicates the position of each abstract declarator. Consider this declaration:

```
declaration-specifiers direct-declarator (T^(U));
```

Here `U` is the type of the single parameter to a function returning type `T`, due to a requirement from subclause 6.5.4.3:

> 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.

Consider this declaration:

```
declaration-specifiers direct-declarator
    (T^(U^(parameter-type-list)));
```

In this example, `U` could be the type returned by a function which takes *`parameter-type-list`*. This in turn would be the single parameter to a function returning type `T`.

Alternatively, `U` could be a redundantly parenthesized name of a function which takes *`parameter-type-list`* and returns type `T`.

Given the spirit of the requirement from subclause 6.5.4.3, the former interpretation seems to be that intended by the Committee. If so, the requirement may be changed to something similar to:

> In a parameter declaration, a direct declarator which redeclares a typedef name shall not be redundantly parenthesized.

Of course, parentheses must not be disallowed entirely... [The original had more, but this will suffice.]

Correction

In subclause 6.5.4.3, page 68, 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.

Defect Report #010

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-044 (Michael S. Ball)

Question 1

Consider:

```
typedef int table[];        /* line 1 */

table one = {1};      /* line 2 */

table two = {1, 2}; /* line 3 */
```

First, is the typedef to an incomplete type legal? I can't find a prohibition in the standard. But an incomplete type is completed by a later definition, such as line 2, so what is the status of line 3?

The type, of which `table` is only a synonym, can't be completed by line 2 if it is to be used in line 3. And what is `sizeof(table)`? What old C compilers seem to do is treat the typedef as some sort of textual equivalent, which is clearly wrong.

Response

A typedef of an incomplete type is permitted.

Regarding objects `one` and `two`, refer to the standard subclause 6.1.2.5, page 24, lines 8-9: "An array of unknown size is an incomplete type. It is completed, *for an identifier of that type*, by specifying the size in a later declaration ..." [emphasis added]. The types of objects `one` and `two` are completed but the type `table` itself is *never* completed. Hence, `sizeof(table)` is not permitted.

An example very similar to that submitted is shown in example 6, subclause 6.5.7 on page 74, lines 16-23.

Defect Report #011

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-008 (Rich Peterson)

Question 1

Merging of declarations for linked identifier

When more than one declaration is present in a program for an externally-linked identifier, exactly when do the declared types get formed into a composite type?

Certainly, if two declarations have file scope, then after the second, the effective type for semantic analysis is the composite type of the two declarations (subclause 6.1.2.6, page 25, lines 19-20). However, if one declaration is in an inner scope and one is in an outer scope, are their types formed into a composite type?

In particular, consider the code:

```
{
extern int i[];
{
    /* a different declaration of the same object */
    extern int i[10];
}
/* Is the following legal?
   That is, does the outer declaration
   inherit any information from the inner one? */
sizeof (i);
}
```

Similar situations can be constructed with internally linked identifiers. For instance:

```
/* File scope */
static int i[];

main()
{
/* a different declaration of the same object */
extern int i[10];
}

/* Is the following legal?
   That is, does the outer declaration
   inherit any information from the inner one? */
int j = sizeof (i);
```

Further variants of this question can be asked:

```
{
extern int i[10];
{
    /* a different declaration of the same object */
    extern int i[];

    /* Is the following legal?
       That is, does the inner declaration
       inherit any information from the outer one? */
    sizeof (i);
}
}
```

Correction

In subclause 6.1.2.6, page 25, 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.]

Question 2

Interpretation of **extern**

Consider the code:

```
/* File scope */
static int i;          /* declaration 1 */

main()
{
extern int i;          /* declaration 2 */
{
    extern int i;      /* declaration 3 */
}
}
```

A literal reading of subclause 6.1.2.2 says that declarations 1 and 2 have internal linkage, but that declaration 3 has external linkage (since declaration 1 is not visible, being hidden by declaration 2). (This combination of internal and external linkage is undefined by subclause 6.1.2.2, page 21, lines 27-28.)

Is this what is intended?

Correction

In subclause 6.1.2.2, page 21, 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.]

Question 3

Initialization of tentative definitions

If the file scope declaration

```
int i[10];
```

appears in a translation unit, subclause 6.7.2 suggests that it is implicitly initialized as if

```
int i[10] = 0;
```

appears at the end of the translation unit. However, this initializer is invalid, since subclause 6.5.7 prescribes that the initializer for any object of array type must be brace-enclosed. We believe that the intention of subclause 6.7.2 is that this declaration has an implicit initializer of

```
int i[10] = {0};
```

Is this true?

Response

Subclause 6.7.2 **External object definitions** contains the following excerpt:

> If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the

translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

This statement describes an effect and not a literal token sequence. Therefore, this example does not contain an error.

Question 4

Tentative definition of externally-linked object with incomplete type

If one writes the file-scope declaration

```
int i[];
```

then subclause 6.7.2 suggests that at the end of the translation unit the implicit declaration

```
int i[] = {0};
```

or equivalently

```
int i[1] = {0};
```

appears. This seems peculiar, since subclause 6.7.2, (page 83, lines 35-36) specifically forbids this case for internally linked identifiers.

Is this what is intended?

Correction

Add to subclause 6.7.2, page 84, 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.

Defect Report #012

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-046 (David F. Prosser)

Question 1

Bug in Standard C

I was asked a question about the validity of various expressions. Among the list there was the following:

```
void *p; &*p;
```

After doing a quick pass through the standard, I found nothing that disallowed such. Moreover, back in September 1987's meeting (I didn't just recall the date ... it took a while to find when it occurred), I distinctly remember a Committee discussion that involved the validity of the expression above. It was as a result of this discussion and vote that the draft was changed to allow the above.

Anyway, I wrote back that the expression was valid. This was eventually followed by a letter from Dennis [Ritchie] pointing out the mistake I made. As it turns out, the definition of lvalue makes at least the unary `&` part of the above a constraint violation. (As Bill [Plauger] would say, "I know what the standard was *supposed* to specify.")

This would be just another, "Oops, well I guess I can live with it" surprise in the standard, except that it turns out that unary `&` of a `void` type is useful! What it provides is a construction that gives C a notion of an address symbol. You are familiar with the symbols that are created by the UNIX linker: `etext`, `edata`, and `end`, which designate special addresses within the `a.out`'s address space. Of these, the last is most useful (it gives the beginning of the dynamically allocated data space). However, the *type* for these symbols was always pretty fuzzy. But, consider a declaration of `end` as

```
extern void end;
```

What this gives is a name that only has an address — exactly what these symbols do, and nothing more. They can only be used in C as the operand of unary `&`, and the address must be converted to something else (say, `char *`) even to do address calculation, making the special nature of the symbol clearly evident.

What I'd like is a vote of the interpretations group that notes that the intent of the Committee was that "`void *p; &*p;`" was supposed to be valid, even though a conforming implementation must diagnose the expression. This means that I can continue to suggest the "`extern void`" approach to address symbols in C.

P.S.: The following is my reply to Dennis's mail that pointed out the error with my original interpretation. The indented parts are from Dennis's mail.

> I don't agree with Dave P's answer about "`void *vp; &*vp;`." There is not a constraint on `*`, but the subclause 6.3.3.2 semantics say, "... if it [the operand of `*`] points to an object, the result is an lvalue designating the object." Does `vp` point to an object? An object is "a region of data storage ... the contents of which can represent values" (clause 3). Dicey at best.

I took some time looking into my records of the Committee's thoughts on this very issue. Back in '87, based on a proposal by Plauger, the Committee voted 27 to 3 that "`* (void *)`" was not to be an error. This was when the unary `*` constraint was simplified to the current form. Since `void` is a special instance of an incomplete object type, it can be thought of as pointing at an object whose size we do not know, but I agree that the argument is strained. I would still recommend that the compiler not produce a hard error in this situation.

> Moreover, the operand of `&` must be an lvalue, and `*vp` is certainly not an lvalue (subclause 6.2.2.1): "An *lvalue* is an expression (with an object type or an incomplete type other than `void`) ..."

Oops. In this case, I completely agree with Dennis: the standard does say that unary `&` should not be applied to an expression with type `void` since such cannot be an lvalue. Unfortunately, this means that the standard is "broken," at least according to the Committee's decisions. One of the major arguments presented as part of the September 1987 meeting for allowing "`* (void *)`" was that it could then be immediately used as the operand of unary `&`!

Therefore, I can state that back in 1987, the Committee's intent was that the examples you gave were valid Standard C, but that the standard as written does not allow the second half of the construction for **void**! Nevertheless, I'd still suggest allowing the code to successfully compile, with at most a warning.

Response

The relevant citations are subclause 6.3.3.2 (page 43, lines 36-38):

> The operand of the unary **&** operator shall be either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier.

and the one supplied by you from subclause 6.2.2.1 (page 36, lines 3-4):

> An *lvalue* is an expression (with an object type or an incomplete type other than **void**) that designates an object.

Given the following declaration:

```
void *p;
```

the expression **&*p** is invalid. This is because ***p** is of type **void** and so is not an lvalue, as discussed in the quote from subclause 6.2.2.1 above. Therefore, as discussed in the quote from subclause 6.3.3.2 above, the operand of the **&** operator in the expression **&*p** is invalid because it is neither a function designator nor an lvalue.

This is a constraint violation and the translator must issue a diagnostic message.

The desired effect can be obtained by using the declaration

```
extern const void end;
```

(where **end** denotes an object of unknown size) since **const void** type is not **void** type and thus **&end** does not violate the constraint in subclause 6.3.3.2.

Footnote 6 (page 6), which is not part of the standard, provides a suggestion for implementors who may wish to assign a meaning to the above expression. It says "(An implementation) may also successfully translate an invalid program." Therefore, as long as a diagnostic message is issued, a translator may assign a meaning to the expression **&*p** discussed above. Conforming programs shall not use this expression, however.

Defect Report #013

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-047 (Sam Kendall)

Question 1

Compatible and composite function types

A fix to both problems Mr. Jones raises in X3J11 Document Number 90-006 is: In subclause 6.5.4.3 on page 68, lines 23-25, change the two occurrences of "its type for these comparisons" to "its type for compatibility comparisons, and for determining a composite type." This change makes the sentences pretty awkward, but I think they remain readable.

This change makes all three of Mr. Jones's declarations compatible:

```
int f(int a[4]);
int f(int a[5]);
int f(int *a);
```

This should be the case; it is consistent with the base document's idea of "rewriting" the parameter type from array to pointer.

Correction

In subclause 6.5.4.3, page 68, 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.)

Question 2

"Compatible" not defined for recursive types

The term "compatible" is not completely defined. Consider the following two file-scope declarations in *separate* translation units:

```
extern struct a { struct a *p; } x;
struct a { struct a *p; } x;
```

Are these two declarations of `x` compatible? Obviously they should be, but subclause 6.1.2.6 does not say so.

Because subclause 6.1.2.6 does not say how to terminate the recursion in testing for compatibility of two recursive types, either interpretation is possible. In other words, it is consistent with the rules in subclause 6.1.2.6 to decide that the two declarations are compatible; but it is also consistent to decide that they are incompatible.

We can solve the problem roughly as follows: append the following draft sentence to the first paragraph of subclause 6.1.2.6 (page 25, line 8):

> If two types declared in separate translation units admit the possibility of being either compatible or incompatible, the two types shall be compatible.* [Footnote *: This case occurs with recursive types.]

This sentence is not satisfactory; perhaps another Committee member can state this rule better.

Response

We agree that the C Standard can be read in a way that it "loops." Our intent, and we feel the only reasonable solution, is that the recursion stops and the two types are regarded as compatible.

Question 3

Composite type of `enum` vs. integer not defined

There is one case where two types are compatible, but their composite type is not defined. To fix this problem, in subclause 6.1.2.6 insert after page 25, line 17:

— If one type is an enumeration and the other is an integer type, the composite type is the enumeration.

There may be other cases where "compatible" is not defined. I made a cursory search and did not find any.

Response

The issue is that in

```
enum {r,w,b} x;
```

and

```
some-int-type x;
```

where ***some-int-type*** happens to be the type that by subclause 6.5.2.2, page 61, line 40 is compatible with the type of the **enum**, what is the resultant composite type?

Subclause 6.1.2.6 on page 25, lines 11-12 says "*a* type that ... satisfies the following conditions" (added emphasis on "*a*"). The composite type of two compatible types is not necessarily unique. In this case both the **enum** type and the ***some-int-type*** satisfy the definition of "composite" type. This refutes the claim that the "composite type is not defined;" the point is that the standard does not guarantee a *unique* composite type.

As an example, in the following declarations:

```
enum {r, w, b} x;
some_int_type x;
```

provided the enumeration type is compatible with the type of **some_int_type**, it is unspecified whether the composite type of **x** is the enumeration type or **some_int_type**.

Question 4

When a structure is incomplete

Reference subclause 6.5.2.3, page 62, lines 25-28:

> If a type specifier of the form
>
> ***struct-or-union identifier***
>
> occurs prior to the declaration that defines the content, the structure or union is an incomplete type.

In the following example, neither the second nor the third occurrence of **struct foo** seem adequately covered by this sentence:

```
struct foo {
     struct foo *p;
} a[sizeof (struct foo)];
```

In the second occurrence **foo** is incomplete, but since the occurrence is within "the declaration that defines the content," it cannot be said to be "prior" that declaration. In the third occurrence **foo** is complete, but again, the occurrence is within the declaration.

To fix the problem, change the phrase "prior to the declaration" to "prior to the end of the ***struct-declaration-list*** or ***enumerator-list***."

Correction

In subclause 6.5.2.3, page 62, 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

Question 5

Enumeration tag anomaly

Consider the following (bizarre) example:

```
enum strange1 {
     a = sizeof (enum strange1)     /* line [2] */
};
```

```
enum strange2 {
     b = sizeof (enum strange2 *)  /* line [5] */
};
```

The respective tags are visible on lines [2] and [5] (according to subclause 6.1.2.1, page 20, lines 39-40, but there is no rule in subclause 6.5.2.3, **Semantics** (page 62) that governs their meaning on lines [2] and [5]. Footnote 62 on page 62 seems to be written without taking this case into account.

The first declaration must be illegal. The second declaration should be illegal for simplicity.

Perhaps these declarations are already illegal, since no rule gives them a meaning. To clarify matters, I suggest in subclause 6.5.2.3 appending to page 62, line 35:

> A type specifier of the form
>
> **enum *identifier***
>
> shall not occur prior to the end of the ***enumerator-list*** that defines the content.

If this sentence is not appended, something like it should appear as a footnote.

Correction

Add to subclause 6.5.2.3, page 63, 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.

Defect Report #014

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-049 (Max K. Goff)

Question 1

X/Open Reference Number KRT3.159.1

There are conflicting descriptions of the `setjmp()` interface in ISO 9899:1990. In subclause 7.6 on page 118, line 8, it is stated that "It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage." Throughout the rest of the standard, however, it is referred to as "the `setjmp` macro"; in addition, the rationale document states that `setjmp` must be implemented as a macro. Please clarify whether `setjmp` must be implemented as a macro, or may be a function as well as a macro, or may just be a function.

Response

The standard states that `setjmp` can be either a macro or a function. It is referred to as "the `setjmp` macro" just to avoid longwindedness. The rationale document is incorrect in saying that it must be a macro.

Question 2

X/Open Reference Number KRT3.159.2

Subclause 7.9.6.2 **The `fscanf` function** states:

> If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current input directive have been read (other than leading white space, where permitted), execution of the current directive terminates with input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

How should an implementation behave when end-of-file terminates an input stream that satisfies all conversion specifications that consume input but there is a remaining specification request that consumes no input (e.g. `%n`)? Should the non-input-consuming directive be evaluated or terminated with an input failure as described above?

Correction

Add to subclause 7.9.6.2, page 137, 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, page 138, 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`.

Defect Report #015

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-051 (Craig Blitz)

Question 1

This question concerns the promoted type of plain `int` bit-fields with length equal to the size of an object of type `int`. I am interested in implementations that have chosen not to regard the high-order bit as a sign bit.

The question is: What is the promoted type of such an object?

Subclause 6.5.2.1 states:

> A bit-field shall have a type that is ... `int`, `unsigned int`, or `signed int`.

The intent of this, I believe, is that the type of a plain `int` bit-field is `int`.

Subclause 6.2.1.1 states:

> A `char`, a `short int`, or an `int` bit-field, or their signed or unsigned varieties, ... may be used in an expression wherever an `int` or `unsigned int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise it is converted to an `unsigned int`...
>
> The integral promotions preserve value including sign.

Tracing this through, then, the type of any promoted plain `int` bit-field is `int`, since `int` can hold all the values of the original type, which is `int`. However, not all values of the bit-field, which may be regarded as non-negative, can be represented by an `int`. By value-preserving promotion rules, I would expect the type of the promoted bit-field to be `unsigned int`.

Can you clarify this?

Response

As described in subclause 6.2.1.1, bit-fields that are being treated as unsigned will promote according to the same rules as other unsigned types: if the width is less than `int`, and `int` can hold all the values, then the promotion is to `int`. Otherwise, promotion is to `unsigned int`.

Defect Report #016

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-052 (Sam Kendall)

Question 1

I can find no prohibition of the following translation unit:

```
struct foo x;
struct foo { int i; };
```

What I was looking for, but didn't find, was a statement that an implicitly initialized declaration of an object with static storage duration must have object type. Is this translation unit legal?

Response

The translation unit cited is valid. It falls into the same category of construct as

```
int array[];
int array[17];
```

Objects may be declared without knowing their size. However, the standard is clear in what cases such an object may or may not be used, prior to the actual definition of the object.

Question 2

This one is relevant only for hardware on which either null pointer or floating point zero is *not* represented as all zero bits.

Consider this sentence in subclause 6.5.7 (starting on page 71, line 41):

> 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.

This implies that you cannot implicitly initialize a union object that could contain overlapping members with different representations for zero/null pointer. For example, given this translation unit:

```
union { char *p; int i; } x;
```

If the null pointer is represented as, say, `0x80000000`, then there is no way to implicitly initialize this object. Either the `p` member contains the null pointer, or the `i` member contains 0, but not both. So the behavior of this translation unit is undefined.

This is a bad state of affairs. I assume it was not the Committee's intention to prohibit a large class of implicitly initialized unions; this would render a great deal of existing code nonconforming.

The right thing — although I can find no support for this idea in the draft — is to implicitly initialize only the first member of a union, by analogy with explicit initialization. Here is a proposed new sentence; perhaps it can be saved for the next time we make a C standard. (This sentence also tries to get around the difficulty of the old "as if ... assigned" language in dealing with `const` items; Dave Prosser tipped me off there.)

> If an object that has static storage duration is not initialized explicitly, it is initialized implicitly according to these rules:
>
> 1) if it is a scalar with pointer type, it is initialized implicitly to a null pointer constant;
>
> 2) if it is a scalar with non-pointer type, it is initialized implicitly to zero;
>
> 3) if it is an aggregate, every member is initialized (recursively) according to these rules;
>
> 4) if it is a union, the first member is initialized (recursively) according to these rules.

Correction

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.

