W614/N377

X3J11/94-062

Defect Report #060

Submission Date: 19 Jul 93
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

When an array of `char` (or `wchar_t`) is initialized with a string literal that contains fewer characters than the array, are the remaining elements of the array initialized?

Subclause 6.5.7 **Initialization**, page 72, only says (emphasis mine):

> If there are fewer initializers in a *brace-enclosed list* than there are members of an aggregate, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

Correction

In the penultimate paragraph of subclause 6.5.7, add after the comma:

or fewer characters in a string literal or wide string literal used to initialize an array of known size,

Defect Report #061

Submission Date: 19 Aug 93
Submittor: X3 Secretariat (USA)
Source: Ed Bendickson

Question 1

I am requesting an interpretation of white space in the format string of a scan statement. One of our customers is concerned about this as it appears to conflict with some books on C. I am referring to subclause 7.9.6.2, page 135, paragraph 3:

> A directive composed of white space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

Page 135, paragraph 7 says:

> If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

My questions are:

1) Is white space in the format string a directive which must be satisfied by white space in the input string?

2) What are the correct answers to the following examples? Note the white space in the format string.

Example 1:

```
inputString = "123ABCD";
numAssigned = sscanf(inputString, "%lu %ls", &ulongVal, junkchar);
```

Should the result be `numAssigned` equal to 1?

Example 2:

```
inputString = "123ABCD";
numAssigned = sscanf(inputString, "%lu%ls", &ulongVal, junkchar);
```

Should the result be `numAssigned` equal to 2?

Response

A directive composed of white-space character(s) can successfully match zero white-space characters in the input stream. The paragraphs that intervene between your two quotations make clear that the second paragraph applies only to a directive that is a conversion specification.

Thus, both examples should assign 2 to `numAssigned`.

Defect Report #062

Submission Date: 19 Aug 93
Submittor: X3 Secretariat (USA)
Source: David J. Hendricksen

Question 1

If the only way to effectuate the renaming of a file on a given system is to copy the contents of the file, does an implementation conform to the C Standard by always returning a failure from the `rename` function? Footnote 113 would seem to imply this.

Response

Yes, subclause 7.9.4.2 permits the `rename` function to fail if it must copy the file contents, among other reasons.

149

Defect Report #063

Submission Date: 01 Dec 93
Submittor: Project Editor (P.J. Plauger)
Source: Thomas Plum

Question 1

[This is Defect Report #056, resubmitted for administrative reasons.]

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.

Response

The C Standard imposes no requirement on the accuracy of floating-point arithmetic.

Further discussion:

The C Standard speaks directly to the matter of floating-point accuracy only in one or two areas. Subclause 6.2.1.4 **Floating types**, page 35, says of conversions from one floating type to one with less range and/or precision:

> 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.

And in subclause 6.2.1.5 **Usual arithmetic conversions**, page 35:

> 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.

Otherwise, arithmetic for both integer and floating types is defined in terms of the usual terminology of mathematics. Nothing in the C Standard suggests that floating arithmetic is excused from the conventional rules of arithmetic.

Nevertheless, it is commonplace for the functions declared in `<math.h>` to deliver results less accurate than the underlying representation can support. It is not uncommon even for simple arithmetic expressions to do the same. And still, implementations document in `<float.h>` properties of the underlying *representation*, not the effective range and precision reliably delivered. The C community has typically tolerated a certain laxity in this area.

Probably the most useful response would be to amend the C Standard by adding two requirements on implementations:

Require that an implementation document the maximum errors it permits in arithmetic operations and in evaluating math functions. These should be expressed in terms of "units in the least-significant position" (ULP) or "lost bits of precision."

Establish an upper bound for these errors that all implementations must adhere to.

The state of the art, as the Committee understands it, is:

correctly rounded results for arithmetic operations (no loss of precision)

1 ULP for functions such as `sqrt`, `sin`, and `cos` (loss of 1 bit of precision)

4-6 ULP (loss of 2-3 bits of precision) for other math functions.

150

Since not all commercially viable machines and implementations meet these exacting requirements, the C Standard should be somewhat more liberal.

The Committee would, however, suggest a requirement no more liberal than a loss of 3 bits of precision, out of kindness to users. An implementation with worse performance can always conform by providing a more conservative version of `<float.h>`, even if that is not a desirable approach in the general case.

The Committee should revisit this issue during the revision of the C Standard.

Defect Report #064

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 1 — Null pointer constants

Consider the following translation unit:

```
char *f1 (int i, int *pi)
        {
        *pi = i;
        return 0;
        }

char *f2 (int i, int *pi)
        {
        return (*pi = i, 0);
        }
```

In `f1`, the 0 is a null pointer constant (subclause 6.2.2.3). Since `return` acts as if by assignment (subclause 6.6.6.4) the function is strictly conforming.

In `f2`, the 0 is a null pointer constant. However, a constant expression cannot contain a comma operator (subclause 6.4), and so the expression being returned is not a null pointer constant per se. Which of the following is the case?

1) The property of being a null pointer constant percolates upwards through an expression, and the function `f2` is strictly conforming.

2) The property of being a null pointer constant does not percolate upwards, and the expression being notionally assigned in the `return` statement, though of value zero, is not a null pointer constant but only of type `int`, thus violating a constraint (subclause 6.3.16.1).

Response

Function `f2` is not strictly conforming, because it violates a constraint for simple assignment (which applies to converting the type of the `return` expression), because the `return` expression is not a null pointer constant.

152

Defect Report #065

Submission Date: 03 Dec 93

Submittor: WG14

Source: Clive Feather

Question 1

Item 2 — locales

Consider the program:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main (void)
    {
    int i;
    char *loc [] = { "English", "En_UK", "Loglan", "" };

    for (i = 0; ; i++)
        if (setlocale (LC_ALL, loc [i]) != NULL)
            {
            /*
             *  We must eventually get here,
             *  because setlocale("") can't yield NULL.
             */
            printf ("Decimal point = '%s'\n",
            localeconv ()->decimal_point);
            exit (0);
            }
    }
```

The valid locales are implementation-defined (subclause 7.4.1.1). Nevertheless, the output produced depends only on the locale, not any other implementation-defined behaviour. Is the program strictly conforming?

Response

The Committee affirms that the intent of this wording is that a program such as that above, whose output varies only according to the locale selected and does not rely on the presence of a specific locale other than the "C" locale or that selected by "", was always intended to be strictly conforming. Nevertheless, it is agreed that a strict reading of the cited extract from subclause 7.4.1.1 could be read as making such programs depend on implementation-defined behavior.

The Committee reaffirms that programs that depend on the identity of the available locales, as opposed to their contents, are not strictly conforming.

The Committee believes that the term "implementation-defined" in the first sentence of the extract from subclause 7.4.1.1 was intended in the sense of "implementation-documented." However, the Committee is reluctant to introduce a new term, with possibly new conformance requirements, in a Technical Corrigendum. The Committee notes that the term "locale-specific," while making the sentence read somewhat awkwardly, carries the necessary requirements (the implementation must document the relevant details).

The Committee decided that, though the question only addresses one issue to do with locales, the above discussion applies to all instances where the behavior of an implementation depends on the locale. For this reason, the Committee decided to address all such issues at this time.

The Committee should revisit this issue during the revision of the C Standard.

Correction

In subclause 5.2.1.2 **Multibyte characters, change:**

wherein each sequence of multibyte characters begins in an initial shift state and enters other implementation-defined shift states

15-2

to:

wherein each sequence of multibyte characters begins in an initial shift state and enters other locale-specific shift states

***In subclause 7.3* Character handling `<ctype.h>`,** change:

Those functions that have implementation-defined aspects only when not in the C locale are noted below.

The term printing character refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device; the term control character refers to a member of an implementation-defined set of characters that are not printing characters.

to:

Those functions that have locale-specific aspects only when not in the C locale are noted below.

The term printing character refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term control character refers to a member of a locale-specific set of characters that are not printing characters.

In subclause 7.3.1.2* The `isalpha` function, *subclause 7.3.1.6* The `islower` function, *subclause 7.3.1.9* The `isspace` function, *and subclause 7.3.1.10* The `isupper` function, *change:

is one of an implementation-defined set of characters

to:

is one of a locale-specific set of characters

In subclause 7.4.1.1* The `setlocale` function, *change:

a value of `""` for `locale` specifies the implementation-defined native environment.

to:

a value of `""` for `locale` specifies the locale-specific native environment.

In subclause 7.10.1.4* The `strtod` function, *subclause 7.10.1.5* The `strtol` function, *and 7.10.1.6* The `strtoul` function, *change:

In other than the `"C"` locale, additional implementation-defined subject sequence forms may be accepted.

to:

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

Change Footnote 131 from:

If the implementation employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

to:

If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

In subclause 7.11.6.2* The `strerror` function, *change:

The `strerror` function returns a pointer to the string, the contents of which are implementation-defined.

to:

The `strerror` function returns a pointer to the string, the contents of which are locale-specific.

In Annex G, move the following bullet items from subclause G.3 to subclause G.4:

G.3.4, item 2 ("The shift states used for the encoding ...")

G.3.14, item 3 ("The sets of characters tested for ...")

G.3.14, item 33 ("The contents of the error message strings ...")

In Annex G.4* Locale-specific behaviour, *change:

The following characteristics of a hosted environment are locale-specific:

to:

The following characteristics of a hosted environment are locale-specific and must be documented by the implementation:

Defect Report #066

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 3 — locales

In a conforming implementation, can the value of any of the following expressions (subclause 7.4.2.1) be a value other than 0 or 1? Can the value of the first expression be 0?