Defect Report #017

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/90-056 (Derek M. Jones)

Question 1

New-line in preprocessor directives

Subclause 5.1.1.2, page 5, line 37 says: "Preprocessing directives are executed and macro invocations are expanded."

Subclause 6.8, page 86, lines 2-5 say: "A preprocessing directive ... and is ended by the next new-line character."

Subclause 6.8.3, page 89, lines 38-39 say: "Within the sequence of preprocessing tokens ... new-line is considered a normal white-space character."

These three statements are not sufficient to categorize the following:

```
#define f(a,b) a+b
#if f(1,
        2)
...
```

It should be defined whether the preprocessing directive rule or macro expansion wins, i.e. is this code fragment legal or illegal?

In translation phase 4 "preprocessing directives are executed and macro invocations expanded."

Now do macro invocations get done first, followed by preprocessor directives? Does the macro expander need to know that what it is expanding forms a preprocessing directive?

Subclause 6.8, page 86, lines 2-5 suggest that the preprocessor directive is examined to look for the new-line character. But how is it examined? Obviously phases 1-3 happen during this examination. So why shouldn't part of phase 4?

Correction

Add to subclause 6.8, page 86, 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.

Question 2

Behavior if no function called `main` exists

According to subclause 5.1.2.2.1, page 6, it is implicitly undefined behavior if the executable does not contain a function called `main`.

It ought to be explicitly undefined if no function called `main` exists in the executable.

Response

You are correct that it is implicitly undefined behavior if the executable does not contain a function called `main`. This was a conscious decision of the Committee.

There are many places in the C Standard that leave behavior implicitly undefined. The Committee chose as a style for the C Standard not to enumerate these places as explicitly undefined behavior. Rather, subclause 3.16, page 3, lines 12-16 explicitly allow for implicitly undefined behavior and explicitly give implicitly undefined behavior equal status with other forms of undefined behavior.

Correction

Add to subclause G.2, page 200:

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

Question 3

Precedence of behaviors

Refer to subclause 6.1.2.6, page 25, lines 9-10 and subclause 6.5, page 57, lines 20-21. The constructs covered by these sentences overlap. The latter is a constraint while the former is undefined behavior. In the overlapping case who wins?

Correction

In subclause 5.1.1.3, page 6, 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, page 6:

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.

Question 4

Mapping of escape sequences

Refer to subclause 6.1.3.4, page 29, line 12 and line 16. Are these values the values in the source or execution character set?

When subclause 6.1.3.4, page 29, line 24 says: "The value of an ...," is this "value" the value in the source character set of the escape sequence or the value of the mapped escape sequence? I would have said that the "value" is the value in the execution environment since in the source environment `\x123` is part of a token.

It might be argued that characters in the source character set do not have values and thus no misinterpretation of "value" can occur. Subclause 5.2.1, page 10, lines 25-26 refer to the value of a character in the source basic character set.

Response

The values of octal or hexadecimal escape sequences are well defined and not mapped. For instance, the value of the constant `'\x12'` is always 18, while the value of the constant `'\34'` is always 28.

The mapping described in subclause 6.1.3.4 on page 28, lines 35-39 only applies to members of the source character set, of which octal and hexadecimal escape sequences clearly are not members.

Question 5

Example of value of character constants

Refer to subclause 6.1.3.4, page 29, lines 24-25 and page 30, lines 9-10. Both of these statements cannot be true.

1) If the constraint is violated, end of story. There is no implementation-defined value.
2) The implementation-defined behavior may be referring to the mapping of the escape sequence to the basic character set, in which case subclause 6.1.3.4, page 29, lines 24-25 should be changed to mention that it will violate a constraint if the mapped value is outside the range of representable values for the type `unsigned char`.

Response

The values of octal or hexadecimal escape sequences are well defined and not mapped. For instance, the value of the constant `'\x123'` has the value 291.

The mapping described in subclause 6.1.3.4 on page 28, lines 35-39 applies only to members of the source character set, of which octal and hexadecimal escape sequences clearly are not members.

The constraint in subclause 6.1.3.4 on page 29, lines 24-25 will be violated only if the implementation uses characters of eight bits.

The text of the example in subclause 6.1.3.4 on page 30, lines 8-10 is slightly opaque, but the parenthesized comment is meant to be subject to the words "Even if eight bits are used ..." The value is implementation-defined only in that the implementation specifies how many bits are used for characters and whether type **char** is signed or not.

This example could be worded a little more clearly to indicate what is implementation-defined about the constant, and that it "violates the above constraint" only if eight bits are used for objects that have type **char**, but we believe that this interpretation is consistent with the intent of the Committee, and that a reasonable reading of the standard supports this interpretation.

Question 6

register on aggregates

```
void f(void)
{
register union{int i;} v;

&v;       /* Constraint error */
&(v.i);   /* Constraint error or undefined? */
}
```

In subclause 6.3.3.2 on page 43, lines 37-38 in a constraint clause, it says "... and is not declared with the **register** storage-class specifier." But in the above, the field **i** is not declared with the **register** storage-class specifier.

Footnote 58, on page 58, states that "... the address of any part of an object declared with storage-class specifier **register** may not be computed ..." Although the reference to this footnote is in a constraints clause I think that it is still classed as undefined behavior.

Various people have tried to find clauses in the standard that tie the storage class of an aggregate to its members. I would not use the standard to show this point. Rather I would use simple logic to show that if an object has a given storage class then any of its constituent parts must have the same storage class. Also the use of storage classes on members is syntactically illegal.

The question is not whether such a construction is legal but the status of its illegality. Is it a constraint error or undefined behavior?

It might be argued that although **register** does not appear on the field **i**, its presence is still felt. I would point out that the standard does go to some pains to state that in the case of **const union{...}** the **const** does apply to the fields. The fact that there is no such wording for **register** implies that **register** does not follow the **const** rule.

Correction

***Add to subclause 6.5.1, page 58* (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.

Question 7

Scope and uniqueness of **size_t**

Subclause 6.3.3.4 on page 45, lines 1-2 says: "... and its type (...) is **size_t** defined in the **<stddef.h>** header." This line could be read as either of the following:

1) "... and its type is **size_t** which happens to be defined in **<stddef.h>**."

2) "... and its type is the **size_t** defined in **<stddef.h>**."

(It was probably intended as a helpful piece of information only.) So what does the compiler do?

In (1) the compiler has to define a **size_t** in some outer scope. This definition does not make **size_t** visible, but gives a type to the return value of **sizeof**. Now if the programmer defines a typedef making **size_t** synonymous with **float** (say) then the compiler now has to use this new type. This interpretation does not require the programmer to include **<stddef.h>** in order to use **sizeof**.

In (2) the compiler picks up the type **size_t** from **<stddef.h>** (assuming that the user included this header). Should the compiler give a diagnostic if this header was not included and **sizeof** was used? A subsequent typedef for **size_t** does not affect the type of the result of **sizeof**.

These problems do not arise with **int**, et al. because they are keywords. Thus "**typedef float int**" would give a syntax error and need not be considered semantically.

According to subclause 6.3.3.4, page 45, **sizeof** has type **size_t**. What happens if the type of **size_t** does not match what the compiler thinks is the type of **sizeof**?

Response

The relevant citations are subclause 6.3.3.4

> The value of the result is implementation-defined, and its type (an unsigned integral type) is **size_t** defined in the **<stddef.h>** header.

and subclause 7.1.6

> The types are ...
>
> **size_t**
>
> which is the unsigned integral type of the result of the **sizeof** operator; ...

These sections, both separately and together, define the relationship between the result type of **sizeof** and the type **size_t** defined in **<stddef.h>**. The result type of **sizeof** and the type **size_t** defined in **<stddef.h>** are an unsigned integral type, and **size_t** defined in **<stddef.h>** is identical to the result type of **sizeof**. To restate, in a conforming implementation, the result type of **sizeof** will be the same as the type of **size_t** defined in **<stddef.h>**.

Since these two types are the same, there need be no mechanism for a compiler to discover the type of **size_t** defined in **<stddef.h>**. A compiler's private knowledge of the result type of **sizeof** is as good as **<stddef.h>**'s private knowledge of the type of **size_t**.

Note that the result of **sizeof** has the same type as not just any **size_t**, but the **size_t** defined in **<stddef.h>**.

Question 8

Compatibility of pointer to **void** with storage class

Refer to subclause 6.3.9, page 49, lines 24-25. Do these lines make the following legal?

```
register void *p;
char *q;
if (p==q)  /* legal */
...
```

The wording on line 25, "... version of **void**; or" does not talk about the "**void** type." This sentence could be taken as simply referring to the occurrence of a qualified or unqualified occurrence of **void**.

Should the wording on line 25 be changed to "... version of the type **void**; or" and thus cause the storage class to be ignored, or does the above example fall outside the scope of the constraint?

Response

The relevant citation is subclause 6.3.9:

> — 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**; or

The Committee believes that the current wording of the standard is clear, and it is not changed in meaning by changing "version of **void**" in the quoted section to "version of the **void** type."

The standard uses the word "**void**" in two contexts: the keyword itself and the type that the keyword names. The context that the word is used in adequately distinguishes between the two. In the section quoted, which discusses type compatibility, a misreading of "**void**" as meaning the keyword quickly results in nonsense.

As to the qualification discussed in the quoted passage, it is type qualification, defined in subclause 6.5.3. The standard only uses the words "qualified" and "unqualified" when discussing type qualification and never uses them when discussing storage classes. Thus, storage classes have no place in the discussion of the quoted passage.

Question 9

Syntax of assignment expression

In subclause 6.3.16.1 on page 53, lines 31-32 there is a typo: "... of the assignment expression ..." should be "... of the unary expression ..."

In subclause 6.3.16 on page 53, lines 3-5 we have

assignment-expression:

```
    ...
    unary-expression  assignment-operator  assignment-expression
```

Now the string "***assignment-expression***" occurs twice.

The use of "assignment expression" in subclause 6.3.16 on page 53, line 12 refers to the first occurrence (the one to the left of the colon).

We suggest changing the use of "assignment expression" in subclause 6.3.16.1 on page 53, line 32 in order to prevent confusion. The fact that any qualifier is kept actually makes more sense, since this qualifier has to take part in any constraint checking.

Correction

Add to subclause 6.3.16.1, page 54, 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.

Question 10

When is `sizeof` needed?

Refer to subclause 6.5.2.3, page 62, lines 28-29. When is the size of an incomplete structure needed? An interpreter may not need the size until run time, while some strictly typed memory architecture may not even allow pointers to structures of unknown size.

In subclause 6.5.2.3, Footnote 63 starts off as an example. The last sentence contains a "shall." Does a violation of this "shall" constitute undefined behavior?

Even though an interpreter may not need the size of a structure until run time its compiler still has to do some checking, i.e. an unexecuted statement may contain `sizeof` an incomplete type; even though the statement is unexecuted the constraint still has to be detected.

Response

Whether the language processor is an interpreter or a true compiler does not affect the language rules about when the size of an object is needed. Both a compiler and an interpreter must act as if the translation phases in subclause 5.1.1.2 were followed. This is a requirement that an implementation act as if the entire program is translated before the program's execution.

The "shall" in Footnote 63 in subclause 6.5.2.3 carries no special meaning: this footnote, like all other footnotes in the standard, is provided to emphasize the consequences of the rules in the standard. The footnote is not part of the standard.

The Committee believes that a careful reading of the standard shows all of the places that the size of an object is needed, and that the translation phases prevent those requirements from being relaxed by an implementation.

Question 11

Clarification of incomplete `struct` declaration

Referring to subclause 6.5.2.3, page 62:

```
struct t;
struct t; /* Is this undefined? */
```

People seem to think that the above is undefined.

The problem arises because no rules exist for compatibility of incomplete structures or unions.

Response

The proposed example is valid. Nothing in the standard prohibits it.

The relevant citation is subclause 6.5.2.3 Semantics, paragraph 2:

> A declaration of the form
>
> ***struct-or-union identifier ;***
>
> specifies a structure or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

Question 12

Ambiguous parsing of typedefs in prototypes

On page 67 in subclause 6.5.4.3, an ambiguity needs resolving in the parsing of the following:

a) `int x(T (U));`

b) `int x(T (U (int a, char b)));`

In (a) `U` is the type of the parameter to a function returning type `T`. From subclause 6.5.4.3, page 68, line 2:

> 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.

Thus in the case of (b):

1) `U` could be a redundantly parenthesized name of a function which takes a ***parameter-type-list*** and returns type `T`, or

2) `U` could be the type returned by a function which takes a ***parameter-type-list***, which in turn is the single parameter of a function returning type `T`.

Response

See Defect Report #009, Question 1 for a clarifying correction in this area.

Question 13

Compatibility of functions with `register` on parameters

Reference subclause 6.5.4.3, page 67.

```
f1(int);
f1(register int a) /* Is this function compatible with the above? */
{
}
```

Subclause 6.5.4.3, page 68, lines 5-7 were presumably intended to make sure that the `register` storage class got kept in the case of a definition so that the appropriate constraints applied, i.e., it is not allowed to take its address, etc. But the further implication of the wording is that the occurrence of `register` lingers on for other uses — but there are no other uses.

Suggest a clarification on this point.

Response

The function is compatible. Storage class is not part of the type.

The relevant citation, as given, is subclause 6.5.4.3, page 68, lines 5-7, but it does not imply any "other uses."

Question 14

`const void` type as a parameter

Refer to subclause 6.5.4.3, page 67, line 37. `f(const void)` should be explicitly undefined; also `f(register void)`, `f(volatile void)`, and combinations thereof.

Correction

Add to subclause G.2, page 201:

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

Question 15

Ordering of conversion of arrays to pointers

In subclause 6.5.4.3 on page 68, line 22 there is a sentence in parentheses. Does the sentence refer to the whole paragraph or just the preceding sentence?

```
int f(int a[4]);
int f(int a[5]);
int f(int *a);
```

1) It refers to the whole paragraph. This makes all of the above three declarations compatible.

2) It does not refer to the whole paragraph. This makes all three declarations incompatible.

Response

Regarding page 68, line 22: There are *two* sentences in parentheses. They apply to the entire paragraph. The declarations are all compatible. (See Defect Report #013, Question 1 for a clarifying correction in this area.)

Question 16

Pointer to multidimensional array

Given the declaration:

```
char a[3][4], (*p)[4]=a[1];
```

Does the behavior become undefined when:

1) `p` no longer points within the slice of the array, or

2) `p` no longer points within the object `a`?

This case should be explicitly stated.

Arguments for/against:

The standard refers to a pointed-to object. There does not appear to be any concept of a slice of an array being an independent object.

Response

For an array of arrays, the permitted pointer arithmetic in subclause 6.3.6, page 47, lines 12-40 is to be understood by interpreting the use of the word "object" as denoting the specific object determined directly by the pointer's type and value, *not* other objects related to that one by contiguity. Therefore, if an expression exceeds these permissions, the behavior is undefined. For example, the following code has undefined behavior:

```
int a[4][5];

a[1][7] = 0;    /* undefined */
```

Some conforming implementations may choose to diagnose an "array bounds violation," while others may choose to interpret such attempted accesses successfully with the "obvious" extended semantics.

Correction

Add to subclause G.2, page 201:

— 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).

Question 17

Initialization of unions with unnamed members

Subclause 6.5.7 on page 71, line 39 says: "All unnamed structure or union members are ignored ..." On page 72, lines 22-23, it says: "... for the first member of the union." Subclause 6.5.2.1, page 60, line 40 and Footnote 60 say that a field with no declarator is a member.

```
union {
        int   :3;
        float f;} u = {3.4};
```

Should page 72 be changed to refer to the first named member or is the initialization of a union whose first member is unnamed illegal?

It has been suggested that the situation described above is implicitly undefined.

I think that it is a straightforward ambiguity that needs resolution one way or the other.

Correction

In subclause 6.5.7, page 71, 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.

In subclause 6.5.7, page 72, 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.

Question 18

Compatibility of functions with `void` and no prototype

```
f2(void);
f2(); /* Is this function compatible with the one above? */
```

Now subclause 6.5.4.3, page 68, line 1 says that the first declaration of `f2` specifies that the function has no parameters.

No rules are given in the subsequent paragraphs to say that a function declaration with a parameter type list, with no parameters, is compatible with a function declaration with an empty parameter list.

If we treat the `void` as a single parameter then page 68, lines 14-18 would make the above two functions incompatible. `void` is not compatible with any default promotions. subclause 6.5.4.3, page 68, lines 18-22 cover the case for declaration and definition.

Thus I think that in the above example the behavior is implicitly undefined.

Response

Subclause 6.5.4.3, page 67, line 37 and page 68, line 1 state, "The special case of `void` as the only item in the list specifies that the function has no parameters." Therefore, in the case of `f2(void);` there are *no* parameters just as there are none for `f2();`. Since both functions have the same return type, these declarations *are* compatible.

Question 19

Order of evaluation of macros

Refer to subclause 6.8.3, page 89. In:

```
#define f(a) a*g
#define g(a) f(a)
f(2)(9)
```

it should be defined whether this results in:

1) `2*f(9)`

or

2) `2*9*g`

X3J11 previously said, "The behavior in this case could have been specified, but the Committee has decided more than once not to do so. [They] do not wish to promote this sort of macro replacement usage."

I interpret this as saying, in other words, "If we don't define the behavior nobody will use it." Does anybody think this position is unusual?

People seem to agree that the behavior is ambiguous in this case. Should we specify this case as undefined behavior?

Response

If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token, it is unspecified whether this macro name may be subsequently replaced. If the behavior of the program depends upon this unspecified behavior, then the behavior is undefined.

For example, given the definitions:

```
#define f(a) a*g
#define g(a) f(a)
```

the invocation:

```
f(2)(9)
```

results in undefined behavior. Among the possible behaviors are the generation of the preprocessing tokens:

```
2*f(9)
```

and

```
2*9*g
```

Correction

Add to subclause G.2, page 202:

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

Question 20

Scope of macro parameters

Refer to subclause 6.8.3 on page 89, line 16; the scope of macro parameters should be defined in the section on scope.

The idea is to enable all references to the scope of names to be under one heading. This is not really a significant issue.

Response

Subclause 6.1.2 on page 20, line 5, states "Macro names and macro parameters are not considered further here." This approach was intentionally adopted to avoid explicitly having to mention exceptions of using identifiers, for example in the sections on scope, linkage, name spaces, and storage durations, none of which applies to macros. The proposed change does *not* clarify the standard and may even obscure it.

Question 21

Self references in translation phase 4

The following queries arise because of the imprecise way in which phase 4 interacts with itself. While processing a token within phase 4 it is sometime necessary to get the following tokens from the input, i.e. reading the arguments to a function-like macro. But when getting these tokens it is not clear how many phases operate on them:

1) Do the following tokens only get processed by phases 1-3?

2) Do the following tokens get processed by phases 1-4?

When an identifier declared as a function-like macro is encountered, how hard should an implementation try to locate the opening/closing parentheses?

In:

```
#define lparen (
#define f_m(a) a
f_m lparen "abc" )
```

should the object-like macro be expanded while searching for an opening parenthesis? Or does the lack of a readily available left parenthesis indicate that the macro should not be expanded?