```
strlen(localeconv()->decimal_point)
strlen(localeconv()->thousands_sep)
strlen(localeconv()->mon_decimal_point)
strlen(localeconv()->mon_thousands_sep)
```

If the value can be greater than 1, can the string contain more than one multibyte character? If so, can the string contain shift sequences? If so, can the string end other than in the initial shift state?

Response

Of the four `strlen` calls, the first must return 1, the second must return 0 or 1, and the other two must return 0 or more, in a conforming implementation. There is a specific requirement for `decimal_point` in the second paragraph of subclause 7.4.2.1 **Description**, and in the individual descriptions "character" is intended to imply 0 or 1 while "string" is meant to imply 0 or more.

Defect Report #067

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 4 — definitions of types

The terms "signed integer type," "unsigned integer type," and "integral type" are defined in 6.1.2.5. The C Standard also uses the terms "integer type," "signed integral type," and "unsigned integral type" without defining them. Integer-valued bitfields are also introduced in 6.5.2.

a) For each of the following types, which if any of the six categories above do they belong to?

```
char
signed char
unsigned char
signed short
unsigned short
signed int
unsigned int
signed long
unsigned long
int : N     /* i.e. bitfield of size N */
signed int : N
unsigned int : N
enumerated type
```

b) For each of these categories, do the **`const`** and/or **`volatile`** qualified versions of the types belonging to the category also belong to the category?

c) Can an implementation extension add other types defined by the C Standard to any of these six categories?

d) Can an implementation define other types (e.g. **`__very long`**) which belong to any of these six categories?

e) If the answer to (c) or (d), or both, is yes, can **`size_t`** and **`ptrdiff_t`** be one of these other types, or must it be a type in the above list?

Response

a) "signed integer type", "unsigned integer type", and plain "integer type" are used interchangeably with "signed integral type", "unsigned integral type", and "integral type" in the C Standard. This observation makes it easy to categorize the types in your list.

b) Yes, see subclause 6.1.2.5.

c) No, the list in the C Standard is meant to be exhaustive. For example, "four" signed types cannot be read as "four or more."

Conforming implementations may add other distinct types (such as **`__int24`**) to these categories, but must not use such types where a standard-specified type is required. For example, **`size_t`** cannot be defined as **`unsigned __int24`**.

d) No strictly conforming program could contain an instance of such a type. The treatment of such types is beyond the scope of the C Standard.

e) No. For example, **`size_t`** cannot be defined as **`unsigned __int24`**.

156

Defect Report #068

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 5 — handling of `char` values

Values of the type `char` must be treated as either "signed" or "nonnegative" integers (subclause 6.1.2.5).

a) Is the treatment determined strictly by the value of the expression `CHAR_MAX == SCHAR_MAX`?

b) If the treatment is as "signed" integers, does the type `char` behave in every instance as the type `signed char` (though of course being a different type)? If not, what are the differences?

c) If the treatment is as "nonnegative" integers, does the type `char` behave in every instance as the type `unsigned char` (though of course being a different type)? If not, what are the differences? In particular, do the "no overflow, reduce modulo" semantics apply?

Response

a) Yes.

b) and c) Yes. Subclause 6.2.1.1, "As discussed earlier, ..." indicates that this is the intent.

157

Defect Report #069

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 6 — representation of integral types

Subclause 6.1.2.5 refers to the representation of a value in an integral type being in a "pure binary numeration system," and defines this further in Footnote 18. On the other hand, the wording of ISO 2382 is:

05.03.15
binary (numeration) system
The *fixed radix numeration system* that uses the *bits* 0 and 1 and the *radix* two.

Example: In this *numeration system,* the numeral 110,01 represents the number "6,25"; that is $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^{-2}$.

05.03.11
fixed radix (numeration) system
fixed radix notation
A *radix numeration system* in which all the *digit places,* except perhaps the one with the highest *weight,* have the same *radix.*

NOTES

1. The weights of successive digit places are successive integral powers of a single radix, each multiplied by the same factor. Negative integral powers of the radix are used in the representation of factors.

2. A fixed radix numeration system is a particular case of a *mixed radix numeration system;* see also Note 2 to 05.03.19.

05.03.08
radix
base (deprecated in this sense)
In a *radix numeration system,* the positive *integer* by which the *weight* of any *digit place* is multiplied to obtain the weight of the digit place with the next higher weight.

Example: In the *decimal numeration system* the radix of each digit place is 10.

NOTE — The term base is deprecated in this sense because of its mathematical use (see definition in 05.02.01).

05.03.07
radix (numeration) system
dix notation
A *positional representation system* in which the ratio of the *weight* of any one *digit place* to the weight of the digit place with the next lower weight is a positive *integer.*

NOTE — The permissible values of the *character* in any digit place range from zero to one less than the *radix* of that digit place.

05.03.04
weight
In a *positional representation system,* the factor by which the value represented by a *character* in a *digit place* is multiplied to obtain its additive contribution in the representation of a number.

05.03.03
digit place
digit position
In a *positional representation system,* each site that may be occupied by a *character* and that may be identified by an ordinal number or by an equivalent identifier.

15-8

05.03.01
positional (representation) system
positional notation
Any *numeration system* in which a number is represented by an *ordered* set of *characters* in such a way that the value contributed by a character depends upon its position as well as upon its value.

a) What is the legal force of the footnote, given that it quotes a definition from a document other than ISO 2382 (see 3)?

b) Is the footnote wording correct, seeing that the ISO 2382 definition does not appear to allow any of the common representations (note the word "positive" in 05.03.07)?

c) Does the C Standard require that an implementation appear to use only one representation for each value of a given type?

d) Does the C Standard require that all the bits of the value be significant?

e) Does the C Standard require that all possible bit patterns represent numbers?

f) Do the answers to questions (c), (d), and (e) depend on whether the type is signed or unsigned, and in the former case, on the sign of the value?

g) If it is permitted for certain bit patterns not to represent values, is generation of such a value by an application (using bit operators) undefined behavior, or is use of such a value strictly conforming provided that it is not used with arithmetic operators?

In particular, are the following five implementations allowed?

h) Unsigned values are pure binary. Signed values are represented using ones-complement (in other words, positive and negative values with the same absolute value differ in all bits, and zero has two representations). Positive numbers have a sign bit of 0, and negative numbers a sign bit of 1. In both cases, all bits are significant.

i) Unsigned values are pure binary. Signed values are represented using sign-and-magnitude with a pure binary magnitude (note that the top bit is not "additive"). Positive numbers have a sign bit of 0, and negative numbers a sign bit of 1. In both cases, all bits are significant.

j) Unsigned values are pure binary, with all bits significant. Signed values with an MSB (sign bit) of 0 are positive, and the remainder of the bits are evaluated in pure binary. Signed values with an MSB of 1 are negative, and the remainder of the bits are evaluated in BCD. If ints are 20 bits, then `INT_MAX` is 524,287 and `INT_MIN` is –79,999.

k) Signed values are twos-complement using all bits. Unsigned values are pure binary, but ignoring the MSB (so each number has two representations). In this implementation, `SCHAR_MAX == UCHAR_MAX`, `SHRT_MAX == USHRT_MAX`, `INT_MAX == UINT_MAX`, and `LONG_MAX == ULONG_MAX`.

l) Signed values are twos-complement. Unsigned values are pure binary. In both cases, the top three bits of the value are ignored (and each number has eight representations). For signed values, the sign bit is the fourth bit from the top.

Furthermore:

m) Does the C Standard require that the values of `SCHAR_MAX`, `SHRT_MAX`, `INT_MAX`, and `LONG_MAX`, defined in `<limits.h>` (subclause 5.2.4.2.1) all be exactly one less than a power of 2?

n) If the answer to (m) is "yes," then must the exponent of 2 be exactly one less than `CHAR_BITS * sizeof (T)`, where `T` is `signed char`, `short`, `int`, or `long`, respectively?

p) Does the C Standard require that the values of `UCHAR_MAX`, `USHRT_MAX`, `UINT_MAX`, and `ULONG_MAX`, defined in `<limits.h>` (subclause 5.2.4.2.1) all be exactly one less than a power of 2?

q) If the answer to (p) is "yes," then must the exponent of 2 be exactly `CHAR_BITS * sizeof (T)`, where `T` is `unsigned char`, `unsigned short`, `unsigned int`, or `unsigned long` respectively?

159

r) Does the C Standard require that the absolute values of `SCHAR_MIN`, `SHRT_MIN`, `INT_MIN`, and `LONG_MIN`, defined in `<limits.h>` (subclause 5.2.4.2.1) all be exactly a power of 2 or exactly one less than a power of 2?

s) If the answer to (r) is "yes," then must the exponent of 2 be exactly one less than `CHAR_BITS * sizeof (T)`, where `T` is `signed char`, `short`, `int`, or `long` respectively?

t) If any of the answers to (m), (p), or (r) is "no," are there any values for each of these expressions that are permitted by subclause 5.2.4.2 but prohibited by the C Standard for other reasons, and if so, what are they?

u) Does the C Standard require that the expressions `(SCHAR_MIN + SCHAR_MAX)`, `(SHRT_MIN + SHRT_MAX)`, `(INT_MIN + INT_MAX)`, and `(LONG_MIN + LONG_MAX)` be exactly 0 or −1? If not, does it put any restrictions on these expressions?

Response

Before providing detailed answers, we want to provide some clarified terminology. For any object type `T`, the underlying bytes of the object can be copied into an array of `unsigned char`:

```
#define N sizeof(T)
union aligned_buf {
      T t;
      unsigned char s[N];
      } buf;
T object;
.....
      memcpy(buf.s, (const void *)&object, N);
```

The *object representation* of an object consists of the resulting sequence of `N` objects of type `unsigned char` in the buffer. The object representation depends upon several features of the implementation such as byte-ordering ("big-endian," "little-endian," etc.), "holes" (i.e., bits within a scalar object which do not participate in forming the value of the object), and "padding" (i.e., bits in a non-scalar object which lie between the component scalar objects or after the last scalar object).

The *value representation* of an object is a sequence of bits structured in a specific conventional way. The scalar components are listed in their declaration sequence. Each scalar component is a sequence of bits (the "participating bits") arranged in a conventional ordering. The value representation of floating-point and pointer types is implementation-defined. The value representation of an integer type is defined as follows: The least-significant bit (the bit which represents the integer value 1) is also called the low-order bit or rightmost bit; the most-significant bit is also called the high-order bit or leftmost bit. The sign bit (if any) is the leftmost bit.

If all the bits in a scalar object representation participate in the value representation (i.e. no holes or padding), then the value representation can be referred to simply as the *representation*. The bits of the value representation determine a *value*, which is one discrete element of an implementation-defined set of values. The conventional depiction of an integer value is as a decimal integer, optionally signed, such as 128 or −1.

Here is an example. Consider a (possibly hypothetical) ones-complement implementation whose `int` value representation provides one sign bit and 40 integer bits.

```
    +-+---------------------+
    | |                     |
    +-+---------------------+
     1          40
```

Its object representation provides one sign bit, a hole containing seven non-participating bits, and 40 integer bits (issues of byte ordering are irrelevant here):

```
    +-+------+---------------------+
    | |      |                     |
    +-+------+---------------------+
     1   7          40
```

The value representation containing 41 zero bits designates the value 0:

```
    +-+---------------------+
    |0|000      ...      000|
```

160

```
  +-+---------------------+
  1          40
```

Depending upon the implementation, the value representation containing 41 one bits may designate the same value 0, in which case it is indistinguishable from the other value representation; or it may designate a distinguishable value, conventionally depicted as –0, which is arithmetically equal to 0 but distinguishable by bitwise operations.

```
  +-+---------------------+
 |1|111      ...       111|
  +-+---------------------+
  1          40
```

Now for detailed replies:

a) Footnotes are not normative. The *legality of a footnote is beyond the scope of WG14/X3J11.*

b) Yes, the footnote is correct.

c) No, there is no such requirement.

d) In view of the discussion above, we assume you mean the following question: does the C standard require that all bits of the object representation participate in the value representation? For character types, all bits of the object representation do contribute. See 7.9.2 (re binary streams) and 7.11.1 (re string functions) for (indirect) justification. More precisely, any bits that do not contribute to the value of a character type must not contribute to the value of any other object type. (Parity bits are an obvious example.) For other types, the answer is no.

e) In view of the discussion above, we assume you mean the following question: does the C Standard require that all possible bit patterns of the value representation represent numbers? For the type `unsigned char`, the answer is yes. (And if all values of the type `char` are non-negative, then the answer is yes.) Otherwise, the answer is no.

f) No.

g) Not applicable, since it is unclear what are the meanings of "bit pattern" and "value" in the question; see the answer to part (e).

h) Yes, provided there is no other violation of the C Standard.

i) Yes, provided there is no other violation of the C Standard.

j) No. It is not a pure binary system.

k) Yes, provided there is no other violation of the C Standard.

l) Yes, provided there is no other violation of the C Standard.

m) Yes, because subclause 6.1.2.5 states that the representation of positive signed integers have the same representation as the corresponding unsigned integers, and because signed integers use a pure binary numeration system. The Committee intended to permit ones complement, twos complement, and signed magnitude implementation.

n) No. There are architectures on which not all bits can be used.

p) Yes, because subclause 6.1.2.5 requires unsigned integers to behave as if a result "is reduced modulo the number that is one greater than the largest value that can represented," and unsigned integers use a pure binary numeration system.

q) No. The memory occupied by a value of an integer is allowed to exceed the number of binary digits used to represent the actual value.

r) Yes. See the answer to part (m).

s) No. See the answer to part (q).

t) Not applicable.

u) Yes, the expression must evaluate to 0 or –1.

161

Defect Report #070

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 7 — interchangability of function arguments

Consider the following program:

```
#include <stdio.h>

void output (c)
      int c;
      {
      printf("C == %d\n", c);
      }
int main (void)
      {
      output(6);
      output(6U);
      return 0;
      }
```

The constant 6 has type `int`, and 6U has type `unsigned int` (subclause 6.1.3.2), and they have the same representation (subclause 6.1.2.5). Footnote 16, which is not a part of the C Standard, states that this implies that they are interchangable as arguments. However, `int` and `unsigned int` are not compatible types, and so 6.3.2.2 makes the second call undefined.

Is the program strictly conforming?

Note that similar issues arise in connection with the other cases mentioned in Footnote 16 (function return values and union members).

Response

The program is not strictly conforming. Since many pre-existing programs assume that objects with the same representation are interchangeable in these contexts, the C Standard encourages implementors to allow such code to work, but does not require it.

Defect Report #071

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 8 — enumerated types

The C Standard states (in effect) that an enumerated type is a set of integer constant values (subclause 6.1.2.5). It also states that an enumerated type must be compatible with an implementation-defined integer type (subclause 6.5.2.2). Finally, the integral promotions (subclause 6.2.1.1) convert an enumerated type to `signed` or `unsigned int`.

Consider:

```
enum foo { foo_A = 0, foo_B = 1, foo_C = 8 };
enum bar { bar_A = -10, bar_B = 10 };
enum qux { qux_A = UCHAR_MAX * 4, qux_B };
```

a) If any value between zero and `SCHAR_MAX` (inclusive) is assigned to a variable of type `enum foo`, and the value of the variable is then converted to type `int` or `unsigned int`, does the C Standard require the original value to result; or is the implementation permitted or required to convert it to one of the three values 0, 1, and 8; or is the result of the assignment undefined?

b) Can a conforming implementation require all enumerated types to be compatible with a single type?

c) If the answer to (b) is "yes," and assuming that the value `UCHAR_MAX * 4` is less than `SHRT_MAX` is the declaration of the type `enum qux` strictly conforming, or can a conforming implementation require all enumerated types to be compatible with a single type which is a character type?

d) Can an implementation make the type that `enum bar` is compatible with be an unsigned type, even though it uses an enumeration constant not representable in that type?

e) Can an implementation make the type that `enum qux` is compatible with be either of `signed char` or `unsigned char`, even though it uses an enumeration constant not representable in that type?

f) If the answer to (d) or (e) is "yes," what is the effect of making one of the enumeration constants of an enumerated type outside the range of the compatible type? What is the effect of assigning the value of that constant to an object of the enumerated type?

g) Can the type that an enumerated type is compatible with be `signed` or `unsigned long`? If so, what are the effects of the integral promotions on a value of that type?

h) If an implementation is allowed to add other types to the list of integer types (see items 4(b) and (c)), then can the type that an enumerated type is compatible with be such a type?

Response

a) Every enumerated type is compatible with some integer type (subclause 6.5.2.2). When conversion takes place between compatible types, values are not altered (subclause 6.2). So for values between 0 and `SCHAR_MAX`, the original value must result, because no matter what type is chosen, the value can be expressed in that type.

b) Yes it can.

c) through g) It is the intention of the C Standard that all the members of the enumeration be representable in the enumerated type, and that the compatible integer type be one which promotes to `int` or `unsigned int`.

h) An implementation is not allowed to add other types to the list. (See reply to Defect Report #067.)

Correction

In subclause 6.5.2.2, change:

Each enumerated type shall be compatible with an integer type; the choice of type is implementation-defined.

to:

153

Each enumerated type shall be compatible with an integer type. The choice of type is implementation-defined, but shall be capable of representing the values of all the members of the enumeration.

Defect Report #072

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 9 — definition of object
Consider the following translation unit:

```
#include <stdlib.h>

typedef double T;
struct hacked
      {
      int size;
      T data [1];
      };

struct hacked *f (void);
      {
      T *pt;
      struct hacked *a;
      char *pc;

      a = malloc (sizeof (struct hacked) + 20 * sizeof (T));
      if (a == NULL)
            return NULL;
      a->size = 20;

      /* Method 1 */
      a->data [8] = 42;                                         /* Line A */

      /* Method 2 */
      pt = a->data;
      pt += 8;                                                  /* Line B */
      *pt = 42;