Subclause 6.8.3, on page 89, lines 34-35 says "... followed by a `(` as the next preprocessing token ..." This sentence does not help because in translation phase 4 all tokens are preprocessing tokens. They don't get converted to "real" tokens until phase 7. Thus it cannot be argued that `lparen` is not correct in this situation, because its result is a preprocessing token.

In:

```
#define i(x) 3
#define a i(yz
#define b )
a b )      /* goes to 3) or 3 */
```

does `b` get expanded to complete the call `i(yz,` or does the parenthesis to its right get used?

Response

Concerning the first example:

```
#define lparen (
#define f_m(a) a
f_m lparen "abc" )
```

According to subclause 5.1.1.2 **Translation phases**, page 5, lines 25-39, the translation phases 1-3 do not cause macros to be expanded. Phase 4 does expand. To apply subclause 6.8.3 **Macro replacement** page 89, lines 34-35 to the example: Since `lparen` is not `(` in "`f_m lparen "abc" )`," this construct is not recognized as a function-like macro invocation. Therefore the example expands to

```
f_m("abc")
```

The same principle applies to the second example:

```
#define i(x) 3
#define a i(yz
#define b )
a b )        /* expands via the following stages: */

i(yz b )     /* ) delimits the argument list before b is expanded */

i(yz ) )

3
```

This is how we interpret subclause 6.8.3, page 89, lines 36-38: The sequence of preprocessing tokens is terminated by the right-parenthesis preprocessing token.

Question 22

Gluing during rescan

Reference: subclause 6.8.3.3, page 90. Does the rescan of a macro invocation also perform gluing?

```
#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[2] = join(x, y);
```

Is the above legal? Does `join` expand to "`xy`" or "`x ## y`"?

It all depends on the wording in subclause 6.8.3.3 on page 90, lines 39-40. Does the wording "... before the replacement list is reexamined ..." mean before being reexamined for the first time only, or before being reexamined on every rescan?

This rather perverse macro expansion is only made possible because the constraints on the use of # refer to function-like macros only. If this constraint were extended to cover object-like macros the whole question goes away.

Dave Prosser says that the intent was to produce "`x ## y`". My reading is that the result should be "`xy`". I cannot see any rule that says a created ## should not be processed appropriately. The standard does say in subclause 6.8.3.3, page 90, line 40 "... each instance of a ## ..."

The reason I ask if the above is legal is that the order of evaluation of # and ## is not defined. Thus if # is performed first the result is very different than if ## goes first.

Correction

Add to subclause 6.8.3.3, page 90:

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.

Question 23

How long does blue paint persist?

Consider the following code:

```
#define a(x) b
#define b(x) x

  a(a)(a)(a)
```

The macro replacement for `a(a)` results in `b`.

First replacement buffer: `b`

Remaining tokens: `(a)(a)`

Inside the first replacement buffer, no further nested replacements will recognize the macro name "`a`." The name "`a`" is painted blue.

The first replacement buffer is rescanned not by itself, but along with the rest of the source program's tokens. "`b(a)`" also causes macro replacement and becomes "`a`."

Second replacement buffer: `a`

Remaining tokens: `(a)`

The second replacement buffer is rescanned not by itself, but along with the rest of the source program's tokens.

The "`a`" in the second replacement buffer did not come from the first replacement buffer. It came from three of the remaining tokens which were in the source file following the first replacement buffer. Is this "`a`" part of a nested replacement? Is it still painted blue?

Note that there are many "paths" that can be taken for a possible macro name to travel from a preprocessing token (outside the replacement buffer) to one that is inside the replacement buffer. When do they stop getting painted blue? If either too early or too late, they cause very surprising results.

Given the amount of discussion involving macro expansion that uses the concept of "blue paint," why doesn't the standard tell the reader about this idea?

Everybody seems to agree that the above is undefined. Does anybody have a set of words to make this and other cases explicitly undefined?

Response

The reference is to subclause 6.8.3.4, page 91.

```
#define a(x) b
#define b(x) x
a(a)(a)(a)      /* may expand as follows: */

     b(a)(a)
a'(a)      or   a(a)
a(a)       or   b

/* a' indicates the symbol a marked for non-replacement */
```

The Committee addressed this issue explicitly in previous deliberations and decided to say nothing about the situation, understanding that behavior in such cases would be undefined.

The result, as with other examples, is intentionally left undefined.

Question 24

Improve English

Just a tidy up. Change subclause 7.1.2, page 96, line 33 from "if the identifier" to "if an identifier."

Correction

In subclause 7.1.2, page 96, 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,

Question 25

"Must" in footnotes

This change is not essential since footnotes have no status. But this change would cut down the number of occurrences of "shall" synonyms used where "shall" itself could have be used.

Response

The standard is clear enough as is.

Question 26

Implicit initialization of unions with unnamed members

Are unnamed union members required to be initialized?

Response

See Defect Report #017, Question 17 for a clarifying correction in this area.

Question 27

`g` conversions

Subclause 7.9.6.1 says on page 132, lines 42: "For `g` and `G` conversions, trailing zeros will *not* be removed ...," whereas on page 133, lines 37-38 it says: "Trailing zeros are removed ..."

It has been suggested that the italics on page 132, lines 42 gives this rule precedence. I don't mind which rule wins as long as the text says so. Do we add text to describe the italics rule or change the conflicting lines?

Response

In the collision between the description of the # flag and the `g` and `G` conversion specifiers to `fprintf`, which takes precedence?

The # flag takes precedence. Subclause 7.9.6.1, page 132, line 1 says, "Zero or more *flags* (in any order) ... modify the meaning of the conversion specification."

Question 28

Ordering of conditions on return

In subclause 7.9.9.1, subclause 7.9.9.3, and subclause 7.9.9.4, the words are "returns ... and stores an implementation-defined positive value in `errno`." This is a strange order of operations — shouldn't the wording be reversed?

Response

No. In subclause 7.9.9.1, subclause 7.9.9.3, and subclause 7.9.9.4, the words "returns ... and stores an implementation-defined positive value in `errno`" do not imply any temporal ordering. There are implementations that may perform these operations in either order and they still meet the standard.

Question 29

Conversion failure and longest matches

Consider `1.2e+4` with field width of 5. Is it input item `1.2e+` that gives a conversion failure? What is the ordering between building input items and converting them? Do they run in parallel, or sequential?

Refer to subclause 7.9.6.2 **The `fscanf` function**, page 135, lines 31-33 concerning the longest matching sequence, and subclause 7.9.6.2, page 137, lines 15-16 concerning a conflicting input character.

For `1.2e-x`, is `1.2` or `1.2e-` read?

The above questions all come about because of page 137, line 15: "If conversion terminates ..." In this context the use of the word "conversion" could be referring to the process of turning a sequence of characters into numeric form. I believe what was intended was "If a conversion specifier terminates ..."

Response

The relevant citations are subclause 7.9.6.2, page 137, lines 15-16:

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

and subclause 7.9.6.2, page 135, lines 31-33:

> 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.

and subclause 7.9.6.2, page 135, lines 38-40:

> If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure.

The "conversion" in the first quoted passage is the process of both forming an input item and converting it as specified by the conversion specifier.

About your example: If the characters available for input are "`1.2e+4`" and input is performed using a "`%5e`," then the input item is "`1.2e+`" as defined by the second passage quoted above. That input item is not a matching sequence, but only an initial subsequence that fails to be a matching sequence in its own right. Under the rules of the third quoted passage, this is a matching failure.

Note that in this case, no characters were pushed back onto the input stream. There was no "conflicting input character" that terminated the field, and so the first quoted passage does not apply.

See the Correction made in response to Defect Report #022, Question 1, for additional clarification.

Question 30

Successful call to `ftell` or `fgetpos`

In subclause 7.9.9.2 on page 145, lines 39-40, "... a value returned by an earlier call to the `ftell` function ..." should actually read "... a value returned by an earlier successful call ..." Similarly for subclause 7.9.9.3.

Correction

In subclause 7.9.9.2, page 145, 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

In subclause 7.9.9.3, page 146, 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

Question 31

Size in bytes

References to the size of an object in other parts of the standard specify that size is measured in bytes. The following lines do not follow this convention: subclause 7.10.3.1 on page 154, lines 26-27 and subclause 7.10.3.3 on page 155, line 8.

Response

There are numerous places in the standard where "size in bytes" is used, and numerous places where "size" alone is used. The Committee does not feel that any of these places need fixing — the meaning is everywhere clear, especially since for `sizeof` in subclause 6.3.3.4 size is specifically mentioned in terms of bytes.

Question 32

char parameters to **strcmp** and **strncmp**

Refer to subclause 7.11.4, page 164. If **char** is signed then **char *** cannot be interpreted as pointing to **unsigned char**. The required cast may give undefined results. This applies to **strcmp** and **strncmp**.

Response

strcmp can compare two **char** strings, even though the representation of **char** may be signed, because subclause 7.11.4, page 164, line 7 says that the interpretation of bytes is done as if each byte were accessed as an **unsigned char**. We believe the standard is clear.

Question 33

Different length strings

Refer to subclause 7.11.4, page 164, lines 5-7. What about strings of different length?

Perhaps the fact that the terminating null character takes part in the comparison ought to be mentioned.

Response

Subclause 7.1.1 on page 96, lines 4-5 says that a string includes the terminating null character. Therefore this character takes part in the comparison. The standard is clear.

Question 34

Calls to **strtok**

In subclause 7.11.5.8 on page 167, line 36, "... first call ..." should read "... all calls ..."

I think that the current wording causes confusion. The first call is the one that takes a non-**NULL** "**s1**" parameter. However, the discussion from line 36 onwards is describing the behavior for all calls.

Response

The Committee felt that the suggested wording for the **strtok** function description is not an improvement. The existing wording is clear as written.

Question 35

When is a physical source line created?

Is the output or input to translation phase 1 a physical source line?

Response

The use of the term "physical source line" occurs only in the description of the phases of translation (subclause 5.1.1.2) and the question of whether the input or output of phase 1 consists of physical source lines does not matter.

Question 36

Qualifiers on function return type

Refer to subclause 6.6.6.4, page 80, line 24-25: "... whose return type is **void**."

The behavior of a type qualifier on a function return is explicitly undefined, according to subclause 6.5.3, page 64, lines 24-25.

This creates a loophole.

An implementation that supports type qualifiers on function return types is not required to flag the constraint given on page 80.

Response

A Constraint on subclause 6.7.1 says "The return type of a function shall be **void** or an object type other than array."

Question 37

Function result type

Refer to subclause 6.3.2.2, page 40, line 35. The result type of a function call is not defined.

Correction

In subclause 6.3.2.2, page 40, 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`.

Question 38

What is an iteration control structure or selection control structure?

An "iteration control structure," a term used in subclause 5.2.4.1 **Translation limits** on page 13, line 1, is not defined by the standard.

Is it:

1) A `for` loop header excluding its body, e.g. `for (;;)`, or

2) A `for` loop header plus its body, e.g. `for (;;) {}`?

Does it make a difference if the compound statement is a simple statement without the braces?

Correction

In subclause 5.2.4.1, page 13, 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

Question 39

Header name tokenization

There is an inconsistency between subclause 6.1.7, page 33, line 8 and the description of the creation of header name preprocessing tokens.

The "shall" on page 32, line 33 does not limit the creation of header name preprocessing tokens to within `#include` directives. It simply states that they would cause a constraint error in this context.

Subclause 6.1.7, page 33, line 8 should read `{0x3}{<1/a.h>}{1e2}`, or extra text needs to be added to subclause 6.1.7.

I have not met anybody who expects `if (a<b || c>d)` to parse as `{if} {(} {a} {<b || c>} {d} {)}`.

Correction

Add to subclause 6.1, page 18 (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.

Add to subclause 6.1.2, page 20 (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.

In subclause 6.1.7, page 32, lines 32-34, delete:

Constraint

Header name preprocessing tokens shall only appear within a `#include` preprocessing directive.

Add to subclause 6.1.7, page 32 (Semantics):

A header name preprocessing token is recognized only within a `#include` preprocessing directive.

Defect Report #018

Submission Date: 10 Dec 92

Submittor: WG14

Source: X3J11/90-066 (Yasushi Nakahara)

Question 1

It is unclear how the **fscanf** function shall behave when executing directives that include "ordinary multibyte characters," especially in the case of shift-encoded ordinary multibyte characters.

The following statements are described in subclause 7.9.6.2 **The fscanf function** of the current standard:

> A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

Assume a typical shift-encoded directive: Ä in 7-bit representation. And consider two different encoding systems, Latin Alphabet No.1 — 8859/1 and German Standard DIN 66 003. The codes are, for example,

```
Ä    in 8859/1:      SO 4/4 SI
Ä    in DIN 66 003: ESC 2/8 4/11 5/11 ESC 2/8 4/2
```

where *SO* is a Shift-Out code (0/15 = 0x0F) and *SI* corresponds to a Shift-In code (0/14). "*ESC 2/8 4/11*" is an escape sequence for the German Standard DIN 66 003, and "*ESC 2/8 4/2*" is for ISO 646 USA Version (ASCII).

Assuming that a subject sequence includes Ä, Ö, and Ü with the following 7-bit representations,

```
in 8859/1:      SO 4/4 5/6 5/12 SI
in DIN 66 003: ESC 2/8 4/11 5/11 5/12 5/13 ESC 2/8 4/2
```

does the "Ä" directive in the **fscanf** format string match the beginning part of the"ÄÖÜ" sequence?

At what position of the target sequence shall the "Ä" directive fail?

One interpretation of this is that because the current standard defined the behavior of the directive in the **fscanf** format based on the word "character" (byte), not using the term "multibyte character," the comparison shall be done on a byte-by-byte basis. One may conclude that the "Ä" directive never matches the "ÄÖÜ" sequence in this case.

Another interpretation may lead to an opposite conclusion, saying that the current standard's statements quoted above do not necessarily mean that such comparison shall be done on a byte-by-byte basis. Instead, it is read that the matching shall be done on a "multibyte character by multibyte character basis" or rather "wide character by wide character basis." Especially, a "ghost" sequence like "*ESC ...*" and *SI*/*SO* characters should not be regarded as independent ordinary multibyte characters in this case.

Which is a correct interpretation of the current standard?

These different interpretations are caused by the ambiguity of the descriptions in the current standard. Also, it should be pointed out that the major problem here is usage of the word "character." The generic word "character" and the specific word "character(=byte)" should be properly discriminated in the standard.

Response

Subclause 7.9.6.2 says, "A directive that is an ordinary multibyte character is executed by reading the next *characters* ..." [emphasis added]. Consistently throughout the standard, plain "characters" refers to one-byte characters. (See subclause 3.5 for the definition of "character.")

Defect Report #019

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-014 (Richard Wiersma)

Question 1

Background:

Subclause 7.3.1.5 states that "the **isgraph** function tests for any printing character except space." Subclause 7.3.1.7 states that "the **isprint** function tests for any printing character including space."

The third paragraph of subclause 7.3 defines the term *printing character* as "a member of an implementation-defined set of characters, each of which occupies one printing position on a display device."

Subclause 5.2.1 defines the source and execution character sets and provides a list of characters which must be contained in both sets.

Question for interpretation: Are the **isprint** and **isgraph** functions required to return a non-zero value for all of the characters defined in subclause 5.2.1?

A scenario for use of **isprint/isgraph** that depends on the interpretation is: A developer may wish to use these functions to determine whether a particular character can be displayed as itself (e.g., whether a square bracket is actually displayed as a square bracket). This could be useful for formatting output in a device-independent manner, since the application could substitute some other character for ones that do not print "correctly."

If **isprint** and **isgraph** are required to return non-zero for all characters in subclause 5.2.1, developers cannot use them for this purpose.

This problem has occurred in a real implementation. The most commonly used terminals and printers for IBM System/370 computers do not support all of the characters listed in subclause 5.2.1. For example, most IBM printers and terminals do not print the square brackets.

The SAS/C implementation of **isprint** and **isgraph** assumes that subclause 7.3 controls the behavior of these functions, and returns non-zero only for those characters that print "correctly." The Plum Hall test suite, however, assumes that **isprint** and **isgraph** return non-zero for all characters listed in subclause 5.2.1.

Response

Subclause 7.3, page 102, line 8 says that *printing character* is implementation-defined. In particular, the value (zero or non-zero) of **isprint('[')** is implementation-defined, *even in the* **"C"** *locale.*

Defect Report #020

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-006 (Bruce Lambert)

Question 1

Is a compiler which allows the Relaxed Ref/Def linkage model to be considered a conforming compiler? That is, can a compiler that compiles the following code with no errors or warnings

filea.c:

```
#include <stdio.h>
void foo(void);
int age;
int main()
{
    age = 24;
    printf("my age is %d.\n", age);
    foo();
    printf("my age is %d.\n", age);
    return 0;
}
```

fileb.c:

```
#include <stdio.h>
int age;
void foo()
{
    age = 25;
    printf("your age is %d.\n", age);
}
```

and which produces the following output

```
my age is 24
your age is 25
my age is 25
```

be called a standard-compliant compiler?

Response

Yes, a compiler that allows the Relaxed Ref/Def model can be standard conforming. (In this case, the model permits two tentative definitions for **age** in two translation units to resolve to a single definition at link time.) See subclause 6.7, page 81, lines 23-25. The code is conforming but not strictly conforming. The behavior is undefined.

Defect Report #021

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-001 (Fred Tydeman)

Question 1

What is the result of: `printf("%#.4o", 345);`? Is it `0531` or is it `00531`?

Subclause 7.9.6.1, on page 132, lines 37-38 says: "For o conversion, it increases the precision to force the first digit of the result to be a zero."

Is this a conditional or an unconditional increase in the precision if the most significant digit is not already a `0`? Which is the correct interpretation?

Correction

In subclause 7.9.6.1, page 132, 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.

Defect Report #022

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-002 (Fred Tydeman)

Question 1

What is the result of: `strtod("100ergs", &ptr);`? Is it 100.0 or is it 0.0?

Subclause 7.10.1.4 **The `strtod` function** on page 150, lines 36-38 says: "The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form." In this case, the longest initial subsequence of the expected form is `100`, so `100.0` should be the return value. Also, since the entire string is in memory, `strtod` can scan it as many times as need be to find the longest valid initial subsequence.

Subclause 7.9.6.2 **The `fscanf` function** on page 136, lines 17-18 says: "`e,f,g` Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function." Later, page 138, lines 6, 16, and 25 show that `100ergs` fails to match `%f`. Those two show that `100ergs` is invalid to `fscanf` and therefore, invalid to `strtod`. Then, subclause 7.10.1.4, page 151, lines 11-12, "If no conversion could be performed, zero is returned" indicates for an error input, 0.0 should be returned. The reason this is invalid is spelled out in the rationale document, subclause 7.9.6.2 **The `fscanf` function**, page 85: "One-character pushback is sufficient for the implementation of `fscanf`. Given the invalid field `-.x`, the characters `-.` are not pushed back." And later, "The conversions performed by `fscanf` are compatible with those performed by `strtod` and `strtol`."

So, do `strtod` and `fscanf` act alike and both accept and fail on the same inputs, by the one-character pushback scanning strategy, or do they use different scanning strategies and `strtod` accept more than `fscanf`?