      /* Method 3 */
      pc = (char *) a;
      pc += offsetof (struct hacked, data);
      pt = (T *) pc;                                            /* Line C */
      pt += 8;                                                  /* Line D */
      *pt = 6 * 9;
      return a;
      }
```

Now, Defect Report #051 has established that the assignment on line A involves undefined behaviour.

a) Is the addition on line B strictly conforming?

b) If the answer to (a) is "yes," are the three statements forming "method 2" a valid way of implementing the `struct hacked`?

c) Is the cast of line C strictly conforming?

d) Is the addition on line D strictly conforming?

e) If the answer to (c) and (d) are "yes," are the five statements forming "method 3" a valid way of implementing the `struct hacked`?

165

Now suppose that the definition of type `T` is changed to `char`. This means that the last bullet in subclause 6.3 ("an object shall have its stored value accessed only by ... a character type") now applies, and furthermore it means that the location accessed is an integral multiple of `sizeof(T)` bytes from the start of the `malloc`ed object, and so constitutes an element of that object when viewed as an array of `T`.

f) Is the assignment on line A now strictly conforming?

g) What are the answers to questions (a) to (e) with this change?

Response

a) Defect Report #051 provides the rationale for why Line A results in undefined behaviour. The same rules also apply to the assignment to `pt`, thus Line B results in undefined behaviour

b) Not applicable given the answer to question (a).

c) Assignment causes the base address of the structure to be assigned to `pc`. The response to Defect Report #044, question 1, states that use of the `offsetof` macro does not result in undefined behaviour. The second line causes `pc` to point at member `data`. Line C does not contain any construct that would result in the program not being strictly conforming.

d) Line D results in undefined behaviour. See answer (a) for rationale.

e) Not applicable given answers (c) and (d).

f) Subclause 6.3 contains additional restrictions, not permissions.

g) The answers to questions (a)-(e) are not affected if `T` has `char` type.

166

Defect Report #073

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 10 — definition of object

Consider the following translation unit:

```
struct complex
        {
        double real [2];
        double imag;
        }
#define D_PER_C (sizeof (struct complex) / sizeof (double))

struct complex *f (double x)
        {
        struct complex *array = malloc(sizeof(struct complex) +
                sizeof (double));
        struct complex *pc;
        double *pd;

        if (array == NULL)
                return NULL;
        array [1].real [0] = x;                              /* Line A */
        array [1].real  [1] = x;                             /* Line B */
        array [1].imag = x;                                  /* Line C */
        pc = array + 1;                                      /* Line D */
        pc = array + 2;                                      /* Line E */
        pd = &(array [1].real [0]);                          /* Line F */
        pd = &(array [1].real [1]);                          /* Line G */
        pd = &(array [1].imag);                              /* Line H */
        pd = &(array [0].real [0]) + D_PER_C;                /* Line I */
        pd = &(array [0].real [1]) + D_PER_C;                /* Line J */
        pd = &(array [0].imag) + D_PER_C;                    /* Line K */
        pd = &(array [0].real [0]) + D_PER_C * 2;            /* Line L */
        pd = &(array [0].real [0]) + D_PER_C + 1;            /* Line M */
        pd = &(array [0].real [0]) + D_PER_C + 2;            /* Line N */
        return array;
        }
```

Subscripting is strictly conforming if the array is "large enough" (subclause 6.3.6). For each of the marked lines, is the assignment strictly conforming?

Response

Lines A, B, C. The identifier `array` points at an object that is not large enough to hold two `struct complex` objects. The dot selection operator is at liberty to require the complete struct denoted by its left hand side to be accessed. Such an access would result in undefined behaviour.

Line D. If `array` is regarded as pointing to a single struct then creating a pointer to one past the end of that object is permitted.

Line E. If `array` is regarded as pointing to a single struct then creating a pointer two past the end of that object is not permitted. Since there is insufficient storage allocated to create a second struct object it is not permitted to point one past this partial struct object.

Lines F, G, H. Same analysis as Lines A, B, C.

Lines I, J, K, L, M, N. All of these calculations will result in pointers that point outside the original object (arrays or structs) and result in undefined behaviour.

167

Defect Report #074

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 11 — alignment and structure padding

The existence of structure padding (subclause 6.5.2.1) can be detected by a strictly conforming program by use of the `sizeof` operator and the `offsetof` macro.

a) If a structure has a field of type `t`, can the alignment requirements of the field be different from the alignment requirements of objects of the same type that are not members of structures?

If the answer to (a) is "yes," then where applicable the remaining questions should be assumed to have been asked for both objects within structures and objects outside structures.

b) If an array has a component type of `t`, can the alignment requirements of the elements of the array be different from those of independent variables of type `t`?

The alignment requirement of a type is that addresses of objects of that type must be multiples of some constant (subclause 3.1); for some type `t`, this is written `A(t)` in this Defect Report.

c) For any type `t`, can the expression `sizeof(t) % A(t)` be non-zero (in other words, can `A(t)` be a value other than 1, `sizeof(t)`, or a factor of `sizeof (t)`)? It would appear not, because otherwise adjacent elements of an array of objects of type `t` would either not be correctly aligned, or else would not be contiguously allocated.

d) Can `A(struct foo)` be greater than the least common multiple of `A(type_1)`, `A(type_2)`, ..., `A(type_n)`, where `type_1` to `type_n` are the types of the elements of `struct foo`? In particular, if a structure holds exactly one element, can `A(structure type)` be different from `A(element type)`? (In each case, if the answer to (a) is "yes," `A(type)` should be interpreted appropriately.)

e) If, at any point in a structure or union (obviously excluding the start), there is more than one size of padding that can satisfy all alignment requirements, can any size be used, or must the smallest (possibly zero) padding be used because that is all that is "necessary to achieve the appropriate alignment?"

f) If a structured type has trailing padding to ensure that its use as an array element would be correctly aligned, must objects of that type which are not array elements also have the padding? If not, what is the effect of using `memcpy` to copy the value of one such object to another thus?

```
struct fred a, b;
     /* ... */
     memcpy(&a, &b, sizeof (struct fred));
```

It appears from subclause 6.3.3.4 ("the size is determined from the type of the operand") that `sizeof a` must equal `sizeof (struct fred)`. Is this corr[illegible]?

g) When an element of a structure is in turn a structure, can trailing padding of the inner structure be reused to hold other elements of the enclosing structure? For example, in:

```
struct outer
     {
     struct inner { long a; char b; } inner;
     char c;
     };
```

is it permitted for `offsetof (struct outer, c)` to be less than `sizeof (struct inner)`?

Response

Subclause 6.1.2.5 says, "... pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements."

Sublause 6.5.2.1 says, "Each non-bit-field member of a structure or union object is aligned in an implementation defined manner appropriate to its type." And later, "There may therefore be unnammed holes within a structure object ... as necessary to achieve the appropriate alignment."

162

a) It is possible for an implementation to state generalized requirements to satisfy sublause 6.1.2.5. These requirements may be further strengthened using the implementation defined behavior made available in subclause 6.5.2.1. Yes, the alignment requirements can be different.

b) In several places the C Standard states that a single object may be treated as an array of one element. Nowhere does it give permission for array element types to have different alignment requirements from isolated object types.

c) We agree with the answer given to this question.

d) Yes. A structure object can have an alignment that is greater than the least common multiple of the alignments of its members.

e) The phrase "necessary to achieve the appropriate padding" is not considered to mean the use of the minimum padding possible. The Committee does not see any advantage to changing this phrase.

f) Yes. See answer to question (b). `sizeof (struct fred)` must equal `sizeof a`.

g) Such sharing of storage by objects would cause the requirements of subclause 6.3 to be violated and is not allowed.

Defect Report #075

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 12 — alignment of allocated memory

Is a piece of memory allocated by `malloc` required to be aligned suitably for any type, or only for those types that will fit into the space? For example, following the assignment:

```
void *vp = malloc (1);
```

is it required that `(void *)(int *)vp` compare equal to `vp` (assuming that `sizeof(int) > 1`), or is it permissible for `vp` to be a value not suitably aligned to point to an `int`?

Response

Subclause 7.10.3 requires allocated memory to be suitably aligned for *any* type, so they must compare equal.

Defect Report #076

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 13 — pointers to the end of arrays

Consider the following code extracts:

```
int a [10];
int *p;
/* ... */
p = &a[10];
```

and

```
int *n = NULL;
int *p
/* ... */
p = &*n;
```

In the first extract, is the assignment strictly conforming (with `p` being set to the expression `a + 10`), or is the constraint in subclause 6.3.3.2 violated because `a[10]` is not an object? Note that this expression is often seen in the idiom:

```
for (p = &a[0]; p < &a[10]; p++)
        /* ... */
```

In the second extract, is the assignment strictly conforming (with `p` being set to a null pointer), or is the constraint in 6.3.3.2 violated because `*n` is not an object?

If only one assignment is strictly conforming, what distinguishes the two cases? If either assignment is strictly conforming, what distinguishes it from the situation described in the following extract from the response to Defect Report #012?

> 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.

Response

1) Subclause 6.3.3.2 requires the operand of `&` to be an lvalue designating an object; `a[10]` is not an object.

2) Subclause 6.3.3.2 requires the operand of `&` to be an lvalue; `NULL` is not an lvalue.

Since the use of either construct prevents a program from being strictly conforming, the remaining portion of the question is not applicable.

121

Defect Report #077

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 14 — stability of addresses

Is the address of an object constant throughout its lifetime? For example, if a pointer to an object is written to a binary file using `fwrite`, and then read back later during the same run of the program using `fread`, is it guaranteed to compare equal to the address of the original object taken again?

Response

The C Standard does not explicitly state that the address of an object remains constant throughout the life of the object. That this is the intent of the Committee can be inferred from the fact that an address constant is considered to be a constant expression. The framers of the C Standard considered that it was so obvious that addresses should remain constant that they neglected to craft words to that effect.

The Committee should revisit this issue during the revision of the C Standard.

Defect Report #078

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 15 — uniqueness of addresses
Consider the following translation unit:

```
unsigned int f (unsigned int a)
     {
     unsigned int x, y;
     x = a;
     x = x * x + a;
     if (x > 100)
          return x; /* Returned value must be > 100 */
     if (&x == &y)
          return 0;
     y = a + 1;
     y = y * y + 19;
     return y;      /* Returned value must be >= 19 */
     }

unsigned int g1 (void) { return 0; };
unsigned int g2 (void) { return 0; };

unsigned int g (void)
     {
     return g1 != g2;
     }

unsigned int h (void)
     {
     return memcpy != memmove;
     }

const int j1 = 1;
const int j2 = 1;

unsigned int j (void)
     {
     return &j1 != &j2;
     }
```

a) Can **f** ever return zero? An aggressive optimizer could notice that **x** and **y** are never used at the same time, and assign them the same memory location. (The optimizer could be designed to conceal the fact that **x** and **y** are sharing storage, for example by forcing the comparison to be unequal. Such an application of the "as if" rule (subclause 5.1.2.3) would become increasingly difficult to implement in the presence of operations such as writing out **&x** to a file (using **fwrite** or the **fprintf %p** conversion specification) and then reading it back in later in the same run of the program. However, this is irrelevant; the issue is whether or not the implementation is required to conceal it in the first place.)

b) Can **g** ever return zero? A optimizer using an intermediate form can easily determine that the two functions have identical effects.

173

c) Can `h` ever return zero? The library function `memmove` (subclause 7.11.2.2) completely meets the requirements for `memcpy` (subclause 7.11.2.1) and so they could be implemented using the same code (even if the answer to (b) is no, this could happen if the system library is not implemented in C).

d) Can `j` ever return zero? Since the two variables are constants, code which uses `j1` instead of `j2` anywhere except in an address comparison cannot distinguish them.

Response

a) `f` can never return zero. There are three `return` statements:

i) Will always return a value greater than 100.

ii) `x` and `y` exist at different addresses. An optimizer may invoke the as-if rule to rearrange code provided it always achieves the required effect. (Subclause 6.1.2.2: "Identifiers with no linkage denote unique entities.")

iii) Modulo arithmetic may wrap to produce zero. On a binary arithmetic CPU it is not possible to square any number, add 19 and get zero as the result.

b) No, `g` cannot return zero.

c) Yes, `h` can return zero.

d) `j` can never return zero. Subclause 6.7.2 says, "If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition of the identifier." Subclause 6.5 says, "A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition." Taken together these two statements can be taken to imply that two file-scope definitions must refer to different objects.

174

Defect Report #079

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 16 — constancy of system library function addresses

(These questions approach the same problem from three slightly different directions.)

a) If a pointer to a given standard library function (say `strlen`) is evaluated in two different translation units, and the pointers compared, must they compare equal?

b) Can a conforming implementation declare a standard library function as having internal linkage, or must the identifiers with file scope declared in standard headers have external linkage?

c) If the contents of the header `<string.h>` include the following definition of `strlen`, is the implementation conforming?

```
static size_t strlen (const char *__s)
     {
     size_t __len = 0;

     while (*__s++)
          __len++;
     return __len;
     }
```

Response

Since the answer to question (b) is needed for question (a) it is given first.

b) Since the library function prototypes are implicitly `extern`, the standard library functions have external linkage.

a) If the usage of "strlen" is such that standard library functions are referred to, the pointers must compare equal by the requirements of subclauses 5.1.1.2 and 6.1.2.2.

c) The contents of system headers are implementation defined. For instance, they may contain code written in other languages. It is not the job of this Committee to mandate implementation. Whatever their contents, including a standard header must achieve the effects required by the C Standard.

175

Defect Report #080

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 17 — merging of string constants

Consider the following code:

```
char *s1 = "abcde" + 2;
char *s2 = "cde";
```

Can the expression `(s1 == s2)` be non-zero? Is the answer different if the first string literal is replaced by the two literals `"ab" "cde"` (because then there are identical string literals)?

Response

When the last paragraph of subclause 6.1.4 refers to "string literals" it is referring to the static arrays created in translation phase 7 as specified in the previous paragraph. Although the current wording of the C Standard may imply that only completely identical arrays need not be distinct, this was not the Committee's intent.

Correction

Change the last paragraph of subclause 6.1.4 from:

Identical string literals of either form need not be distinct. If the program attempts to modify a string literal of either form, the behavior is undefined.

to:

These arrays need not be distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

176

Defect Report #081

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 18 — left shift operator

The result of the left shift operator **`E1 << E2`**, when **`E1`** is signed, is defined (subclause 6.3.7) as **`E1`** left-shifted by **`E2`** bits, with vacated bits filled with zeros. But what exactly does this mean?

The C Standard defines a bit (subclause 3.3) only as a unit of data storage. Bits are related to the value of an object only in 6.1.2.5, which specifies the representation of certain types. It may therefore be claimed that the left shift operator must act on representations, which are of fixed length. In this interpretation, the left **`E2`** bits (including the sign bit) are lost, as they would be if **`E1`** was unsigned; the sign bit of the result is taken from a bit in **`E1`**, **`E2`** places to the right of the sign bit and, provided that the resultant bit pattern actually represents a value of the result type, an exception is impossible.

On the other hand, it may also be claimed that the whole of subclause 6.3 specifies the meaning of operations in abstract mathematical terms, subject to the general note about exceptions. In this view, the bit sequence representing the non-sign part of a signed integer is converted by the shift operation to a bit sequence of indefinite length, and, to avoid an exception due to overflow, this bit sequence must fit back in the non-sign part without the loss at the left of anything but copies of the sign bit.

a) Which of these two views is correct?

b) If the answer to (a) is the first view, does undefined behaviour occur if the resulting bit pattern is not the representation of an integer?

The following questions apply only if the answer to (a) is that the second view is correct.

c) If **`E1`** is positive, and **`E1`** times 2 to the power **`E2`** is less than or equal to **`INT_MAX`** (or **`LONG_MAX`**), is the result always **`E1`** times 2 to the power **`E2`**?

d) Under what circumstances is the result undefined?

Response

Subclause 6.3, page 38, states that the binary operator <<, among others, has implementation-defined aspects for signed types. Therefore, the answer to "what does it mean to left shift a signed value" is that it is implementation defined.

177

Defect Report #082

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 19 — multiple varargs

Consider the following translation unit:

```
#include <stdarg.h>
#include <stdio.h>

extern int is_final_arg (int);

void f1 (int n, ...)
     {
     va_list ap1, ap2;

     va_start (ap1, n);
     va_start (ap2, n);
     while (va_arg (ap1, int) != 0)
          printf ("Value is %d\n", va_arg (ap2, int));
     va_end (ap1);
     va_end (ap2);
     }

void f2 (int n, ...)
     {
     va_list ap;

     va_start (ap, n);
     for (;;)
          {
          n = va_arg (ap, int);
          if (is_final_arg (n))
               {
               va_end (ap);
               return;
               }
          printf ("Value is %d\n", n);
          }
     }

void f3 (int n, ...)
     {
     va_list ap;

     va_start (ap, n);
     while (n = va_arg (ap, int), n != 0)
          printf ("Value is %d\n", n);
     va_start (ap, n);
     while (n = va_arg (ap, int), n != 0)
          printf ("Value is still %d\n", n);
     va_end (ap);
     }

void f4a (va_list *pap)
```

17

```
        {
        int n;

        while (n = va_arg (*pap, int), n != 0)
             printf ("Value is %d\n", n);
        }

void f4 (int n, ...)
        {
        va_list ap;

        va_start (ap, n);
        f4a (&ap);
        va_end (ap);
        }

void f5a (va_list apc)
        {
        int n;

        while (n = va_arg (apc, int), n != 0)
             printf ("Value is %d\n", n);
        }

void f5 (int n, ...)
        {
        va_list ap;

        va_start (ap, n);
        f5a (ap);
        va_end (ap);
        }
```

a) Is each function in this translation unit strictly conforming? Note in particular:

in `f1`, the use of simultaneous `va_lists` in `f1`;

in `f2`, `va_start` and `va_end` are in different scopes;

in `f3`, there are two `va_starts` and one `va_end`;

in `f4`, the address of an object of type `va_list` is taken;

in `f4a` and `f5a`, `va_arg` is called with a first parameter which is not "the same as the `va_list ap` initialized by va_start (subclause 7.8.1.2).

b) Is the following implementation conforming?

`va_start` allocates a block of memory with `malloc`;

a `va_list` is a pointer to the block;

`va_end` frees the same block;

c) Is there any portable method to copy the current state of a `va_list`, for example in order that the remaining arguments can be scanned twice without knowledge of the `va_arg` calls made previous to that point. If the answer to (b) is "yes," I believe the answer to (c) must be "no."

Response

a) All functions listed except for `f3` contain strictly conforming code. The function `f3` violates the intended requirement for `va_start` and `va_end` to be invoked in matching pairs, as reflected in the following Correction.

b) There is nothing described in this section that would make such an implementation non-conforming.

c) No.

179

Correction

Change the last sentence of subclause 7.8.1 from:

The **`va_start`** and **`va_end`** macros shall be invoked in the function accepting a varying number of arguments, if access to the varying arguments is desired.

to:

The **`va_start`** and **`va_end`** macros shall be invoked in corresponding pairs in the function accepting a varying number of arguments, if access to the varying arguments is desired.

Add to subclause 7.8.1.1 at the end of the second paragraph:

`va_start` shall not be invoked again for the same **`ap`** without an intervening invocation of **`va_end`** for the same **`ap`**.

Defect Report #083

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 20 — use of library functions

Consider the following program:

```
#include <stdio.h>

int main (void)
     {
     printf ("%d\n", 42.0);
     return 0;
     }
```

This program clearly should have undefined behaviour, but I can find no wording which states so.

Correction

In subclause 7.1.7, page 99, line 5, insert after the words in parentheses in the second sentence:

or a type (after promotion) not expected by a function with variable number of arguments

Defect Report #084

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 21 — incomplete type in function declaration

Consider the following declarations:

```
struct tag;
extern void (*f) (struct tag);
```

At the point of the declaration of `f`, the type of the parameter is incomplete. Now a parameter is an object (subclause 3.15) with no linkage (subclause 6.1.2.2), but it is unclear whether this is a declaration of the parameter. If it is, then the declaration of `f` is forbidden by subclause 6.5. If it is not, then the declaration is strictly conforming. Which is the case?

If the type `struct tag` is completed before a call to `f`, is the call strictly conforming? Alternatively, since the declaration of `f` includes an incomplete type, is it possible to make a call to it at all?

Response

In responding to Defect Reports, the Committee has discussed at length when the size of an object is actually required. The C Standard is inconclusive with regard to whether or not the size of the structure is needed in the example given, leaving the behavior undefined.

The Committee should revisit this issue during the revision of the C Standard.

Defect Report #085

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 22 — returning from `main`

Consider the following program:

```
#include <stdlib.h>
#include <stdio.h>

int *pi;

void handler (void)
      {
      printf ("Value is %d\n, *pi);
      }

int main (void)
      {
      int i;

      atexit (handler);
      i = 42;
      pi = &i;
      return 0;
      }
```

Return from `main` is defined to be equivalent to calling `exit` (subclause 5.1.2.2.3). If the `return` statement was replaced by the equivalent call, the program would be strictly conforming. Is it strictly conforming without this replacement?

Note that if the answer is "yes," special processing will be required for return from `main`, which will depend on whether the call being returned from is the initial call or a recursive one.

Correction

Add at the end of the first sentence of subclause 5.1.2.2.3 the footnote:

In accordance with subclause 6.1.2.4, objects with automatic storage duration declared in `main` will no longer have storage guaranteed to be reserved in the former case even where they would in the latter.

183

Defect Report #086

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 23 — object-like macros in system headers

Consider an implementation where `<string.h>` defines the macro `strlen` thus:

```
#define strlen __internal_fast_strlen
```

and declares functions (defined elsewhere) called `__internal_fast_strlen` and `strlen`, both with the functionality of `strlen` in subclause 7.11.6.3. Is such an implementation conforming with respect to the rules of subclause 7.1.7?

Note that a strictly conforming application can detect this situation by comparing the value of the expression `strlen` taken before and after a `#undef`.

Response

The question asks whether a system header can define the name of a library function as an object-like macro, and cites subclause 7.1.7 as not using the term "function-like."

The Committee notes the absence of this term, but also notes that subclause 7.1.7 requires that the macro definition always be supressed when not followed by an open parenthesis. Therefore any macros must either be function-like, or the implementation must cause them to act as function-like macros.

Defect Report #087

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 24 — order of evaluation
Consider the following program:

```
int g;

int main (void)
        {
        int x;
        x = (10, g = 1, 20) + (30, g = 2, 40);        /* Line A */
        x = (10, f (1), 20) + (30, f (2), 40);        /* Line B */
        x = (g = 1) + (g = 2);                        /* Line C */
        return 0;
        }

int f (int i)
        {
        g = i;
        return 0;
        }
```

Subclause 6.3 makes the statement:

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

Consider line A. The full expression (the assignment to **x**) assigns two values (1 and 2) to **g**. Each such assignment is surrounded by sequence points. However, there is no sequence point between the two operands of the addition, and therefore no defined order of evaluation of the two inner assignments. There are a number of possible interpretations of the C Standard that can be made.

1) Multiple threads of evaluation may take place at one time (or equivalently, the evaluation of various parts of the expression may be interleaved to any level of detail), provided that anything specified to occur before a given sequence point actually takes place before anything specified to occur after the same sequence point. (This is equivalent to the "collateral evaluation" of Algol 68.)

In this interpretation, the expression is clearly undefined, because the two assignments to **g** may take place simultaneously and interfere destructively with one another. However, if this model is applied to line B, it yields the same result (since the sequence points occur at the same places). This means that, in effect, two function calls can be taking place simultaneously, and, if they modify the same object, the effect is undefined. This would surprise many users of the C Standard.

2) As (1), but assignments are atomic. This means that **g** has the value 1 or 2, though it is unspecified which. When applied to line C, this would also mean that **x** is specified to be assigned the value 3. This seems counter to the quoted provision of 6.3.

3) Any expression which completely fills the interval between two sequence points, and does not contain any embedded sequence points, is an "atomic sequence." The operations of any one atomic sequence are carried out together, and cannot be interleaved with any other atomic sequence. The order of the atomic sequences is unspecified, except that if the ending sequence point of one atomic sequence is the same as the starting point of another atomic sequence, they must be executed in that order.

In line A, there are five atomic sequences:

(i) evaluate 10

(ii) assign 1 to **g**

185

(iii) evaluate 30

(iv) assign 2 to g

(v) evaluate 20 and 40, add, and assign to x

(i) must come before (ii), (iii) must come before (iv), (v) must come after (ii) and (iv).

In line A this model has the same effect as (2), but it could differ in more complex expressions.

4) Multiple threads of execution can occur within an expression, but all except one are suspended while a function is being executed (this may, of course, spawn off new threads). This interpretation could be viewed as supported by the wording in subclause 6.6: "Except as indicated, statements are executed in sequence." It would have the effect of leaving line A undefined while line B is conforming (with it being unspecified whether the latter assigns 1 or 2 to g).

Which, if any, of these interpretations is correct? If none of them, what is the correct interpretation to make?

Response

In lines A and B, the expressions do not exhibit undefined behavior, but because the order of evaluation of the operands of the addition operator is not specified, it is unspecified whether g will attain the value 1 or 2. Line C violates the quoted restriction from subclause 6.3, so the behavior is undefined.

Defect Report #088

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 25 — compatibility of incomplete types

According to subclause 6.1.2.6 **Compatible type and composite type**, an incomplete structure type is incompatible with everything except "the same type:"

> Two types are compatible if their types are the same.

The C Standard fails to define when exactly two types are "the same." It is intuitively clear in context of basic types and array or pointer derivation, but becomes vague when genuinely new struct or union types are involved, especially when they are created as incomplete types first and completed later.

a) Are two incomplete structure types with a (lexically) identical tag always "the same" in the sense of subclause 6.1.2.6? It would appear not, unless they are declared in the same scope of the same translation unit.

b) Can two different incomplete structure types be compatible in other ways? If so, how?

c) Is a struct type before and after completion "the same type" in the sense of subclause 6.1.2.6? If the answer to (c) is no, then questions (d) to (g) apply.

d) Are the types before completion and after completion compatible?

Consider the following translation unit (the file a.c):

```
struct tag;

int a1 (struct tag * p)
      { a2 (p); }           /* Line A */

struct tag { int i; } s;

int main ()
      {
      a1 (&s);
      return 0;
      }

int a2 (struct tag * p)
      { /* ... */ }
```

e) Is the call to `a2` in line A valid? The parameter and argument types appear to be incompatible.

f) Suppose that the definition of `a2` were moved to a separate translation unit, preceeded by a definition of `struct tag` which was compatible with the one in the above translation unit. Would the call in line A then be valid?

g) A constraint in subclause 6.5 demands that:

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

Does this mean that:

```
struct tag;
extern struct tag* p;          /* Line B */

struct tag { int x; }
extern struct tag* p;
```

requires a diagnostic since the two declarations of `p` specify incompatible types? If not, what is the type `p` is declared as in Line B ?

If the answer to (c) is yes, then question (h) applies.

187

h) If two types A and B are compatible, is A compatible with all types that are the same as B? For example, is the call in line D below valid? If the redeclaration in line C is omitted, does undefined behaviour result?