Correction

In subclause 7.9.6.2, page 135, 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.

In subclause 7.9.6.2, page 137, 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, page 137:

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`.]

Defect Report #023

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-003 (Fred Tydeman)

Question 1

Assuming that 99999 is larger than `DBL_MAX_10_EXP`, what is the result of:

```
strtod("0.0e99999", &ptr);
```

Is it 0.0, `HUGE_VAL`, or undefined?

Subclause 6.1.3.1 **Floating constants** on page 26, lines 30-32 says: "The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled." In this case `0.0e99999` means 0.0 times 10 to the power 99999, or 0.0×10^{99999}, which has a scaled value of 0.0; therefore, return 0.0.

Subclause 7.10.1.4 **The `strtod` function** on page 151, lines 12-14 says: "If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`." Since the exponent (99999 in this case) is larger than `DBL_MAX_10_EXP`, the value is outside the range of representable values (overflow). Therefore, return `HUGE_VAL`.

Subclause 5.2.4.2.2 **Characteristics of floating types `<float.h>`**, pages 14-16, describes the model that defines the floating-point types. The number `0.0e99999`, as written, is not part of that model (it cannot be represented since the exponent is larger than e_{max}). From subclause 6.2.1.4 **Floating types** page 35, lines 11-13, "... if the value being converted is outside the range of values that can be represented, the behavior is undefined." Therefore, since this number, as written, has no representation, the behavior is undefined.

Response

According to our response to Defect Report #025, Question 1, the result of `strtod("0.0e99999", &ptr)` is exactly representable, i.e., it lies within the range of representable values. Therefore, by subclause 7.10.1.4, **Returns**, the value zero shall be returned in this case, and `errno` shall not be set. (This means that implementations have to test for the special case of zero when creating floating-point representations from characters.)

Note also that `strtod("0.0e-99999", &ptr)` is not a case of underflow, so `errno` shall not be set to `ERANGE` in this case either.

Defect Report #024

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-004 (Fred Tydeman)

Question 1

In subclause 7.10.1.4 **The `strtod` function** page 151, line 5: What does "`"C"` locale" mean?

a) `setlocale(LC_ALL,NULL) == "C"`

b) `setlocale(LC_NUMERIC,NULL) == "C"`

c) a) && b)

d) a) || b)

e) something else.

What does "other than the `"C"` locale" mean?

a) `setlocale(LC_ALL,NULL) != "C"`

b) `setlocale(LC_NUMERIC,NULL) != "Ct`

c) a) && b)

d) a) || b)

e) something else.

Subclause 7.4.1 **Locale control**, page 107 may help answer the questions.

Response

Subclause 7.4.1.1, page 107, lines 11-17 describe what is affected by each locale portion.

Is it the `LC_NUMERIC` locale category which affects the implementation-defined behavior of `strtod`, etc.?

Answer: Yes.

How can one guarantee that `strtod` functions are in the `"C"` locale?

Answer: Execute `setlocale(LC_NUMERIC, "C")` or execute `setlocale(LC_ALL, "C")`.

What is meant by "other than the `"C"` locale?" That is, how can one ensure that `strtod` is not in the `"C"` locale?

Answer: Successfully execute `setlocale(LC_NUMERIC, str)` or `setlocale(LC_ALL, str)` to some implementation-defined string `str` which specifies a locale that is different from the `"C"` locale. No universally portable method can be provided, because the functionality is implementation-defined.

Defect Report #025

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-005 (Fred Tydeman)

Question 1

What is meant by "representable floating-point value?" Assume double precision, unless stated otherwise.

First, some definitions based partially upon the floating-point model in subclause 5.2.4.2.2, on pages 14-16 of the C Standard:

1) +Normal Numbers: `DBL_MIN` to `DBL_MAX`, inclusive; normalized (first significand digit is non-zero), sign is +1.
2) –Normal Numbers: `-DBL_MAX` to `-DBL_MIN`, inclusive; normalized.
3) +Zero: All digits zero, sign is +1; (true zero).
4) –Zero: All digits zero, sign is –1.
5) Zero: Union of +zero and –zero.
6) +Denormals: Exponent is "minimum" (biased exponent is zero); first significand digit is zero; sign is +1. These are in range `+DBL_DeN` (inclusive) to `+DBL_MIN` (exclusive). (Let `DBL_DeN` be the symbol for the minimum positive denormal, so we can talk about it by name.)
7) –Denormals: same as +denormals, except sign, and range is `-DBL_MIN` (exclusive) to `-DBL_DeN` (inclusive).
8) +Unnormals: Biased exponent is non-zero; first significand digit is zero; sign is +1. These overlap the range of +normals and +denormals.
9) –Unnormals: Same as +unnormals, except sign; range is over –normals and –denormals.
10) +infinity: From IEEE-754.
11) –infinity: From IEEE-754.
12) Quiet NaN (Not a Number); sign does not matter; from IEEE-754.
13) Signaling NaN; sign does not matter; from IEEE-754.
14) NaN: Union of Quiet NaN and Signaling NaN.
15) Others: Reserved (VAX?) and Indefinite (CDC/Cray?) act like NaN.

On the real number line, these symbols order as:

```
[   1   )[   2   ](   3   )(  4 )[5]( 6  )[   7   )[   8   ](   9   ]
+-------+------+-------+------+-+------+-------+------+-------+
-INF -DBL_MAX -DBL_MIN -DBL_Den -0 +0 +DBL_Den +DBL_MIN +DBL_MAX +INF
```

Non-real numbers are: SNaN, QNaN, and NaN; call this region 10.

Regions 1 and 9 are overflow, 2 and 8 are normal numbers, 3 and 7 are denormal numbers (pseudo underflow), 4 and 6 are true underflow, and 5 is zero.

So, the question is: What does "representable (double-precision) floating-point value" mean:

a) Regions 2, 5 and 8 (+/– normals and zero)
b) Regions 2, 3, 5, 7, and 8 (+/– normals, denormals, and zero)
c) Regions 2 through 8 [`-DBL_MAX` ... `+DBL_MAX`]
d) Regions 1 through 9 [–INF ... +INF]
e) Regions 1 through 10 (reals and non-reals)
f) What the hardware can represent
g) Something else? What?

Some things to consider in your answer follow. The questions that follow are rhetorical and do not need answers.

Subclause 5.2.4.2.2 **Characteristics of floating types <float.h>**, page 14, lines 32-34:

> The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.

Same section, page 15, line 6:

> A normalized floating-point number *x* ... is defined by the following model: ...

That model is just normalized numbers and zero (appears to include signed zeros). It excludes denormal and unnormal numbers, infinities, and NaNs. Are signed zeros required, or just allowed?

Subclause 6.1.3.1 **Floating constants**, page 26, lines 32-35: "If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest value, chosen in an implementation-defined manner."

```
    A       B    y   C     x     D          E    z   F
-DBL_Den   0.0   +DBL_Den   +DBL_MIN   +DBL_MAX   +INF
```

The representable numbers are A, B, C, D, E, and F. The number x can be converted to B, C, or D! But what if B is zero, C is **DBL_DeN** (denormal), and D is **DBL_MIN** (normalized). Is x representable? It is not in the range **DBL_MIN ... DBL_MAX** and its inverse causes overflow; so those say not valid. On the other hand, it is in the range **DBL_DeN ... DBL_MAX** and it does not cause underflow; so those say it is valid.

What if B is zero, A is **-DBL_DeN** (denormal), and C is **+DBL_DeN** (denormal); is y representable? If so, its nearest value is zero, and the immediately adjacent values include a positive and a negative number. So a user-written positive number is allowed to end up with a negative value!

What if E is **DBL_MAX** and F is infinity (on a machine that uses infinities, IEEE-754)? Does z have a representation? If z came from 1.0/x, then z caused overflow which says invalid. But on IEEE-754 machines, it would either be **DBL_MAX** or infinity depending upon the rounding control, so it has a representation and is valid.

What is "nearest?" In linear or logarithmic sense? If the number is between 0 and **DBL_DeN**, e.g., 10^{-99999}, it is linear-nearest to zero, but log-nearest to **DBL_DeN**. If the number is between **DBL_MAX** and INF, e.g., 10^{+99999}, it is linear- and log-nearest to **DBL_MAX**. Or is everything bigger than **DBL_MAX** nearest to INF?

Subclause 6.2.1.3 **Floating and integral**, page 35, Footnote 29: "Thus, the range of portable floating values is (–1,**Utype_MAX**+1)."

Subclause 6.2.1.4 **Floating types**, page 35, lines 11-15: "When a **double** is demoted to **float** or a **long double** to **double** or **float**, if the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen in an implementation-defined manner."

Subclause 6.3 **Expressions**, page 38, lines 15-17: "If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined."

```
w = 1.0 / 0.0 ;      /* infinity in IEEE-754 */
x = 0.0 / 0.0 ;      /* NaN in IEEE-754 */
y = +0.0 ;      /* plus zero */
z = - y ; /* minus zero:  Must this be -0.0? May it be +0.0? */
```

Are the above representable?

Subclause 7.5.1 **Treatment of error conditions**, page 111, lines 11-12: "The behavior of each of these functions is defined for all representable values of its input arguments."

What about non-numbers? Are they representable? What is **sin(*NaN*)** ? If you got a NaN as input, then you can return NaN as output. But, is it a domain error? Must **errno** be set to **EDOM**? The NaN already indicates an error, so setting **errno** adds no more information. Assuming NaN is not part of Standard C

"representable," but the hardware supports it, then using NaNs is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

Subclause 7.5.1 **Treatment of error conditions**, on page 111, lines 20-27 says: "Similarly, a *range error* occurs if the result of the function cannot be represented as a `double` value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign (except for the `tan` function) as the correct value of the function; the value of the macro `ERANGE` is stored in `errno`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; whether the integer expression `errno` acquires the value of the macro `ERANGE` is implementation-defined."

What about denormal numbers? What is `sin(DBL_MIN/3.0L)`? Must this be considered underflow and therefore return zero, and maybe set `errno` to `ERANGE`? Or may it return `DBL_MIN/3.0`, a denormal number? Assuming denormals are not part of Standard C "representable," but the hardware supports it, then using them is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

What about infinity? What is `exp(INF)`? If you got an INF as input, then you can return INF as output. But, is it a range error? The output value is representable, so that says: no error. The output value is bigger than `DBL_MAX`, so that says: an error and set `errno` to `ERANGE`. Assuming infinity is not part of Standard C "representable," but the hardware supports it, then using INFs is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

What about signed zeros? What is `sin(-0.0)`? Must this return –0.0? May it return –0.0? May it return +0.0? Signed zeros appear to be required in the model in subclause 5.2.4.2.2 on page 15.

What is `sqrt(-0.0)`? IEEE-754 and IEEE-854 (floating-point standards) say this must be –0. Is –0.0 negative? Is this a domain error?

Subclause 7.9.6.1 **The `fprintf` function** on page 132, lines 32-33 says: "(It will begin with a sign only when a negative value is converted if this flag is not specified.)"

What is `fprintf(stdout, "%+.1f", -0.0);`? Must it be –0.0? May it be +0.0? Is –0.0 a negative value? The model on page 15 appears to require support for signed zeros.

What is `fprintf(stdout, "%f %f", 1.0/0.0, 0.0/0.0);`? May it be the IEEE-854 strings of `inf` or `infinity` for the infinity and `NaN` for the quiet NaN? Would `NaNQ` also be allowed for a quiet NaN? Would `NaNS` be allowed for a signaling NaN? Must the sign be printed? Signs are optional in IEEE-754 and IEEE-854. Or, must it be some decimal notation as specified by subclause 7.9.6.1, page 133, line 19? Does the locale matter?

Subclause 7.10.1.4 **The `strtod` function** on page 151, lines 2-3 says: "If the subject sequence begins with a minus sign, the value resulting from the conversion is negated."

What is `strtod("-0.0", &ptr)`? Must it be –0.0? May it be +0.0? The model on page 15 appears to require support for signed zeros. All floating-point hardware I know about support signed zeros at least at the load, store, and negate/complement instruction level.

Subclause 7.10.1.4 **The `strtod` function** on page 151, lines 12-15 say: "If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`."

If `HUGE_VAL` is +infinity, then is `strtod("1e99999", &ptr)` outside the range of representable values, and a range error? Or is it the "nearest" of `DBL_MAX` and INF?

Response

Principles for C floating-point representation:

(These principles are intended to clarify the use of some terms in the standard; they are not meant to impose additional constraints on conforming implementations.)

1) "Value" refers to the abstract (mathematical) meaning; "representation" refers to the implementation data pattern.

2) Some (not all) values have exact representations.

3) There may be multiple exact representations for the same value; all such representations shall compare equal.

4) Exact representations of different values shall compare unequal.

5) There shall be at least one exact representation for the value zero.

6) Implementations are allowed considerable latitude in the way they represent floating-point quantities; in particular, as noted in Footnote 10 on page 14, the implementation need not exactly conform to the model given in subclause 5.2.4.2.2 for "normalized floating-point numbers."

7) There may be minimum and/or maximum exactly-representable values; all values between and including such extrema are considered to "lie within the range of representable values."

8) Implementations may elect to represent "infinite" values, in which case all real numbers would lie within the range of representable values.

9) For a given value, the "nearest representable value" is that exactly-representable value within the range of representable values that is closest (mathematically, using the usual Euclidean norm) to the given value.

(Points 3 and 4 are meant to apply to representations of the same floating type, not meant for comparison between different types.)

This implies that a conforming implementation is allowed to accept a floating-point constant of any arbitrarily large or small value.

Defect Report #026

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-007 (Randall Meyers)

Question 1

Example:

```
#include <stdio.h>
int main()
{
      puts("@$(etc.)");
      return 0;
}
```

Is this a strictly conforming program?

Response

Strictly conforming programs cannot depend on unspecified or implementation-defined behavior (cf. clause 4, page 3, lines 31-32). Note that `@` and `$` are extended source characters. Source characters are translated to execution characters in an unspecified manner (cf. subclause 5.2.1). This is in the `"C"` locale. The `@` character is either a printing character or a control character, either of which is implementation-defined (subclause 7.3, page 102, lines 8-11). Therefore, the program is *not* strictly conforming.

Defect Report #027

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-008 (Randall Meyers)

Question 1

May a standard conforming implementation make characters in its character set that are not in the required source character set identifier characters? Can these additional identifier characters be used in preprocessor identifier tokens as well as post-processor identifier tokens?

Subclause G.5.2 states:

> Characters other than the underscore _, letters, and digits, that are not defined in the required source character set (such as the dollar sign $, or characters in national character sets) may appear in an identifier (subclause 6.1.2).

Response

May a standard conforming implementation make characters in its character set that are not in the required source character set identifier characters?

Answer: Yes.

Can these additional identifier characters be used in preprocessor identifier tokens as well as post-processor identifier tokens?

Answer: Yes, but the C Standard is currently ambiguous about the parsing of a definition such as:

```
#define abc$ x
```

This could either define `abc$` as `x` or `abc` as `$x`. The Correction that follows resolves the ambiguity.

Correction

Add to subclause 6.8, page 86 (*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 `THIS$AND$THAT`, rather than an object-like macro `THIS`. Whichever choice it makes, it must also issue a diagnostic.]

Defect Report #028

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-009 (Randall Meyers)

Question 1

Subclause 6.3, page 38, lines 18-27 state some very important rules governing how a strictly conforming program can access the value of an object. The basic theme of the rules is that an object's value may only be accessed through an lvalue of the appropriate type. These rules are required to permit C programs to be optimized.

The rules depend on the "declared type of the object." This seems to make the rules not apply if the object was not declared, which is the case for an object allocated using `malloc()`.

Do the rules somehow apply to dynamically allocated objects? Is a compiler free to optimize the following function:

```
void f(int *x, double *y)
{
    *x = 0;
    *y = 3.14;
    *x = *x + 2;
}
```

into the equivalent function:

```
void f(int *x, double *y)
{
    *x = 0;
    *y = 3.14;
    *x = 2;     /* *x known to be zero */
}
```

Or must an optimizer prove that pointers are not pointing at dynamically allocated storage before performing such optimizations?

Response

Case 1: unions `f(&u.i, &u.d)`

Subclause 6.3.2.3, page 42, lines 5-6:

> ... if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined.

Therefore, an alias is not permitted and the optimization is allowed.

Case 2: declared objects `f((int *)&d, &d)`

Subclause 6.3, page 38, lines 18-27 list specific ways in which declared objects can be accessed. Therefore, an alias is not permitted and the optimization is allowed.

Case 3: any other, including `malloc`ed objects `f((int *)dp, dp)`

We must take recourse to intent. The intent is clear from the above two citations and from Footnote 36 on page 38:

> The intent of this list is to specify those circumstances in which an object may or may not be aliased.

Therefore, this alias is not permitted and the optimization is allowed.

In summary, yes, the rules do apply to dynamically allocated objects.

Defect Report #029

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-016 (Sam Kendall)

Question 1

Subclause 6.1.2.6 says:

> ... two structure, union, or enumeration types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

I have one question and one clarification, both about compatibility between two **`struct/union/enum`** types declared in separate translation units.

(1) Was it the Committee's intent that the two types must have the same tag (or both lack tags) to be compatible? As the standard is written, the following is legal:

```
One Translation Unit:
struct foo { int i; } x;

Another Translation Unit:
extern struct bar { int i; } x;
```

Recommendation: This seems like an accidental omission. To be compatible, the two types should have the same tag, or both lack tags. I would guess that such was the Committee's intent.

(2) Clarification: The phrase "two structure, union, or enumeration types" should be written "two structure types, two union types, or two enumeration types." The current standard, interpreted literally, allows a structure and a union with identical member lists to be compatible, even though this is clearly not the intent of the Committee.

```
One Translation Unit:
union foo { int i; } x;
union bar { int i, j; } y;

Another Translation Unit:
extern struct foo { int i; } x;
extern struct bar { int i, j; } y;
```

Response

Subclause 6.1.2.6 says (by omission) that tags do not have to be the same for structure, union, or enumeration types to be compatible in separate translation units. Tags are used in succeeding declarations to ensure that they are of the same type. They are not used for type compatibility.

Does "two structure, union, and enumeration types" mean "two structure types, two union types, or two enumeration types?"

Answer: Yes.

Defect Report #030

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-017 (Pawel Molenda)

Question 1

Reference: subclause 7.5.1 **Treatment of error conditions**, page 111, lines 14-17:

> For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. ... an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.

If `sin(DBL_MAX)` results in `errno` being set to `EDOM`, is this a violation of the standard? If yes, what should be the result of this call?

Response

Subclause 7.5.1 does not give license for an implementation to set `errno` to `EDOM` for `sin(DBL_MAX)`. The mathematical function is defined for that argument value. While a conforming hosted implementation must not set `errno` to `EDOM` for this case, the standard imposes no constraint on the accuracy of the result value.

Defect Report #031

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-018 (Pawel Molenda)

Question 1

Referring to subclause 6.3, page 38, lines 15-17:

> If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

and subclause 6.4, page 55, lines 11-12:

> Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

What should be the result of the constant expression:

```
INT_MAX + 2
```

Is this a constraint violation, or it should be mapped onto the set of representable values?