```
/* First translation unit */

struct tag;
int c1 (struct tag * p)
     { /* ... */ }

struct tag { int i; };          /* Line C */

/* Second translation unit */

struct tag { int i; } s;
int main()
     {
     c1 (&s);          /* Line D */
     return 0;
     }
```

Response

a) Yes.

b) The question is too vague.

c) Yes. The C Standard failed to make clear that the type remains the same, but that is the obvious intent.

d) through (g) not applicable, because of the response to (c).

h) Yes, yes, and yes.

Defect Report #089

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 26 — multiple definitions of macros

Consider the following code:

```
#define macro            object_like
#define macro(argument)  function_like
```

Does this code require a diagnostic?

The wording of subclause 6.8.3 specifies that a macro may be redefined under certain circumstances (basically identical definitions), but does not actually forbid any other redefinition. Thus it can be argued that the constraint in subclause 6.8.3 is not violated, and a diagnostic is not required.

Correction

In subclause 6.8.3, change, in two places:

may be redefined by another **`#define`** preprocessing directive provided that

to:

shall not be redefined by another **`#define`** preprocessing directive unless

189

Defect Report #090

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 27 — multibyte characters in formats

Consider a locale where the characters `'\xE'` and `'\xF'` start and end an alternate shift state (i.e., the latter reverts to the initial shift state), and where multibyte characters whose first byte is greater than or equal to 0x80 are two bytes long. The multibyte characters and the alternate shift state characters are all distinct from the basic execution character set (subclause 5.2.1). What is the output generated by the following **`fprintf`** calls?

```
    fprintf (stdout, "Test A: (%d)\n", 42);
    fprintf (stdout, "Test B: (\xE%d\xF)\n", 42);
    fprintf (stdout, "Test C: (\xE%\xF" "d)\n", 42);
    fprintf (stdout, "Test D: (\xCC%d)\n", 42);
    fprintf (stdout, "Test E: (\xE\xCC%d\xF)\n", 42);
    fprintf (stdout, "Test F: (\xE\xCC%\xF" "d)\n", 42);
```

Response

The first call contains no locale-specific characters and must produce the obvious output. The remainder of this response addresses the subsequent calls.

The hypothetical locale is defined such that "the multibyte characters and the alternate shift state characters are all distinct from the basic execution character set." Thus the `%` character in the string literal is not the same character as the `%` that introduces a conversion specification (subclauses 7.9.6.1 and 7.9.6.2) because it is distinct.

The C Standard says, "the format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream." Therefore, the output generated by the example **`fprintf`** calls is the format argument copied unchanged to the output stream. Note that the third argument in each call to **`fprintf`** is not needed.

190

Defect Report #091

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 28 — multibyte encodings

Does a locale with the following encoding of multibyte characters conform to the C Standard?

The 99 characters of the basic execution character set have codes 1 to 99, in the order mentioned in subclause 5.2.1.1 (so `'A' == 1, 'a' == 27, '0' == 53, '!' == 63, '\n' == 99`).

The extended execution character set consists of 16,256 (127 * 128) two-byte characters. For each two-byte character, the first byte is between 1 and 127 inclusive, and the second byte is between 128 and 255 inclusive.

Note that any sequence of bytes can unambiguously be broken into multibyte characters, but the basic characters are prefixes of other characters.

Response

The hypothetical locale described does conform to the C Standard because the specified encoding does not violate the requirements imposed on multibyte characters by subclause 5.2.1.2. No additional requirements are needed.

Defect Report #092

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 29 — partial initialization of strings

Consider the following program:

```
#include <stdio.h>

int main (void)
    {
    char s [10] = "Hello";

    printf ("s [9] is %d\n", s [9]);
    return 0;
    }
```

Is this program strictly conforming? If so, is the value of `s [9]` guaranteed to be zero? Subclause 6.5.7 states:

> If there are fewer initializers in a brace-enclosed list than there are members of an aggregate, the remainder of the aggregate shall be initialized the same as objects that have static storage duration.

However, the initializer is not brace-enclosed, so this clause does not apply.

Response

See the response to Defect Report #060.

192

Defect Report #093

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Item 30 — reservation of identifiers

Can a conforming freestanding implementation reserve identifiers? Subclause 5.1.2.1 states that only one identifier (the equivalent of `main`) is reserved in a freestanding implementation. Subclause 7.1.3 states that certain identifiers are reserved, even when the corresponding headers are not included. This is a direct contradiction.

Response

The Committee observes that conforming freestanding implementations tend to vary widely in the library facilities provided, and that the simple binary choice implied by the above text is really a continuum. It also notea that it is difficult to provide a C implementation with no reserved names (not even those beginning with two underscores). It is therefore felt to be unreasonable to restrict the names available to implementors of freestanding implementations compared with hosted implementations.

The Committee notes that certain freestanding programs (such as UNIX kernels) have tended to use names such as `exit`, but agrees that existing practice dictates that the authors of such programs must already be prepared to change such names when using certain compilers.

Correction

In subclause 5.1.2.1, delete:

There are otherwise no reserved external identifiers.

Defect Report #094

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg1:

There appears to be an inconsistency between the constraints on "passing" values versus "returning" values. The constraints for function calls clearly indicate that a diagnostic is required if any given actual argument is passed (to a prototyped function) into a corresponding formal parameter whose type is not assignment compatible with respect to the type of the passed value. In the case of values returned by a return statement however, there seems to be no such compatibility constrain imposed upon the expression given in the `return` statement and the corresponding (declared) function return type.

A new constraint should be added to the C Standard like:

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

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

Response

The constraint in the description of the `return` statement is unneeded. Early on, the Committee decided that if a behavior was described as being equivalent to another construct, all of the constraints of that construct would apply. This "chaining" process means that any violation of a constraint in any section referred to explicitly or by the phrases "equivalent behavior" or "as if" will generate a diagnostic.

The semantics section of the `return` statement (subclause 6.6.6.4) states: "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." The constraints in the section on simple assignment (subclause 6.3.16.1) are sufficient to assure assignment compatibility of the two types.

Defect Report #095

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg2:

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

Subclause 6.5.7 says:

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

Note however that this rule itself appears within a semantics section, thus leading some implementors to feel that no diagnostics are required in cases where an attempt is made to provide an initializer for a given scalar and where the type of the initializer is *not* assignment compatible with the type of the scalar object being initialized. This ambiguity should be removed by adding an explicit constraint to the section covering initializations, such as:

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

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

Response

An explicit constraint is not required in the initializer section. Early on, the Committee decided that if a behavior was described as being equivalent to another construct, all of the constraints of that construct would apply. This "chaining" process means that any violation of a constraint in any section referred to explicitly or by the phrases "equivalent behavior" or "as if" will generate a diagnostic.

The constraints in the section on simple assignment (subclause 6.3.16.1) are sufficient to assure type compatibility of the object and the initializer.

195

Defect Report #096

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg3:

Subclause 6.5.4.2 **Array declarators** fails to contain any constraint which would prohibit the element type in an array declarator from being a type which is not an object type. (Note that subclause 6.1.2.5 seems to suggest that such usage is prohibited by saying that "An array type describes a contiguously allocated nonempty set of objects..." But this still leaves the matter rather unclear.)

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

Please consider the case of an array declarator, occuring at some point within a given translation unit, and indicating an element type `T`, where `T` is one of the following:

1) A function type.
2) A void type.
3) An incomplete struct or union type which is *never* completed within the given translation unit.
4) An incomplete struct or union type which *is* completed later within the given translation unit.
5) An incomplete array type which is *never* completed within the given translation unit.
6) An incomplete array type which *is* completed later within the given translation unit.

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

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

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

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

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

As I say, I believe that the very least the Committee should do is to add a constraint requiring diagnostics for array declarators whose element types fall into categories (1) or (2). The Committee may wish to provide an even more stringent interpretation of subclause 6.1.2.5 and also require diagnostics for element types falling into categories (3) and/or (5). The Committee may even wish to take *the* simplest approach to this entire problem, and simply require diagnostics for *any* case in which an array declarator specifies an element type which is not (already) an object type.

Regardless of which choice is made, I feel strongly that it is important for subclause 6.5.4.2 **Array declarators** to be revised to fully reflect both common sense and (to the extent possible) the intent of subclause 6.1.2.5.

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

Examples:

196

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

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

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

Response

Subclause 6.1.2.5 does clearly state, "An array type describes a contiguously allocated non-empty set of objects with a particular member object type, called the element type.[17]" Footnote 17 and the first paragraph of subclause 6.1.2.5 both state that object types do not include incomplete types. Nor do object types include function types. Thus, the array element type must not be any of the items you have listed. A diagnostic is not required. The Committee believes that this should be a quality of implementation issue whether or not a diagnostic is issued.

197

Defect Report #097

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg4:
Subclause 7.1.6 fails to contain any constraint which would prohibit the type argument given in an invocation of the **offsetof** macro from being an incomplete type. This situation can arise in examples such as the following:

```
#include <stddef.h>

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

I believe that a constraint prohibiting the type argument to **offsetof** from being an incomplete type is clearly needed.