What should be the result of:

```
INT_MAX + 2ul
```

How should compilers that do not evaluate the constant expressions at compile time behave?

What is the result of:

```
(INT_MAX*4)/4
```

Referring to subclause 6.5.2.2, page 61, lines 29-30:

> The expression that defines the value of an enumeration constant shall be an integral constant expression that has a value representable as an `int`.

What is the result of:

```
enum { a=INT_MAX, b };
```

Does this violate the C Standard?

Response

`case INT_MAX + 2:` is a constraint violation.

`case INT_MAX + 2ul:` is okay, representable.

`case (INT_MAX*4)/4:` is a constraint violation.

When subclause 6.4 says on page 55, lines 11-12:

> Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

the Committee's judgement of the intent is that the "representable" requirement applies to each subexpression of a constant expression, as shown in the third example. A constant expression is meant as defined by the syntax rules.

`enum {a=INT_MAX, b};` is a constraint violation.

Defect Report #032

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-036 (Stephen D. Clamage)

Question 1

In subclause 6.4, page 55, line 10, a constraint specifies that a comma operator may not appear in a constant expression (except within the operand of a `sizeof` operator).

At the end of the same section, page 56, line 1, it says, "An implementation may accept other forms of constant expressions."

Does the later statement give a license to relax the earlier constraint? For example, may a conforming implementation accept

```
int i = (1, 2);
```

without issuing a diagnostic?

Response

No, a conforming implementation may not accept this example without issuing a diagnostic. Constraint violations always require a diagnostic (subclause 5.1.1.3). The intent of the statement "An implementation may accept other forms of constant expressions" (subclause 6.4) is to allow an implementation to accept syntactic forms, such as might be generated by the `offsetof` macro, that may not otherwise be semantically allowed.

Defect Report #033

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-037 (Mike Vermeulen)

Question 1

Is a conforming implementation required to diagnose all violations of "shall" and "shall not" statements in the standard, even if those statements occur outside of a section labeled **Constraints**?

An example that illustrates this question is:

```
struct s { char field:1; };
```

This fragment violates a statement in subclause 6.5.2.1 on page 60, line 30: "A bit-field shall have a type that is a qualified or unqualified version of one of `int`, `unsigned int`, or `signed int`." Must a conforming implementation issue a diagnostic for this violation of "shall?"

Following are two different ways in which the C Standard has been interpreted. These interpretations came up during discussions over NIST conformance tests for an ANSI-C FIPS. I would like to ask the Committee for an interpretation of this issue, perhaps based on one or both of the interpretations given.

Suggested Interpretation #1:

Clause 3 **Definitions and conventions** states in the very beginning: "In this International Standard, 'shall' is to be interpreted as a requirement on an implementation or on a program; conversely, 'shall not' is to be interpreted as a prohibition."

Therefore *every* "shall" is viewed as testable. The question is what happens if a "shall" is violated.

Subclause 5.1.1.3 **Diagnostics** provides the answer: "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*. Diagnostic messages *need not be* produced in other circumstances." (emphasis added)

Therefore every violation of a "shall" should be treated as a failure to meet the requirements of the C Standard (first definition). Any violation of syntax rules, semantic rules, or sections labeled as **Constraints** should therefore generate a diagnostic.

According to this interpretation, a diagnostic should be produced for the example given above.

Suggested Interpretation #2:

Subclause 5.1.1.3 states that diagnostics must be produced "for every translation unit that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances."

Syntax rules are those items listed in the **Syntax** sections of the standard. *Constraints* are those items listed in the **Constraints** sections of the standard.

The C Standard specifies in clause 3, page 3, lines 12-13 that when the words "shall" or "shall not" appearing outside of a constraint are violated, the behavior is undefined.

For undefined behavior, the C Standard specifies in clause 3, page 3, lines 6-7 that the standard "imposes no requirements." Thus a conformance suite should not test for the words "shall" or "shall not" outside of a **Constraints** section, since the standard imposes no requirements.

According to this interpretation, the C Standard imposes no requirements on a conforming implementation for the program fragment above. A conforming implementation could choose to accept this program (see also Footnote 6 to subclause 5.1.1.3 on page 6), it could issue a diagnostic, or have any other behavior.

Response

Concerning a violation of subclause 6.5.2.1, **Semantics**, page 60, line 30: No diagnostic is required; this is undefined behavior. It is not a violation of a constraint or syntax.

Concerning a violation of clause 3, page 2, lines 2-3, No diagnostic is required.

Suggested Interpretation #2 is the correct one.

Conformance to FIPS is beyond the scope of WG14. We can't comment on this.

Defect Report #034

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-038 (Stephen D. Clamage)

Question 1

In *The C Users Journal*, Vol. 8 No. 7, July 1990, P.J. Plauger gives the following example on page 10:

```
extern int a[];
int f() {
    extern int a[10];
    ...
    }
int sizea = sizeof a;     /* error */
```

Mr. Plauger claims that the size information from the inner scope "evaporates" when its scope ends, and the operand to the `sizeof` operator has an incomplete type. We cannot find unequivocal support for this claim in the standard.

Subclause 6.1.2.2 says on page 21, lines 10-11:

> ... each instance of a particular identifier with *external linkage* denotes the same object or function.

Combining subclause 6.1.2.6 and subclause 6.5.4.2, we find that the two declarations for **a** are compatible and we may construct a composite type. The composite type is array of 10 `int`.

Subclause 6.1.2.6 on page 25, lines 19-20, discusses the case of two declarations in the same scope, but does not discuss the case of two declarations for the same object in different scopes.

But subclause 6.1.2.5 says on page 24, lines 8-9:

> An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage).

The identifier **a** appears in two declarations, and denotes the same object. The second declaration completes the type for the identifier in the inner scope. The two identifiers denote the same object, so it would seem reasonable to say the type of that object is completed.

Is the size information in the inner scope lost upon leaving the scope?

Response

Is the size information in the inner scope lost upon leaving the scope?

Answer: Yes.

See the correction in response to Defect Report #011.

Question 2

If no size information is known in the outer scope, then consider the following example:

```
extern int a[];
int f() {
    extern int a[10];
    ...
    }
int g() {
    extern int a[20];     /* error? */
    ...
    }
```

Is this legal? If not, does it violate a constraint?

Response

The example exhibits undefined behavior. It does not violate a constraint. Subclause 6.1.2.2, page 21, lines 10-13 describe "same object;" subclause 6.1.2.6, page 25, lines 9-10 require that "All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined."

Defect Report #035

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-039 (Derek M. Jones)

Question 1

```
void f(a, b)
int a(enum b {x, y});
int b;
{
}
```

Now this example is perverse because a prototype declaration is used to declare the parameter of an old-style function declaration. But anyway ...

Is the declaration of the parameter `a` legal or a constraint error?

Now `a(...)` is a declarator.

Subclause 6.7.1 says on page 82, lines 7-8:

> ... each declaration in the declaration list shall have at least one declarator, and those declarators shall declare only identifiers from the identifier list.

The identifier list contains `a` and `b`.

The declarator for parameter `a` declares the identifiers `a`, `b`, `x`, and `y`.

`b` is in the identifier list, so that is okay. But `x` and `y` are not. Constraint error (methinks so)?

See subclause 6.1.2, page 19 for a definition of an identifier.

Response

There is no constraint violation. The scopes of `b`, `x`, and `y` end at the right-parenthesis at the end of the `enum`, so there is no violation. It is difficult to *call* the function `f`, but there is no constraint violation. The phrase "each declarator declares one identifier" in subclause 6.5.4 refers to `a`, not to `b`, `x`, or `y`.

As an example, in the conforming definition:

```
void f(a, b)
int a(enum b{x, y});
int b;
{
}
```

the scope of `b` (the enum tag), `x`, and `y` ends at the right-parenthesis at the end of the enum (prototype scope).

Question 2

Also consider:

```
void g(c)
enum m{q, r} c;
{
}
```

What is the scope of `m`, `q`, and `r`?

Subclause 6.1.2.1 says on page 20, lines 28-29 "... appears outside of any block or list of parameters, the identifier has *file scope*, ..."

It says on page 20, lines 30-31 "... appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, ..."

Now the above three identifiers appear outside of any block or list of parameters but they are within the list of parameter declarations.

Who wins?

Response

The scope of `m`, `q`, and `r` ends at the close-brace (block scope). The operative wording is the more specific statement on page 20, lines 30-31 "... appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, ..."

As an example, in the code fragment:

```
void g(c)
enum m{q, r} c;
{
}
```

the scope of `m`, `q`, and `r` ends at the closing brace of the function definition (block scope).

Defect Report #036

Submission Date: 10 Dec 92

Submittor: WG14

Source: X3J11/91-040 (Fred Tydeman)

Question 1

May floating-point constants be represented with more precision than implied by its type? Consider the following code fragment:

```
float f;
double d;
long double ld;
ld = ld + 0.1;          /* add a long double and a double */
ld = ld + 1.0 / 10.0;   /* expression with "same" value */
d = f + 0.1f;           /* "+" is allowed to be double precision */
```

In the above example, the decimal number `0.1`, when converted to binary, is a non-terminating repeating binary number; so the more bits used to represent the number, the closer it will be to its true value. Hence, if **doubles** are 64 bits and **long doubles** are 80 bits, the **long double** will be more accurate. So in essence, may `0.1` (a **double**) be represented with more precision, e.g. as `0.1L` (a **long double**)?

Parts of the C Standard that may help answer the question follow.

Subclause 5.1.2.3 **Program execution**, page 7, line 36:

> In the abstract machine, all expressions are evaluated as specified by the semantics.

I believe that this is the "as if" rule that applies to this case.

Subclause 5.1.2.3 **Program execution**, page 8, lines 44-45:

> Alternatively, an operation involving only **ints** or **floats** may be executed using double-precision operations if neither range nor precision is lost thereby.

Clearly, `d = f + 0.1F` may be done using a double-precision add. But may `0.1f` be represented as the **double** `0.1`?

Subclause 6.1.3.1 **Floating constants**, page 26, lines 32-35:

> If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner.

I believe that the above does not require that the result be the nearest representable value (for its type).

Subclause 6.2.1.5 **Usual arithmetic conversions**, page 35, lines 38-39:

> The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.

I believe that a floating constant is a floating operand, so is allowed greater precision. Clearly, the expression `1.0 / 10.0` is allowed greater precision than just **double**, so it would make sense to allow an equivalent constant (`0.1`) to have greater precision.

Subclause 6.4 **Constant expressions**, page 55, lines 14-16:

> If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment.

Response

The Committee concurs with all the arguments presented — a floating constant may be represented in more precision than implied by its type.

Defect Report #037

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-043 (Isai Scheinberg)

Question 1

Subclause 5.2.1.2 **Multibyte characters** states:

> The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:
>
> — The single-byte characters defined in 5.2.1 shall be present.

and, a bit later on:

> — A byte with all bits zero shall not occur in the second or subsequent bytes of a multibyte character.

My interpretation (and all of the experts that I consulted with) of the first rule, is that the basic character set (A-z, 0-9, etc.) shall be coded in one-byte code. All multibyte locales that I know (EUC variants, SJIS) follow this rule. But I may still be wrong.

If the above is true, then both 10646 (other than CM 5) and UNICODE fail this rule and cannot be used as multibyte characters. UNICODE also fails the second rule.

Response

The following answers apply (almost) equally to ISO 10646-1 and UNICODE. They are expressed in terms of ISO 10646-1.

Clause 3, page 2, lines 18-24 and 40-42 define "byte," "character," and "multibyte character" as follows:

> **byte**: The unit of data storage large enough to hold any member of the basic character set of the execution environment.
>
> **character**: A bit representation that fits in a byte. The representation of each member of the basic character set in both the source and execution environments shall fit in a byte.
>
> **multibyte character**: A sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set."

Therefore, if ISO 10646-1 were used as a basic character set, then by definition a byte would have to be large enough to hold each member of the ISO 10646-1 character set. Also by definition this would make ISO 10646-1 a valid multibyte character set.

If a byte were only eight bits long, the following answer would hold. ISO 10646-1 represents, in a particular byte order, the character `'a'` for example as follows.

```
0 0 0 97
    ----   16-bit version
--------   32-bit version
```

This fails subclause 5.2.1.2, page 11, lines 30-32:

> — A byte with all bits zero shall be interpreted as a null character independent of shift state.
>
> — A byte with all bits zero shall not occur in the second or subsequent bytes of a multibyte character.

Therefore, 8-bit bytes preclude the use of ISO 10646-1 as a multibyte character set.

Defect Report #038

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-046 (Kuo-Wei Lee)

Question 1

Under subclause 6.8.3.1 **Argument substitution**, the C Standard states on page 90, lines 12-14:

> Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they form the rest of the translation unit; no other preprocessing tokens are available.

It is not clear to us what should happen if, after the first replacement, the argument is a valid preprocessing number. Consider the following example:

```
#define X 0x000E
#define Y 0x0100
#define FOO(a) a
FOO(X+Y)
```

After `X` is replaced, `FOO(X+Y)` becomes `FOO(0x000E+Y)`. At this point, should the macro replacement continue and expand `Y` to be 0x0100 with the final result being `FOO(0x000E+0x0100)`; or should the expansion stop since `0x000E+Y` is a syntactically valid preprocessing number?

In other words, should `FOO(X+Y)` be expanded into `FOO(0x000E+0x0100)`, or should it be `FOO(0x000E+Y)`?

Response

Subclause 5.1.1.2, page 5, lines 32-39 indicate that translation must proceed as if all creation of preprocessing tokens completes before any macro expansion begins. These are translation phases 3 and 4:

> 3. The source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments)...
>
> 4. Preprocessing directives are executed and macro invocations are expanded.

Therefore, if `X+Y` were expanded to `0x000E+Y`, a new preprocessing number would not be created. The macro expansion proceeds as follows.

```
FOO(X+Y)              (6 tokens) -->
FOO(0x000E+0x0100)    (6 tokens) -->
0x000E+0x0100         (3 tokens)
```

This sequence is required by subclause 6.8.3.1, page 90, lines 10-14:

> A parameter in the replacement list ... is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the translation unit...

Defect Report #039

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-061 (Vania Joloboff)

Question 1

My interpretation of the Standard is that the value of `MB_CUR_MAX` must be one in the `"C"` locale. I infer that from the fact that:

1) The characters in the `"C"` locale must be alphanumeric + space.
2) The `isXXX` functions specify character constant values for the `"C"` locale.
3) A character constant consists of one or more characters that are enclosed within apostrophes. A character is regarded as having type `char`.
4) The data type `char` consists of one byte of storage.

However this clarification should be made explicit.

Response

In fact 3, we presume the second sentence was intended to be: "A character *constant* is regarded as having type `char`," in order to be applicable to this request. That is not true; a character constant is of type `int`. Also facts 1-4 deal with the single-byte chars and not the extended character set.

In any case, the facts as listed do not logically lead to the conclusion that `MB_CUR_MAX` must be one (1) in the `"C"` locale. In fact, this conclusion is not true. It is possible for `MB_CUR_MAX` to be greater than one in the `"C"` locale. In subclause 7.10, page 149, `MB_CUR_MAX` is "the maximum number of bytes in a multibyte character for the extended character set specified by the current locale." In subclause 7.4.1.1, page 107, the `"C"` locale is "the minimal environment for C translation." The minimal environment may still require more than one byte for multibyte characters.

Question 2

I also would like to make a requirement that if the current locale is the `"C"` locale, the value returned by `setlocale(LC_ALL, NULL)` be a string of length one, consisting of the single character `C`.

Currently the value of `setlocale(LC_ALL, NULL)` is unspecified for the `"C"` locale.

This makes it difficult to build libraries where you want to maintain the behavior pre-existing to internationalization for backward compatibility.

Typically you want to say in these programs:

```
if (*setlocale(LC_ALL, NULL) == 'C')
     do the old thing
else
     do the new thing
```

Response

The reference is to subclause 7.4.1.1, page 107.

The Committee acknowledges that there exists no strictly portable method for determining whether the current locale is the `"C"` locale. The request for this feature is neither an erratum nor a request for interpretation; it is a request for an amendment. The Committee will consider this request for a future revision of the C Standard.

Defect Report #040

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-062 (Derek M. Jones)

Question 1

Composite type

Rule for function parameter compatibility, subclause 6.7,1, page 82, lines 24-25:

```
void f(const int);
void f(int a)
{
      a = 4;
}
```

In the above case what is the composite type of `f`? The legality of the assignment to `a` depends on the answer.

```
int f(int a[4]);
int f(int a[5]);
```

The parameters are compatible because they are converted to *pointer to* ..., but what is the composite type?

Response

```
void f(const int);
void f(int a)
{
      a = 4;
}
```

What is the composite type of `f`?

Answer: `void f(int)`. Defect Report #013, Question 1 describes the correct manner for constructing the composite type.

Is the assignment valid?

Answer: Yes. The type of a parameter is independent of the composite type of the function, so the assignment is valid (cf. subclause 6.7.1).

Another example:

```
int f(int a[4]);
int f(int a[5]);
```

The parameters are compatible because they are converted to *pointer to* ..., but what is the composite type?

Answer: The response to the Defect Report mentioned above answers this question as well.

Question 2

Is an implementation that fails to equal (or exceed) the value of an environmental limit conforming? Subclause 5.2.4 says that those in that subclause must be equalled in a conforming implementation. There is no such wording for the environmental limits in the Library (subclauses 7.9.2, 7.9.3, 7.9.4.4, 7.9.6.1, 7.10.2.1).

Correction

Add to subclause G.2, page 203:

— 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).

Question 3

Is an "environmental constraint" a constraint?

In subclause 7.6.1.1, page 118, lines 22-30, we have a set of environmental constraints on where `setjmp` may occur.

Does violating these rules require a constraint error to be flagged, or is it undefined behavior?

Some examples:

```
i = setjmp(a);
if (setjmp(a) == i)
...
```

Response

Must an implementation diagnose violations of environmental constraints?

Answer: Diagnostics are not required for constraint violations in clause 7, since subclause 5.1.1.3 refers to a constraint as defined in clause 3, which applies to language elements only.

Question 4

For the fragment

```
if (a<b||c>d)
    ;
```

Defect Report #017 Question 39 states that this is lexed as:

a) `{if} {(} {a} {<} {b} {||} {c} {>} {d} {)}`

not as:

b) `{if} {(} {a} {<b||c>} {d} {)}`

The rationale for this interpretation was that the constraint in subclause 6.1.7, page 32, lines 33-34 disallowed a header name preprocessing token anywhere except within a `#include`. Since the header name preprocessing token could not exist it was not lexed as such.

It was pointed out that the "longest possible token" rule was not influenced by rules elsewhere in the C Standard, i.e. `i+++++j` is lexed as:

c) `{i} {++} {++} {+} {j}`

not as:

d) `{i} {++} {+} {++} {j}`

Now (c) is a constraint violation by subclause 6.3.2.4, page 42, lines 38-39, the operand of the second ++ is not a modifiable lvalue. But this constraint does not require that the input be re-lexed to form the preprocessing tokens given in (d), which is conforming code.

As the UK C Panel saw it, the first example should be lexed as given in (b) and a diagnostic issued. Having violated a constraint, we are now into undefined behavior. An implementation could define the behavior in this circumstance to be a re-lex of the input to produce the preprocessing tokens given in (a).

As far as the user was concerned, they would get the expected behavior with the added value of a diagnostic being issued.

All those present felt that the interpretation was incorrect and recommended that the UK ask the Committee to reconsider its decision.

To summarize, there is no ambiguity in the C Standard and the original X3J11 interpretation is incorrect.

Response

Is a diagnostic required for an input such as

```
if (a<b||c>d)
```

because of a violation of the constraint specified in subclause 6.1.7, page 32, lines 33-34?

Answer: No. Our response to Defect Report #017 Question 39 addresses this issue.

Question 5

In the constraint for subclause 6.5.2, page 59, lines 2-4: What does the C Standard mean when it says "set?"

Does it mean that the construct:

```
int int i;
```

violates a constraint?

It has been suggested that this wording was left vague to allow such constructs as `long long` (which is supported by some compilers) to fall into the undefined behavior category.

Would the Committee clarify the situation with regard to duplicate type specifiers? Do such constructs result in a constraint error or undefined behavior?

The related case `static static` is explicitly ruled out by the constraints in the previous subclause.

Additionally, `volatile volatile` is ruled out by the constraint in subclause 6.5.3.

Response

Example:

```
int int i;
```

Must this be diagnosed?

Answer: Yes. It is allowed to rearrange the order of type specifiers within a set, but not to duplicate them (cf. subclause 6.5.2). Thus `int int` is a constraint violation.

Question 6

The definition of the `offsetof` macro in subclause 7.1.6 does not cover all its possible occurrences:

a) There are no restrictions on the structure being a completed type.

```
struct t1 {
      char c;
      short s;
      int i[offsetof(struct t1, s)];
      }
```

When discussing the use of incomplete types, recourse usually has to be made to the rules relating to where an object of unknown size may appear.

Would the Committee agree that there are not any rules prohibiting the above construction?

b) In this structure we are asked to find the offset of a field that has not yet been encountered:

```
struct t2 {
      char c;
      union {
            int i[offsetof(struct t2, s)];
            short s;
            } u;
      };
```

Would the Committee agree that there do not appear to be any rules that make this construct illegal?

c) The following structure has infinitely many "solutions:"

```
struct t3 {
      char a[offsetof(struct t3, i)];
      int i;
      }
```

since `char` has size 1, any size of array will be the same as the `offsetof` the field `i`.

d) The following structure has no "solutions:"

```
struct t4 {
      int a[offsetof(struct t3, i)];
      int i;
      }
```

`int` is always larger than 1.

Response

a) Example:

```
struct t1 {
char c;
short s;
int i[offsetof(struct t1, s)];
};
```

This is *not* a valid use of the `offsetof` macro. The hypothetical `static type t;` declaration required for `offsetof` (cf. subclause 7.1.6) could not have validly appeared prior to the invocation of `offsetof`

because the type `struct t1` is incomplete (cf. subclause 6.7.2); therefore the `offsetof` invocation is not strictly conforming.

b) The answer is the same as (a) above. In addition, the members mentioned in these invocations are not in scope.

c) The answer is the same as (a) above. In addition, the members mentioned in these invocations are not in scope.

d) The answer is the same as (a) above. In addition, the members mentioned in these invocations are not in scope.

Question 7

`sizeof` various identifiers (subclause 7.1.6)

a)

```
void f(int c, char a[sizeof(c)]);
```

b)

```
int i;
struct {
      int i;
      char a[sizeof(i)];
      };
```

Now the argument to `sizeof` must be an expression or a type.

In (a) is `c` an expression? I think not because:

expression –> object –> has storage in execution envrionment

and `c` does not have storage allocated to it. So (a) violates a semantic "shall" and is undefined behavior.

Now in (b) the field `i` is obviously not an expression. But is it visible? Like the outer `i`, it has file scope. However, it is in a different namespace. There are no rules for namespace resolution in the `sizeof` subclause.

So is (b) legal or undefined behavior?

Response

a) Example:

```
void f(int c, char a[sizeof(c)]);
```

The reference to `c` is an expression because the previosly declared identifier designates a function parameter (cf. subclause 6.5.4.3), which is an object (subclause 3.15), thus meeting the requirement in subclause 6.3.1.

b) Another example:

```
int i;
struct {
      int i;
      char a[sizeof(i)];
      };
```

In C, this is okay. Subclause 6.1.2.3, **Name spaces of identifiers**, requires that `i` in the `sizeof` expression refers to the external `i`, not the member.

Question 8

Refer to subclause 6.1.2.5, page 22, lines 32-36:

a)

```
char c = 7;     /* implementation defined behavior, since 7 need not
          be a member of the basic execution character set */
```

b)

```
c = 'a';  /* ok */
c++;      /* implementation defined */
```

c)

```
c = '1';  /* ok */
c++;      /* ok? */
```

It has been suggested that the above constructs are not implementation defined.

Subclause 6.1.3.4, page 29, lines 30-33:

d)

```
c = '\07';     /* what is in the source/execution character set is
          given in subclause 5.2.1. Anything else is an extension. */
```

e)

```
c = '$';
```

It has been suggested that characters may be added to the basic source/execution character set without implementation defined behavior being invoked. (I guess my position on this item can be deduced from the text.)

Response

a) Subclause 6.1.2.5 says "An object declared as type `char` is large enough to store any member of the basic execution character set... If other quantities are stored in a `char` object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers." Consider this example:

```
char c = 7;
```

The assignment `c = 7` is not implementation-defined because, from a reasonable reading of subclause 6.1.2.5, it is clear that the only implementation-defined behavior here is the signedness of the value of the `char` object.

b) Another example:

```
c = 'a';
c++;
```

The increment of `c` after assigning an `'a'` to it is defined by the implementation because the numeric encoding of `'a'` is defined by the implementation. If `'a'` were equal to `CHAR_MAX`, the increment could even cause an overflow (cf. subclause 5.2.1).

c) Another example:

```
c = '1';
c++;
```

The increment of `c` after assigning a `'1'` to it is not implementation-defined because the characters `'0'` through `'9'` are required to be a contiguous range (cf. subclause 5.2.1). Thus, the result is `'2'`.

d) Another example:

```
c = '\07';
```

The value of the character constant `'\07'` is defined by the C Standard (cf. subclause 6.1.3.4, page 29, line 10-13). The implementation-defined behavior of some escape sequences, described on page 29, lines 30-33, is clarified in the example on page 30, lines 8-14.

e) Another example:

```
c = '$';
```

If `$` is in the execution character set, the value of `'$'` is locale-specific and so must be defined by the implementation (cf. subclause 5.2.1).

Question 9

re: UK request for interpretation cai027 (Defect Report #017 Question 27)

X3J11 refs: 90-056, 90-083

It has been pointed out, and the UK C panel agreed at its last meeting, that the request for interpretation was unnecessary. The C Standard was clear and unambiguous as is.

To make matters worse, X3J11 appears to have given an interpretation that is the opposite of what the C Standard says.

The UK would like to withrdaw this request for interpretation and ask the Committee to reconsider its position.

Response

We reaffirm the previous interpretation.

Defect Report #041

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/91-076 (Andrew Josey)

Question 1

Does the description in subclause 7.3.1 imply that the characters defined in subclause 5.2.1 are always classified as implied by subclause 5.2.1 regardless of the locale specified?

In particular, do the characters `'a'` through `'z'` and `'A'` through `'Z'` have to be classified as "lower case and "upper case," respectively, in every locale?

The specific lines needing interpretation are lines 20-21 in subclause 7.3.1.6, page 103, and lines 16-17 in subclause 7.3.1.10, page 104. The word "or" can be interpreted to require a superset of the characters specified as lower/upper case in subclause 5.2.1 or to allow an implementation-defined set of characters (which might contain none of the subclause 5.2.1 designated lower/upper case characters).

Response

Does the description in subclause 7.3.1 imply that the characters defined in subclause 5.2.1 are always classified as implied by subclause 5.2.1 regardless of the locale specified?

Answer: By subclause 7.3.1.6 **The `islower` function** and subclause 7.3.1.10 **The `isupper` function** which refer to lower- and upper-case letters, respectively, and by subclause 5.2.1: "basic source and basic execution character sets shall have at least ... upper-case letters of the English alphabet" (with example) ... "lower-case letters of the English alphabet" (with example), and by subclause 5.2.1.2 "The single-byte characters defined in subclause 5.2.1 shall be present," which refers to multibyte characters, therefore, yes, the characters defined in subclause 5.2.1 are always classified as implied by subclause 5.2.1 regardless of the locale specified.

Do the characters `'a'` through `'z'`, and `'A'` through `'Z'`, have to be classified as "lower case" and "upper case," respectively, in every locale?

Answer: Yes, the characters `'a'` through `'z'`, and `'A'` through `'Z'`, have to be classified as "lower case" and "upper case," respectively, in every locale (following the citations above).

Defect Report #042

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-001 (Tom MacDonald)

Question 1

The description of **memcpy** in subclause 7.11.2.1 says:

```
void *memcpy(void *s1, const void *s2, size_t n);
```

> The **memcpy** function copies **n** characters from the object pointed to by **s2** to the object pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

The definition of the term *object* in subclause 3.14 is:

> **object** — A region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. When referenced, an object may be interpreted as having a particular type...

Are the objects in the description of **memcpy** the largest objects into which the arguments can be construed as pointing?

In particular, is the behavior of the call of **memcpy** in Example 1 defined:

```
void f1(void) {
     extern char a[2][N];
     memcpy(a[1], a[0], N);
     }
```

because the arguments point into the disjoint array objects, **a[1]** and **a[0]**? Or is the behavior undefined because the arguments both point into the same array object, **a**?

Response

From subclause 3.14, an object is "a region of data storage ... Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined ..." From subclause 7.11.1, "the header **<string.h>** declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type." "Various methods are used for determining the lengths of the arrays..." From subclause 7.11.2.1, description of **memcpy**, "if copying takes place between objects that overlap, the behavior is undefined." Therefore, the "objects" referred to by subclause 7.11.2.1 are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of **N** bytes in length (i.e. treated as an array of **N** elements of character type).

a) So, no, the objects are not "the largest objects into which the arguments can be construed as pointing."

b) In Example 1, the call to **memcpy** has defined behavior.

c) The behavior is defined because the pointers point into different (non-overlapping) objects.

Question 2

For the purposes of the description of **memcpy**, can a contiguous sequence of elements within an array be regarded as an object in its own right? If so, are the objects in the description of **memcpy** the smallest contiguous sequences of bytes that can be construed as the objects into which the arguments point?

In Example 2:

```
void f2(void) {
     extern char b[2*N];
     memcpy(b+N, b, N);
     }
```

can each of the first and last half of array **b** be regarded as an object in its own right, so that the behavior of the call of **memcpy** is defined? (Although they are not declared as separate objects, each half does seem to satisfy the definition of object quoted above.) Or is the behavior undefined, since both arguments point into the same array object **b**?

In Example 3:

```
void f3(void) {
     void *p = malloc(2*N);    /* Allocate an object. */
     {
     char (*q)[N] = p;    /* The object pointed to by p may
                             be interpreted as having type
                             (char [2][N]) when referenced
                             through q. */
     /* ... */
     memcpy(q[1], q[0], N);
     /* ... */
     }
     {
     char *r = p;         /* The object pointed to by p may
                             be interpreted as having type
                             (char [2*N]) when referenced
                             through r. */
     /* ... */
     memcpy(r+N, r, N);
     /* ... */
     }
}
```

the types of the objects are inferred from the pointers, and the underlying storage is dynamically allocated. Is the behavior of each call of **memcpy** defined?

Since the relationship between the values of the arguments presented to **memcpy** is the same in all the above calls, it seems reasonable to expect that either all these calls of **memcpy** give defined behavior, or none do. But which is it?

Response

a) Yes, for **memcpy**, a contiguous sequence of elements within an array can be regarded as an object in its own right.

b) The objects are not the smallest contiguous sequence of bytes that can be construed; they are exactly the regions of data storage starting at the pointer and of **N** bytes in length.

c) Yes, the non-overlapping halves of array **b** can be regarded as objects in their own rights.

d) The behavior (in Example 2) is defined.

e) The definition of object is independent of the *method* of storage allocation. The array length is determined by "various methods." So, yes, the behavior of each call of **memcpy** is well-defined.

f) All of the calls of **memcpy** (in Example 3) give defined behavior.

Question 3

Similar questions arise for the other library string handling functions that have undefined behavior when copying between overlapping objects. These include **strcpy**, **strncpy**, **strcat**, **strncat**, **strxfrm**, **mbstowcs**, **wcstombs**, **strftime**, **vsprintf**, **sscanf**, and **sprintf**. For these functions, however, the number of bytes referenced through each pointer depends, at least in part, upon the values stored in the bytes.

Consider a library function for which the number of bytes accessed or modified is affected by the values of the bytes. Is the object associated with each of its pointer arguments the smallest contiguous sequence of bytes actually accessed or modified through that pointer?

In Example 4:

```
void f4(void) {
     extern char b[2*N];
     strcpy(b+N, b);
     }
```

is the behavior defined if **N > strlen(b)**?

In Example 5:

```
void f5(void) {
     extern char c[2*N];
     strcat(c+N, c);
     }
```

is the behavior defined if both `N > strlen(c)` and `N > strlen(c) + strlen(c+N)`?

Response

Length is determined by "various methods." For strings in which all elements are accessed, length is inferred by null-byte termination. For **`mbstowcs`**, **`wcstombs`**, **`strftime`**, **`vsprintf`**, **`sscanf`**, **`sprintf`** and all other similar functions, it was the intent of the C Standard that the rules in subclause 7.11.1 be applicable by extension (i.e., the objects and lengths are similarly dynamically determined). The behavior (in Examples 4 and 5) is defined.

Defect Report #043

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-004 (Robert Paul Corbett)

Question 1

Defining `NULL`

Subclause 7.1.6 defines `NULL` to be a macro "which expands to an implementation-defined null pointer constant." Subclause 6.2.2.3 defines a null pointer constant to be "an integral constant expression with the value 0, or such an expression cast to type `void *`." The expression `4-4` is an integral constant expression with the value 0. Therefore, Standard C appears to permit

```
#define NULL    4 - 4
```

as one of the ways `NULL` can be defined in the standard headers. By allowing such a definition, Standard C forces programmers to parenthesize `NULL` in several contexts if they wish to ensure portability. For example, when `NULL` is cast to a pointer type, `NULL` must be parenthesized in the cast expression.

At least one book about Standard C suggests defining `NULL` as

```
#define NULL    (void *) 0
```

That definition leads to a subtler version of the problem described above. Consider the expression `NULL[p]`, where `p` is an array of pointers. The expression expands to `(void *)0[p]` which is equivalent to `(void *)(p[0])`. I doubt many users would expect such a result.

Have I correctly understood Standard C's requirements regarding `NULL`? If not, what are those requirements?

Correction

Add to subclause 7.1.2, page 96 (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.

Question 2

Subclause 7.1.3 implies that an identifier that begins with an underscore cannot be defined as a macro name in any source file that includes at least one standard header. Footnote 91 emphasizes this restriction. Nonetheless, there are texts on Standard C that imply that such macro definitions are allowed.

The first paragraph of subclause 7.1.3 states that each header optionally declares or defines identifiers which are always reserved either for any use or for use as file scope identifiers. The second bullet item states, "All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces." The final sentence states, "If the program declares or defines an identifier with the same name as an identifier reserved in that context (other than as allowed by 7.1.7), the behavior is undefined." Taken together, these statements imply that an identifier that starts with an underscore cannot be defined as a macro in a source file that includes at least one of the standard headers.

Can an identifier that starts with an underscore be defined as a macro in a source file that includes at least one standard header?

Response

No. See subclause 7.1.3 and Footnote 91.

Defect Report #044

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-010 (Steve M. Hoxey)

Question 1

Subclause 7.1.6, page 98, lines 24-30 describe the macro
`offsetof(type, member_designator)`
"which expands to an integral constant expression that has type **`size_t`**, ..."
How is this statement to be interpreted? The expansion of the macro **`offsetof`** is

a) an expression which can be evaluated during translation, the value of which is in the range representable by a **`size_t`** type.

Or

b) an expression as (a) above, but further constrained to be an "integral constant expression" as defined in subclause 6.4, page 55, lines 17-21.

Response

Neither alternative (a) nor (b) in Question 1 fully captures the intent. What is intended is exactly what is specified in the C Standard. A strictly conforming program shall not produce output that varies depending upon details of implementation of facilities defined by the standard headers. Hence, use of the **`offsetof`** macro, in a context requiring an integer constant expression, per se does not render a program not strictly conforming.

Further clarification provided by David Prosser:

Although the replacement for the **`offsetof`** macro must be an integral constant expression, and must follow all the constraints appropriate to expressions, an implementation is permitted to make use of its extensions to constant expressions that behave like integral constant expressions. This is why the sample replacement expressions for the **`offsetof`** macro in the Rationale are valid candidates (for many implementations) but do not come under the strict definition of integral constant expression that strictly conforming code must follow. In particular, this is why the **`offsetof`** macro exists: there was otherwise no portable means to compute such translation-time constants. Therefore, of the two choices, (b) is the closest, but it is not the whole story.

Question 2

Subclause 5.1.1.1, page 5, lines 11-20 define a "translation unit" to be equivalent to the sequence of preprocessing tokens and white-space characters which exists at the end of translation phase 4 (subclause 5.1.1.2). Later in translation phases 5, 6, 7, these preprocessing tokens are converted to tokens and syntactically and semantically analyzed and translated.

Therefore, must a conforming implementation provide strictly conforming expansions of macros defined by the standard headers, such that any use of the resulting preprocessing token sequence, and ultimately the token sequence, beyond phase 4 does not alter the behavior of an otherwise strictly conforming program? See also clause 4 **Compliance**, page 4, lines 24-26.

Response

A conforming implementation need not provide strictly conforming expansion of macros defined by the standard headers.

Question 3

Assuming (b) is the correct interpretation of Question 1, if a particular implementation expands **`offsetof`** into an expression which contains operands and/or operators which result in a violation of the definition of "integral constant expression" from subclause 6.4, page 55, lines 17-21, does this situation constitute:

a) a constraint violation since the expansion presented for further translation is not an "integer constant expression?"

or

b) undefined behavior since the definition of "integral constant expression" appears in a "shall" requirement in the semantic description of subclause 6.4 **Constant expressions**?

Response

The response to Question 1 makes this a moot question.

Question 4

Assuming (b) is the correct interpretation of Question 3, if within a translation unit at a point where an "integer constant expression" is required to satisfy a language constraint — such as to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a case constant — does the use of the macro **`offsetof`** constitute:

a) a constraint violation?

or

b) the use of undefined behavior, which renders the translation unit to be not strictly conforming?

Response

The response to Question 1 makes this a moot question.

Question 5

Revisiting (b) as the correct interpretation of Question 1, it seems the only possibility for a definition of the macro **`offsetof`** constitutes use of an identifier from the reserved name space to define a builtin which interrogates the translator's symbol table in a fashion analogous to the **`sizeof`** operator. Further, this builtin must appear syntactically as a keyword rather than an identifier to avoid the constraint violation of subclause 6.4, page 55, line 9, which invalidates the use of what appears to be a function call within that which is otherwise required to be a constant expression.

Further, implementing an expansion for **`offsetof`** as described in the previous paragraph would violate the implementation constraint outlined in Question 2 above, since the expansion would inject preprocessing tokens requiring recognition of a keyword outside the scope of a strictly conforming program.

In any case, the implication is that the fragment:

```
#include <stddef.h>
static struct x {int field1, field2; } s;
enum fields {F0, F1, F2 = offsetof(struct x,field2), F3 };
```

is either rendered not strictly conforming or the implementation is rendered a nonconforming implementation.

Alternatively, if the answer to Question 2 above is no, then the following questions are raised:

Since translation phases 1 through 4 may introduce into the translation unit token sequences which are not strictly conforming, what mechanism exists, if any, to determine whether such sequences originated from the program source?

How is one to interpret the meaning of "strictly conforming program" from clause 4, page 3, lines 38-40, given that subclause 5.1.1.1, page 5, lines 12-15 define the translation unit to be "a source file together with all the headers and source files included via the preprocessing directive **`#include`**, less any source lines skipped by any of the conditional inclusion preprocessing directives?"

It seems that any program which makes use of the macro **`offsetof`** in the context of a constraint requirement mandating an "integer constant expression" will require use of unspecified, undefined, or implementation-defined behavior.

As near as I can tell, **`offsetof`** is the only macro defined by the C Standard which can alter the behavior of a strictly conforming program as a consequence of its own definition.

Response

The response to Question 1 addresses this issue.

Defect Report #045

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-036 (David J. Hendricksen)

Question 1

Under subclause 7.9.5.4 **The `freopen` function**, the C Standard states on page 130, lines 24-29:

> The **`freopen`** function opens the file whose name is the string pointed to by **`filename`** and associates the stream pointed to by **`stream`** with it. The **`mode`** argument is used just as in the **`fopen`** function.
>
> The **`freopen`** function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

It is not clear whether the following situations have defined behavior:

1) Calling **`freopen`** where **`stream`** points to uninitialized storage. For example:

```
{ FILE a, *b;
b = freopen("c.d", "r", &a);
}
```

(It may not be possible to detect that the information contained within **`a`** is not valid when the close for **`freopen`** is attempted.)

2) Calling **`freopen`** where **`stream`** is associated with a previously closed file. (The storage pointed to by **`stream`** may have been deallocated.)

Response

The behavior is undefined in both cases; case (2) is clear from subclause 7.9.3 **Files**, page 126, lines 24-27, "A file may be disassociated from a controlling stream by *closing* the file... The value of a pointer to a **`FILE`** object is indeterminate after the associated file is closed (including the standard text streams)." Also subclause 7.9.3 **Files**, page 126, lines 2-3 and lines 37-39, "A stream is associated with an external file... by *opening* a file, ... At program startup, three text streams are predefined and need not be opened explicitly..." Also subclause 7.9.5.3 **The `fopen` function**, and, similarly, subclause 7.9.5.4 **The `freopen` function**: "The ... function opens the file ... and associates a stream with it..." Thus when subclause 7.9.5.4 says "The **`freopen`** function ... associates the stream pointed to by **`stream`** with it," the intention is certainly that **`stream`** already points to a valid stream. Extending this to case (1), we observe that **`a`** (or **`&a`**) might not refer to a stream, since none has been "associated" by any means specified in the C Standard.

Defect Report #046

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-041 (Neal Weidenhofer)

Question 1

In subclause 6.7.1, page 82, line 9, it says, "An identifier declared as a typedef name shall not be redeclared as a parameter."

The question I have is: Does that sentence stand by itself absolutely or is it intended to be read in the context of the paragraph in which it appears?

The beginning of the paragraph says, "If the declarator includes an identifier list, ..." Function declarators including a parameter type list are dealt with in the preceding paragraph which says nothing about typedef names.

In other words, is the following valid Standard C?

```
typedef int foo;
int bar(int foo) {return foo; }
```

Response

The sentence is a part of the paragraph in which it appears. An identifier declared as a typedef name may be redeclared as a parameter in a parameter type list. The example is strictly conforming.

Defect Report #047

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-040 (Randall Meyers)

Question 1

Are the following declarations strictly conforming?

```
/* 1 */   struct S;
/* 2 */   struct S *f(struct S *p) {return p; }
/* 3 */   struct S *g(struct S a[]) {return a; }
/* 4 */   int *h(int a2[][]) {return *a2; }
/* 5 */   extern struct S es1;
/* 6 */   extern struct S es2[1];
```

The declaration of struct tag `S` introduces an incomplete type (subclause 6.5.2.3, page 62, lines 25-29) that may only be used when the size of the type is not needed.

The function `f` therefore is a fairly common and non-controversial use of an incomplete pointer type by a function. It is strictly conforming.

The function `g` is more interesting. A parameter of type array is adjusted to pointer type (subclause 6.7.1, page 82, lines 23-26). (Note that is an adjustment of the type of the parameter definition. It is not a conversion, as is what happens when an argument of type array is passed to a function.) Thus, the type of parameter `a` is pointer to struct `S`. This would seem to make the function `g` the same case as function `f`. However, subclause 6.1.2.5, page 23, lines 23-24 (also Footnote 17) disallow array types from having an incomplete element type (like struct `S`). This raises the question, is function `g` strictly conforming because the type of `a` is really pointer, or is function `g` not strictly conforming because `a` had an invalid array type before the compiler in effect rewrote the declaration?

The function `h` is similar to function `g`. The type of `a2` after adjustment is pointer to array of unknown size of `int`, which does not violate any rules. However, before adjustment, the type of `a2` is illegal because it is an array whose element type is array of unknown size, which is an incomplete type.

In previous Committee discussion that occurred concerning Defect Report #017 Question 10, the Committee took the position that a declaration like that of `es1` was strictly conforming, since the size of `es1` is not needed for an external reference, and thus was similar to the cases described in Footnote 63 in subclause 6.5.2.3 on page 62.

The declaration of `es2` also does not require its size to be known. However, it appears that the rule from subclause 6.1.2.5, page 23, lines 23-24 that prohibits an incomplete array element type makes `es2` not strictly conforming.

Response

First of all, no constraints are violated. Therefore, no diagnostics are required.

Declarations 1, 2, and 5 are strictly conforming. Declarations 3, 4, and 6 are not, and therefore cause undefined behavior.

The struct `S` is an incomplete type (subclause 6.5.2.3, page 62, lines 25-28). Also, an array of unknown size is an incomplete type (subclause 6.5.4.2, page 67, lines 9-10). Therefore, arrays of either of the above are not strictly conforming (subclause 6.1.2.5, page 23, lines 23-24). This makes declarations 3, 4, and 6 not strictly conforming. (But an implementation could get it right.)

As an aside, array parameters are adjusted to pointer type (subclause 6.7.1, page 82, lines 23-24). However, there is nothing to suggest that a not-strictly-conforming array type can magically be transformed into a strictly conforming pointer parameter via this rule.

The types in question can be interpreted two different ways. (Array to pointer conversion can happen as soon as possible or as late as possible.) Hence a program that uses such a form has undefined behavior.

Defect Report #048

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-043 (David F. Prosser)

Question 1

This Defect Report requests a clarification regarding the valid interpretations of the **abort** function, especially when the implementation must also match the requirements of POSIX.1 (ISO/IEC 9945-1:1990).

The C Standard states (subclause 7.10.4.1, page 155):

> The **abort** function causes abnormal termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open output streams are flushed or open streams closed or temporary files removed is implementation-dependent. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise(SIGABRT)**.

and (subclause 7.10.4.3, page 156):

> The **exit** function causes normal program termination to occur.

and (subclause 7.10.4.1, page 101 [Rationale]):

> The Committee vacillated over whether a call to **abort** should return if the signal **SIGABRT** is caught or ignored. To minimize astonishment, the final decision was that **abort** never returns.

The POSIX.1 Standard states (subclause 3.2, page 46):

> There are two kinds of process termination:
>
> (1) Normal termination occurs by a return from **main()** or when requested with the **exit()** or **_exit()** functions.
>
> (2) Abnormal termination occurs when requested by the **abort()** function or when some signals are received (see 3.3.1).
>
> The **exit()** and **abort()** functions shall be as described in the C Standard {2}. Both **exit()** and **abort()** shall terminate a process with the consequences specified in 3.2.2, except that the status made available to **wait()** or **waitpid()** by **abort()** shall be that of a process terminated by the **SIGABRT** signal.

and (subclause 8.2.3.12, page 161):

> The **exit()** function shall have the effect of **fclose()** ... as described above. The **abort()** function shall also have these effects if the call to **abort()** causes process termination, but shall have no effect on streams otherwise. The C Standard {2} specifies the conditions where **abort()** does or does not cause process termination. For the purposes of that specification, a signal that is blocked shall not be considered caught.

and (subclause B.8.2.3.12, page 291 [Rationale]):

> POSIX.1 intends that processing related to the **abort()** function will occur unless "the signal **SIGABRT** is being caught, and the signal handler does not return," as defined by the C Standard {2}. This processing includes at least the effect of **fclose()** on all open streams, and the default actions defined for **SIGABRT**.
>
> The **abort()** function will override blocking or ignoring the **SIGABRT** signal. Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-provided library functions.
>
> Note that the term "program termination" in the C Standard {2} is equivalent to "process termination" in POSIX.1.

The above quotes make it clear that the POSIX.1 Standard intends to have the abort function implementation be roughly the following:

1. Inquire about **SIGABRT** handling.

2. If currently blocked, unblock **SIGABRT**.
3. If currently **SIG_IGN**, reset **SIGABRT** to **SIG_DFL**.
4. If currently **SIG_DFL**, flush all open output streams.
5. **raise(SIGABRT)**.
6. Reset **SIGABRT** to **SIG_DFL** (handler must have returned).
7. Go to step 5.

As far as the C Standard is concerned, step 2 is outside its scope, so it can be part of a valid implementation. (The effects cannot be noticed by a strictly conforming program.) Step 4 is clearly permitted as well. It is step 3 and the loop that are the key of this Defect Report. (Note that step 3 could have been skipped above as it would be handled by the 5-6-7 loop, but I've left it explicit for clarity.)

The special case in the C Standard regarding **SIGABRT** handlers that don't return is intended to keep the implementation straightforward. (It is, in general, difficult to determine whether a handler will return without calling it!) The POSIX.1 Standard has understood the C Standard to require, in effect, an implementation to force an uncaught **SIGABRT** to terminate the program. But, is this actually the C Standard's intent? The Rationale quote can certainly be taken to indicate that catching and ignoring **SIGABRT** are in the same category.

Does the C Standard either permit or require an implementation to reset an ignored **SIGABRT** to **SIG_DFL**? Or, does the C Standard permit or require a call similar to **exit(EXIT_FAILURE)**? Is the distinction between abnormal termination and unsuccessful normal termination beyond the scope of the C Standard? (After all, how can it be tested?) And, finally, can a portable application find any utility in setting **SIGABRT** to **SIG_IGN**?

Response

Does the C Standard either permit or require an implementation to reset an ignored **SIGABRT** to **SIG_DFL**?

Answer: Yes, it permits it. There is no way to detect such a change in a strictly conforming program.

Or, does the C Standard permit or require a call similar to **exit(EXIT_FAILURE)**?

Answer: No. Abnormal termination does not allow calls to the **atexit**-registered functions.

Does the C Standard? (After all, how can it be tested?)

Answer: No. See above.

And, finally, can a portable application find any utility in setting **SIGABRT** to **SIG_IGN**?

Answer: Not within the context of **abort**.

We note that therefore there is no clash between Standard C and POSIX.1.

Defect Report #049

Submission Date: 10 Jan 93
Submittor: Project Editor (P.J. Plauger)
Source: David Metsky

Question 1

It has been suggested that, at least in the "C" locale, the transformed string output from **strxfrm** will not contain more characters than the original string. I believe that this suggestion is overly restrictive, and that the standard does not impose such a restriction on implementations. I am requesting a clarification from the appropriate standards committee(s). I hope that you will agree with the following resolution:

> The C Standard does not impose a requirement upon the length of the transformed string output from **strxfrm**. (The returned value does indicate the necessary length.)

Here are some citations from the C Standard:

Subclause 7.4.1.1 **The setlocale function:**

> **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions... A value of "C" for **locale** specifies the minimal environment for C translation... At program startup, the equivalent of
>
> ```
> setlocale(LC_ALL, "C");
> ```
>
> is executed.

Subclause 7.11.4.3 **The strcoll function:**

> The **strcoll** function compares the string pointed to by **s1** to the string pointed to by **s2**, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Subclause 7.11.4.5 **The strxfrm function:**

> The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings... The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is **n** or more, the contents of the array pointed to by **s1** are indeterminate.

I haven't located any requirement that the "C" locale behavior of **strcoll** must be identical to **strcmp**. Even if there were such a requirement, I haven't located any requirement that the transformed string must not be longer than the original string.

Response

We support your resolution:

The C Standard does not impose a requirement upon the length of the transformed string output from **strxfrm**, other than a limitation on the size of objects. (The returned value does indicate the necessary length.)

Defect Report #050

Submission Date: 24 Feb 93
Submittor: Project Editor (P.J. Plauger)
Source: C. Breeus

Question 1

Subclause 6.1.3.4 says that the type of a wide character constant is `wchar_t`, and subclause 6.1.4 says the type of a wide character string is array of `wchar_t`.

Subclause 7.1.6 says the typedef name `wchar_t` must be defined in `<stddef.h>`.

Question: When a compiler sees a literal of the form `L'...'` or `L"..."` must it not check that

1) The name `wchar_t` is visible at that place.
2) The name is a typedef name. It could be redefined in an inner scope.
3) It is a typedef for an integral type. Again, it could be redefined.

And then, take that integral type as the meaning of `wchar_t`. I suppose it cannot just hope for the best and take a type that makes it feel good.

Response

A similar issue was explained in response to Defect Report #017 Question 7, regarding `size_t`. The relevant citation here is from subclause 6.1.3.4, page 29, lines 36-37:

> A wide character constant has type `wchar_t`, an integral type defined in the `<stddef.h>` header.

The intent of this sentence is to note that a wide character constant has an integral type. That integral type is the same integral type used to define `wchar_t` in the header `<stddef.h>`. The sentence imposes no requirement that this particular definition of `wchar_t` be in scope wherever you write a wide character constant. It certainly does not suggest that the translator should honor any other definition of `wchar_t` that may be in scope, as the type for a wide character constant.

Rather, the sentence suggests that the translator knows what integral type to assign to a wide character constant. The implementation further knows to define `wchar_t` within the header `<stddef.h>` as having that same integral type. Thus, the program has a way of obtaining a name for this type, if it chooses — by including the header `<stddef.h>`. But it need not invoke that mechanism just to assist the translator.

It is an unfortunate, but widespread, practice within the C Standard to use abbreviated language for describing some types. Thus, subclause 6.1.4, page 31, lines 5-6 say:

> for wide string literals, the array elements have type `wchar_t`, ...

instead of the more long winded (but clearer):

> for wide string literals, the array elements have the same type used to define `wchar_t` in the header `<stddef.h>`, ...

We feel the usage is sufficiently uniform that the meaning intended by the Committee is sufficiently clear, even as we acknowledge that the words can be (and have been) misread.

So to put the matter crassly, the translator *does* "just hope for the best and take a type that makes it feel good," as you conjectured.

Defect Report #051

Submission Date: 08 Mar 93
Submittor: Project Editor (P.J. Plauger)
Source: Andrew R. Koenig

Question 1

I'll give you the short form first. I can haul out lots of related material if it becomes necessary, but perhaps the bare question is enough. Is the following program strictly conforming?

```
#include <stdlib.h>

struct A {
      char x[1];
      };

main()
      {
      struct A *p = (struct A *) malloc(sizeof(struct A) + 100);
      p->x[5] = '?';       /* This is the key line */
      return 0;
      }
```

If I remember correctly from reading the C Standard, pointer arithmetic is illegal if it results in an address outside the object to which the original pointer refers. The question here is essentially whether the "object" is all the memory returned by `malloc` or the single `char` denoted by `p->x[0]`.

I do not believe there is any language in the C Standard that clearly answers this question. I understand that this particular programming technique is quite common, but that is more likely to affect whether a program is "conforming" than whether it is "strictly conforming."

Response

Subclause 6.3.2.1 describes limitations on pointer arithmetic, in connection with array subscripting. (See also subclause 6.3.6.) Basically, it permits an implementation to tailor how it *represents pointers* to the size of the objects they point at. Thus, the expression `p->x[5]` may fail to designate the expected byte, even though the `malloc` call ensures that the byte is present. The idiom, while common, is *not* strictly conforming.

A safer idiom is:

```
#include <stdlib.h>
#define HUGE_ARR    10000      /* largest desired array */

struct A {
      char x[HUGE_ARR];
      };

main()
      {
      struct A *p = (struct A *) malloc(sizeof(struct A)
            - HUGE_ARR + 100);  /* want x[100] this time */
      p->x[5] = '?';       /* now strictly conforming */
      return 0;
      }
```

Defect Report #052

Submission Date: 21 Mar 93
Submittor: Project Editor (P.J. Plauger)
Source: Paul Edwards

Question 1

In subclause 7.12.2.3, page 172, the example is not strictly conforming. The `mktime` return is compared against `-1` instead of `(time_t)-1`, which could cause a problem with a strictly conforming implementation.

Correction

In subclause 7.12.2.3, page 172, line 16, change:

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

to:

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

Question 2

Index entry for `static` lists subclause 3.1.2.2 instead of subclause 6.1.2.2.

Correction

In the index, page 217, 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

Question 3

Footnote 1, page 1, says that the C Standard comes with a Rationale; it doesn't.

Response

The footnote actually states, in part, "It is accompanied by a Rationale document that explains ..." And indeed, the C Standard was accompanied by such a document throughout its approval process. ISO, unfortunately, has elected not to distribute the Rationale with the C Standard. "Accompanied by" does not promise "comes with" when you buy the C Standard.

Defect Report #053

Submission Date: 25 Mar 93
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

There's been a discussion on `comp.std.c` recently about accessing a pointer to a function with parameter type information through a pointer to a pointer to a function without parameter type information. For example:

```
int f(int);
int (*fp1)(int);
int (*fp2)();
int (**fpp)();

fp1 = f;
fp2 = fp1;      /* pointers to compatible types, assignment ok */
(*fp2)(3);      /* function types are compatible, call is ok */
fpp = &fp1;     /* pointer to compatible types, assignment ok */
(**fpp)(3);     /* valid? */
```

The final call itself should be valid since the resulting function type is compatible with the type of the function being called, but there's still a problem: Subclause 6.3 **Expressions**, page 38, says:

> 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,
>
> — a type that is the signed or unsigned type corresponding to the declared type of the object,
>
> — a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
>
> — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
>
> — a character type.
>
> [Footnote 36: The intent of this list is to specify those circumstances in which an object may or may not be aliased.]

This would appear to render the final call undefined since the stored value of `fp1` is being accessed by an lvalue that does not match its declared type: `(int (*)())` vs. `(int (*)(int))`.

I think that this example should be valid and that the above limitation is too strict. I think what we meant to say was "*a type compatible with* the declared type of the object," which would allow "reasonable" type mismatches without allowing aliasing between wildly different types.

Correction

In subclause 6.3, page 38, 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,

Defect Report #054

Submission Date: 01 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

Are the string handling functions defined in subclause 7.11 that have an explicit length specification (`memcpy`, `memmove`, `strncpy`, `strncat`, `memcmp`, `strncmp`, `strxfrm`, `memchr`, and `memset`) well-defined when the length is specified as zero?

Taking `memcpy` as an example, the description in subclause 7.11.2.1 states:

> The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

The response to Defect Report #042 Question 1 indicates that:

> ... the "objects" referred to by subclause 7.11.2.1 are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of `N` bytes in length (i.e. treated as an array of `N` elements of character type).

Since, by definition, objects consist of at least one byte, this would imply that `s1` and `s2` are not pointing to objects when `N` is zero and thus are outside the domain of the function leading to undefined behavior.

I do not recall whether this was the Committee's intent or not, but it would seem that some clarification is in order.

Correction

Add to subclause 7.11.1, page 162:

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.

Defect Report #055

Submission Date: 14 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Loren Schall

Question 1

It has been suggested that the six macros **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** must have distinct values. Here is the relevant portion of subclause 7.7:

"The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to and the return value of the **signal** function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

...

An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function."

On the one hand, the reference to "the signal number corresponding to the specified condition" might be assumed to imply different numbers for each signal. On the other hand, the words "distinct values" were explicitly used for the three **SIG_*** macros and are conspicuously missing for the others.

Also, I think it's worth noting that the standard expects **raise** to work meaningfully (i.e. to be able to tell them apart).

Summary: must **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** have distinct values?

Correction

In subclause 7.7, page 120, 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:

Defect Report #056

Submission Date: 15 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Thomas Plum

Question 1

The following requirement is implied in several places, but not explicitly stated. It should be explicitly affirmed, or alternative wording adopted.

The representation of floating-point values (such as floating-point constants, the results of floating-point expressions, and floating-point values returned by library functions) shall be accurate to one unit in the last position, as defined in the implementation's `<float.h>` header.

Discussion: The values in `<float.h>` aren't required to document the underlying bitwise representations. If you want to know how many bits, or bytes, a floating-point values occupies, use `sizeof`. The `<float.h>` values document the mathematical properties of the representation, the behaviors that the programmer can count upon in analyzing algorithms.

It is a quality-of-implementation question as to whether the implementation delivers accurate bits throughout the bitwise representation, or alternatively, delivers considerably less accuracy. The point being clarified is that `<float.h>` documents the delivered precision, not the theoretically possible precision.

Open Issue

Defect Report #057

Submission Date: 07 Jun 93
Submittor: Project Editor (P.J. Plauger)
Source: Fred Tydeman

Question 1

Must there exist a user-accessible integral type for every pointer? If an implementation provides 48-bit pointers, must there be an integral type, such as `long` or `int`, that is at least 48 bits? Parts of the C Standard that may help answer the question follow:

Subclause 6.3.4, **Cast operators**, page 45, lines 30-34 and Footnote 45:

> A pointer may be converted to an integral type. The size of integer required and the result are implementation-defined. If the space provided is not long enough, the behavior is undefined.
>
> An arbitrary integer may be converted to a pointer. The result is implementation-defined.[45] [Footnote 45: The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.]

Response

Integral types and pointer types are incommensurate. An implementation need not provide an integral type that can accept the conversion from a pointer type without loss of information.

Defect Report #058

Submission Date: 07 Jun 93
Submittor: Project Editor (P.J. Plauger)
Source: Fred Tydeman

Question 1

What is the minimum value for the maximum number of digits in a number that can be processed by the **`scanf`** family and the **`strtod`** family?

1) 509
2) 32767
3) something else

Parts of the the C Standard that may help answer the question follow. Subclause 7.9.6.1**The `fprintf` function**, page 134, lines 16-18:

> **Environmental limit**
>
> The minimum value for the maximum number of characters produced by any single conversion shall be 509.

But, note, there is no such environmental limit for **`fscanf`**.

Subclause 5.2.4.1 **Translation limits**, page 13, line 17:

> — 509 characters in a logical source line

But, note, there is no execution limit. Subclause 5.2.4.1 **Translation Limits**, page 13, line 19:

> — 32767 bytes in an object (in a hosted environment only)

Consider the number 1.0 written as **`.00000...00001e32759`** that is 32767 characters long (including terminator). There is only one significant digit, the **`1`**. It can be stored in an array of 32767 characters, so it should be possible to pass this string to **`atof`**, **`strtod`**, or **`sscanf`** and get the value 1.0. Correct?

Response

You are correct; the C Standard imposes no execution limit on the maximum number of digits in the subject sequence of **`fscanf`** conversion specifiers and the **`strto*`** functions, other than on the size of objects.

Defect Report #059

Submission Date: 15 Jun 93
Submittor: Project Editor (P.J. Plauger)
Source: Martin Ruckert

Question 1

The ISO Standard for the programming language C explains the notion of *incomplete type* in subclause 6.1.2.5 and subclause 6.5.2.3 (for structures). Both sections do not explicitly require that an incomplete type eventually must be completed, nor do they explicitly allow incomplete types to remain incomplete for the whole compilation unit.

Since this feature is of importance for the declaration of true opaque data types, it deserves clarification. I propose to add to the fourth paragraph on page 24 (subclause 6.1.2.5) the sentence: "It is admissable that an incomplete type remain incomplete in the whole compilation unit."

The type `void` is already an incomplete type which is never completed.

The examples given in the standard document explain that incomplete types are exclusively needed to define mutual referential structures. Opaque data types, however, constitute a second use for this feature. Considering mutual referential structures defined and implemented in different compilation units makes the idea of an opaque data type a natural extension of an incomplete data type.

Response

The C Standard intentionally contains no prohibition against leaving a type incomplete. (As you so aptly observe, `void` is an incomplete type that is never completed.) There is no need to make a positive statement about the absence of a prohibition.

Moreover, the examples were not intended to represent that mutual referencing was the only reason for permitting incomplete structure types. Opaque data types were considered, and endorsed, by the Committee when drafting the C Standard.

Summary of Issues

The list that follows provides a brief summary of all issues raised as separate questions within Defect Reports #001 through #059. Please note that the one-sentence summaries that follow seldom do justice to the issues, which are often subtle or complex. Read them to get a sense of the area of the C Standard requiring interpretation or correction. Be warned that they may well fail to properly characterize the precise concern.

#001 10 Dec 92 X3J11/90-009 (Paul Eggert)

Q1: Do functions return values by copying?

#002 10 Dec 92 X3J11/90-010 (Terence David Carroll)

Q1: Should \ be escaped within macro actual parameters?

#003 10 Dec 92 X3J11/90-011 (Terence David Carroll)

Q1: Are preprocessing numbers too inclusive?

Q2: Should white space surround macro substitutions?

Q3: Is an empty macro argument a constraint violation?

Q4: Should preprocessing directives be permitted within macro invocations?

#004 10 Dec 92 X3J11/90-012 (Paul Eggert)

Q1: Are multiple definitions of unused identifiers with external linkage permitted?

#005 10 Dec 92 X3J11/90-020 (Walter J. Murray)

Q1: May a conforming implementation support a pragma which changes the semantics of the language?

#006 10 Dec 92 X3J11/90-020 (Walter J. Murray)

Q1: How does `strtoul` behave when presented with a subject sequence that begins with a minus sign?

#007 10 Dec 92 X3J11/90-043 (Paul Eggert)

Q1: Are declarations of the form `struct tag` permitted after `tag` has already been declared?

#008 10 Dec 92 X3J11/90-021 (Otto R. Newman)

Q1: Is dead-store elimination permitted near `setjmp`?

Q2: Should volatile-qualified functions be added?

#009 10 Dec 92 X3J11/90-023 (Bruce Blodgett)

Q1: Are typedef names sometimes ambiguous in parameter declarations?

#010 10 Dec 92 X3J1190-044 (Michael S. Ball)

Q1: Is typedef to an incomplete type valid?

#011 10 Dec 92 X3J11/90-008 (Rich Peterson)

Q1: When do the types of multiple external declarations get formed into a composite type?

Q2: Does `extern` link to a static declaration that is not visible?

Q3: Are implicit initializers for tentative array definitions syntactically valid?

Q4: Does an incomplete array get completed as a tentative definition?

#012 10 Dec 92 X3J11/90-046 (David F. Prosser)

Q1: Can one take the address of a void expression?

#013 10 Dec 92 X3J11/90-047 (Sam Kendall)

Q1: How does one form the composite type of mixed array and pointer parameter types?

Q2: Is compatability properly defined for recursive types?

Q3: What is the composite type of an enumeration and an integer?

Q4: When is a structure type complete?

Q5: When is the size of an enumeration type known?

#014 10 Dec 92 X3J11/90-049 (Max K. Goff)

Q1: Is `setjmp` a macro or a function?

Q2: How does `fscanf("%n")` behave on end-of-file?

#015 10 Dec 92 X3J11/90-051 (Craig Blitz)

Q1: How does an unsigned plain bit-field promote?

#016 10 Dec 92 X3J11/90-052 (Sam Kendall)

Q1: Can a tentative definition have an incomplete type initially?

Q2: Can you implicitly initialize a union when null pointers have nonzero bit patterns?

#017 10 Dec 92 X3J11/90-056 (Derek M. Jones)

Q1: Are newlines permitted within macro invocations in preprocessing directives?

Q2: Should the absence of function `main` be explicitly undefined?

Q3: Does a constraint violation win over undefined behavior?

Q4: Do numeric escape sequences map from source to execution character sets?

Q5: When are character constants implementation-defined?

Q6: Is taking the address of a register aggregate member a constraint error?

Q7: What is the scope and uniqueness of `size_t`?

Q8: What types are compatible with pointer to `void`?

Q9: What is the type of an assignment expression?

Q10: When is the size of an object needed?

Q11: Is `struct t; struct t;` valid?

Q12: How do typedefs parse in function prototypes?

Q13: How does `register` affect compatibility of function parameters?

Q14: Can `void` parameter have storage class or type qualifier?

Q15: What kinds of array parameter declarations are compatible?

Q16: Can a matrix element be accessed via a pointer to a different row?

Q17: How do you initialize the first member of a union if it has no name?

Q18: Are `f()` and `f(void)` compatible?

Q19: Are macro expansions ambiguous in a borderline case?

Q20: Is the scope of macro parameters defined in the right place?

Q21: Is translation phase 4 defined unambiguously?

Q22: Does the rescan of a macro invocation also perform token pasting?

Q23: How long does "blue paint" persist on macro names?

Q24: Can subclause 7.1.2 be better expressed?

Q25: Should "must" appear in footnotes?

Q26: Are unnamed union members required to be initialized?

Q27: Does the # flag alter zero stripping of `%g` in `fprintf`?

Q28: Does `errno` get stored before library functions return?

Q29: When does conversion failure occur in floating-point `fscanf` input?

Q30: Do `fseek/fsetpos` require values from successful calls to `ftell/fgetpos`?

Q31: Are object sizes always in bytes?

Q32: Are `strcmp/strncmp` defined when `char` is signed?

Q33: Are `strcmp/strncmp` defined for strings of differing length?

Q34: Is `strtok` described properly?

Q35: When is a physical source line created?

Q36: Is a function returning `const void` defined?

Q37: What is the type of a function call?

Q38: What is an iteration control structure or selection control structure?

Q39: Are header names tokens outside `#include` directives?

#018 10 Dec 92 X3J11/90-066 (Yasushi Nakahara)

Q1: Does `fscanf` recognize literal multibyte characters properly?

#019 10 Dec 92 X3J11/91-014 (Richard Wiersma)

Q1: Are printing characters implementation defined?

#020 10 Dec 92 X3J11/91-006 (Bruce Lambert)

Q1: Is the Relaxed Ref/Def linkage model conforming?

#021 10 Dec 92 X3J11/91-001 (Fred Tydeman)

Q1: What is the result of `printf("%#.4o", 345)`?

#022 10 Dec 92 X3J11/91-002 (Fred Tydeman)

Q1: What is the result of `strtod("100ergs", &ptr)`?

#023 10 Dec 92 X3J11/91-003 (Fred Tydeman)

Q1: If `99999 > DBL_MAX_10_EXP`, what is the result of `strtod("0.0e99999", &ptr)`?

#024 10 Dec 92 X3J11/91-004 (Fred Tydeman)

Q1: For `strtod`, what does `"C"` locale mean?

#025 10 Dec 92 X3J11/91-005 (Fred Tydeman)

Q1: What is meant by "representable floating-point value?"

#026 10 Dec 92 X3J11/91-007 (Randall Meyers)

Q1: Can one use other than the basic C character set in a strictly conforming program?

#027 10 Dec 92 X3J11/91-008 (Randall Meyers)

Q1: May a standard conforming implementation add identifier characters?

#028 10 Dec 92 X3J11/91-009 (Randall Meyers)

Q1: What are the aliasing rules for dynamically allocated objects?

#029 10 Dec 92 X3J11/91-016 (Sam Kendall)

Q1: Must compatible structures have the same tag in different translation units?

#030 10 Dec 92 X3J11/91-017 (Pawel Molenda)

Q1: May `sin(DBL_MAX)` set `errno` to `EDOM`?

#031 10 Dec 92 X3J11/91-018 (Pawel Molenda)

Q1: How are exceptions handled in constant expressions?

#032 10 Dec 92 X3J11/91-036 (Stephen D. Clamage)

Q1: Can an implementation permit a comma operator in a constant expression?

#033 10 Dec 92 X3J11/91-037 (Mike Vermeulen)

Q1: Must a conforming implementation diagnose "shall" violations outside **Constraints**?

#034 10 Dec 92 X3J11/91-038 (Stephen D. Clamage)

Q1: Is size information lost when a declaration goes out of scope, for objects with external linkage?

Q2: If so, can one then write conflicting declarations in disjoint scopes?

#035 10 Dec 92 X3J11/91-039 (Derek M. Jones)

Q1: Can one declare an enumeration or structure tag as part of an old-style parameter declaration?

Q2: If so, what is the scope of enumeration tags and constants declared in old-style parameter declarations?

#036 10 Dec 92 X3J11/91-040 (Fred Tydeman)

Q1: May a floating-point constant be represented with more precision than implied by its type?

#037 10 Dec 92 X3J11/91-043 (Isai Scheinberg)

Q1: Can UNICODE or ISO 10646 be used as a multibyte code?

#038 10 Dec 92 X3J11/91-046 (Kuo-Wei Lee)

Q1: What happens when macro replacement creates adjacent tokens that can be taken as a single token?

#039 10 Dec 92 X3J11/91-061 (Vania Joloboff)

Q1: Must `MB_CUR_MAX` be one in the `"C"` locale?

Q2: Should `setlocale(LC_ALL, NULL)` return `"C"` in the `"C"` locale?

#040 10 Dec 92 X3J11/91-062 (Derek M. Jones)

Q1: What is the composite type of `f(int)` and `f(const int)`?

Q2: Is an implementation that fails to equal the value of a library environmental limit conforming?

Q3: Does violation of an "environmental constraint" require a diagnostic?

Q4: Should the response to Defect Report #017 Q39 be reconsidered?

Q5: Can a conforming implementation accept `long long`?

Q6: Can one use `offsetof(struct t1, mbr)` before `struct t1` is completely defined?

Q7: Can `sizeof` be applied to earlier parameter names in a prototype, or to earlier fields in a struct?

Q8: What arithmetic can be performed on a `char` holding a defined character literal value?

Q9: Should the response to Defect Report #017 Q27 be reconsidered?

#041 10 Dec 92 X3J11/91-076 (Andrew Josey)

Q1: Are `'A'` through `'Z'` always `isupper` in all locales?

#042 10 Dec 92 X3J11/92-001 (Tom MacDonald)

Q1: Does `memcpy` define a (sub)object?

Q2: If so, how big is the object defined by `memcpy`?

Q3: How big is a string object defined by the `str*` functions?

#043 10 Dec 92 X3J11/92-004 (Robert Paul Corbett)

Q1: Can `NULL` be defined as `4-4`?

Q2: Can a macro that starts with an underscore be defined if a standard header is included?

#044 10 Dec 92 X3J11/92-010 (Steve M. Hoxey)

Q1: What does it mean to say that the type of `offsetof` is `size_t`?

Q2: Must the expansion of a standard header be a strictly conforming program?

Q3: Can expanding `offsetof` result in a non-strictly conforming program?

Q4: Can one use `offsetof` in a strictly conforming program?

Q5: How can`offsetof` be reconciled with the requirements for strictly conforming programs?

#045 10 Dec 92 X3J11/92-036 (David J. Hendricksen)

Q1: Can one `freopen` an already closed file?

#046 10 Dec 92 X3J11/92-041 (Neal Weidenhofer)

Q1: May a typedef be redeclared as a parameter in a new-style function parameter type list?

#047 10 Dec 92 X3J11/92-040 (Randall Meyers)

Q1: Can an array parameter have elements of incomplete type?

#048 10 Dec 92 X3J11/92-043 (David F. Prosser)

Q1: Is `abort` compatible with POSIX?

#049 10 Jan 93 David Metsky

Q1: Can `strxfrm` produce a longer translation string?

#050 24 Feb 93 C. Breeus

Q1: Does a proper definition of `wchar_t` need to be in scope to write a wide-character literal?

#051 08 Mar 93 Andrew R. Koenig

Q1: Can one index beyond the declared end of an array if space is allocated for the extra elements?

#052 21 Mar 93 Paul Edwards

Q1: Should the `mktime` example use `(time_t)-1` instead of `-1`?

Q2: Is the index entry for `static` correct?

Q3: Does the ISO C Standard come with a Rationale, as indicated in Footnote 1?

#053 25 Mar 93 Larry Jones

Q1: Do the aliasing rules cover accesses to compatible types properly?

#054 01 Apr 93 Larry Jones

Q1: What is the behavior of various string functions with a specified length of zero?

#055 14 Apr 93 Loren Schall

Q1: Must the `SIG*` macros have distinct values?

#056 15 Apr 93 Thomas Plum

Q1: How accurate must floating-point arithmetic be?

#057 07 Jun 93 Fred Tydeman

Q1: Must there exist a user-accessible integral type for every pointer?

#058 07 Jun 93 Fred Tydeman

Q1: What is the number of digits that can be processed by the `scanf` and `strtod` families?

#059 15 Jun 93 Martin Ruckert

Q1: Must an incomplete type be completed by the end of a translation unit?

#060 19 Jul 93 Larry Jones

Q1: Does a short string literal initialize an entire array?

#061 19 Aug 93 Ed Bendickson

Q1: Can a white-space directive in `fscanf` match zero input bytes?

#062 19 Aug 93 David J. Hendrickson

Q1: Can `rename` always fail if it must copy the file?