This problem could be solved by adding an explicit constraint to subclause 7.1.6, such as:

> The type argument given in an invocation of the **offsetof** macro shall be the name of a complete struct type or a complete union type. (Note that this way of expressing the constraint also makes it completely clear that diagnostics are required for cases where the type given in the invocation is, for instance, a function type, an array type, an enum type, a pointer type, or a built-in arithmetic type.)

Response

See the response to Defect Report #040, question 6. This code is not strictly conforming.

198

Defect Report #098

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg5:

Subclause 6.3.3.4 provides the following constraint:

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

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

I have noted however that neither subclause 6.3.2.4 **Posfix increment and decrement operators** nor subclause 6.3.3.1 **Prefix increment and decrement operators** contain any constraints which would prohibit the incrementing or decrementing of pointers to function types or pointers to incomplete types.

I believe that this logical inconsistency needs to be addressed (and rectified) in the C Standard. It seems that the most appropriate way to do this is to add the following additional constraint to subclause 6.3.2.4:

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

Likewise, the following new constraint should be added to subclause 6.3.3.1:

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

Response

The explicit constraint on pre/post increment/decrement operators (subclauses 6.3.2.4 and 6.3.3.1) is not required. Early on, the Committee decided that if a behavior was described as being equivalent to another construct, all of the constraints of that construct would apply. This "chaining" process means that any violation of a constraint in any section referred to explicitly or by the phrases "equivalent behavior" or "as if" will generate a diagnostic.

Both subclauses 6.3.2.4 and 6.3.3.1 state in their respective **Semantics** sections, "See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers."

The **Semantics** section of subclause 6.3.16.2 states, "A compound assignment of the form `E1 op= E2` differs from the simple assignment expression `E1 = E1 op E2` only in that the lvalue of `E1` is evaluated only once."

This makes the pre/post increment/decrement equivalent to adding or subtracting 1 to/from an object. Looking at subclause 6.3.6 for the constraints on additive operators, in each case which refers to pointer operands, the C Standard uses the phrase "pointer to an object type." Since "object types" do not include "incomplete types" or "function types," their use as operands of these operators is precluded.

19

Defect Report #099

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg6:

Subclause 6.2.1.5 explicitly allows an implementation to evaluate a floating-point expression using some type which has *more* precision than the apparent type of the expression itself:

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

A footnote on this rule also says explicitly that:

> Cast and assignment operators still must perform their specified conversions, as described in 6.2.1.3 and 6.2.1.4.

As noted in the first of these two quotes (above) some compilers (most notably for x86 and mc680x0 target systems) may perform floating-point expression evaluation using a type which has more precision and/or range than that of the "apparent type" of the expression being evaluated.

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

My question is simply this: For the purposes of the above rules, does the term "assignment operator" mean exactly (and only) those operators listed in subclause 6.3.3.16, or should implementors and users expect that other operations described within the C Standard as being similar to "assignment" will also producing floating-point narrowing effects (under the right conditions)?

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

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

Example 1:

```
extern void callee ();    /* non-prototyped */
double a, b;
void caller ()
    {
    callee(a+b);   /* evaluated in long double then narrowed? */
    }
```

Example 2:

```
double a, b;

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

Response

A careful reading of the C Standard indicates that everything that is done "as if by assignment" must pass through a knot-hole (be narrowed). See the following references: subclause 6.3.16 for assignment, 6.3.2.2 for parameters, and 6.6.6.4 for return types.

207

Defect Report #100

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg7:

Subclause 6.6.6.4 The `return` statement says:

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

This is nonsensical. The type of the containing function is a function type, and that's different from an object type. I believe that should be changed to read:

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

Response

This error was corrected in response to Defect Report #001.

2c

Defect Report #101

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg8:

Subclause 6.3.2.2 **Function calls** says:

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

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

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

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

A similar problem exists within subclause 6.6.6.4 **The `return` statement**. It says:

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

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

const int returner () { return 99; }

Last but not least, subclause 6.5.7 **Initialization** says:

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

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

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

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

Response

There are three questions about mismatched type qualifiers in places where conversions "as if by assignment" takes place. Two of these are in initialization and in function returns. A careful reading of the C Standard shows that mismatched qualifiers are allowed in these two cases.

The other issue deals with a qualifier mismatch between arguments and the parameters of a called function. The C Standard should be modified to clarify that such a mismatch is allowed.

Correction

In subclause 6.3.2.2, page 41, lines 10-12, change:

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

to:

> If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type.

2[illegible]

Defect Report #102

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg9:

Subclause 6.5 **Constraints** says:

> If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.5.2.3."

Subclause 6.5.2.3, semantics subsection says:

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

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

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

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

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

Response

A diagnostic is required for the struct, union, and enum redeclarations indicated in the question. Subclause 6.5 indicates that there must be a diagnostic "except for tags as specified in 6.5.2.3." In subclause 6.5.2.3, the specified exception is for subsequent declarations that omit the bracketed list.

See also the response to Defect Report #017, Question 3.

20

Defect Report #103

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg10:

According to subclause 6.5:

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

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

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

```
typedef int AT[];

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

I believe that subclause 6.5 needs to be reworded so as to clarify that code such as that shown above is perfectly valid, and that conforming implementations should not reject such code out of hand.

Response

The types of the parameters are rewritten, as in subclause 6.7.1. No incomplete object types are involved.

205

Defect Report #104

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg11:

According to subclause 6.5:

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

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

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

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

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

Is it the intention of the Committee that code such as that shown above should be considered to be "strictly conforming?" If so, then some change to the wording now present in subclause 6.5 is in order (to allow for such cases).

Response

See Defect Report #084.

205

Defect Report #105

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg12:

Subclause 6.5 says (in its constraints section):

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

However in subclause 6.1.2.6 we have the following rule:

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

There is a conflict between the meaning of these two rules. The former rule indicates declaring something in two or more incompatible ways (in a given scope) *must* cause a diagnostic, while the latter rule indicates that doing the exact same thing may result in undefined behavior (i.e. possibly silent acceptance of the code by the implementation). (Note that this same issue was raised previously in the C Information Bulletin #1, RFI #17, question #3. While the response to that question indicated that no change was needed, a change *is* clearly need in order to resolve this ambiguity.)

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

To eliminate the first problem I would suggest that the rules quoted above from subclause 6.1.2.6 should be clarified as follows:

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

(Actually the rule regarding declaration compatability which now appears in subclause 6.1.2.6 seems entirely misplaced anyway. Shouldn't it just be taken out of subclause 6.1.2.6 and moved to the subclause on declarations, i.e. subclause 6.5?)

Response

This error was corrected in response to Defect Report #017, Question 3.

206

Defect Report #106

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg13:

Subclause 6.2.2.2 says:

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

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

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

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

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

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

Subclause 6.2.2.1 contains the following rules:

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

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

Furthermore, it should be noted that the definition of an lvalue (quoted above) *does not include* all void types. Rather, it only includes *the* `void` type.

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

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

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

20

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

Response

In the first function called **`example`**, the expression statement **`&*pv`** is dealt with in Defect Report #012. The remaining three statements are well formed. See the last sentence of the cited reference and also subclause 6.6.3.

In the second function called **`example`**, the expression statements **`&*pcv`** and **`&*pvv`** are dealt with in Defect Report #012. The remaining six statements are well formed. The restrictions given in subclause 6.5.3 apply to object types, not incomplete types.

The Committee was unanimous in agreeing that the existing wording was clear and did not need a careful reading of the C Standard.

20

Defect Report #107

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg14:

Subclause 7.2.1.1 (synopsis) says:

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

This synopsis raises several related questions.

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

b) Must a conforming implementation issue diagnostics for any and all attempts to invoke the **assert** macro for an expression having some type which is not directly convertible to type **int**?

Examples:

```
#include <assert.h>

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

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

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

Example:

```
#include <assert.h>

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

Response

a) The definition of **assert** depends on the **NDEBUG** macro. The synopsis provides information on how an implementation may use the parameter. If **NDEBUG** is defined as a macro, the parameter is not used and hence cannot cause undefined behavior. If **NDEBUG** is not defined as a macro, the implementation may rely on the parameter having type **int**. Passing a non-**int** argument in such a context will render the translation unit not strictly conforming.

b) If **NDEBUG**is defined as a macro, the parameter is not used and no diagnostic should occur. Otherwise, a violation of this requirement may be undefined behavior, which requires no diagnostic.

c) No requirements are imposed on an implementation in its treatment of programs that violate this requirement.

209

Defect Report #108

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg15:

Subclause 7.1.2.1 lists the set of reserved identifiers, but this list does not include keywords (subclause 6.1.1).

Subclause 6.1.1 says (in a semantics subsection):

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

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

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

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

My impression is that few (if any) existing implementations now accept such code. I am therefore inclined to believe that the Committee's true intentions were that *all* keywords (as listed in subclause 6.1.1) should be considered to be reserved identifiers, at least during translation phase 4, and at least while processing **#include** directives which name standard include files provided by the implementation (as listed in subclause 7.1.2).

I believe that the proper way to address this problem would be to add another stipulation (regarding reserved identifiers) to subclause 7.1.2.1. This additional stipulation might read as follows:

> If, during inclusion of any one of the standard headers listed in the preceeding section (during translation phase 4) any one of the keywords listed in subclause 6.1.1 is defined as a preprocessor macro, the behavior is undefined.

Response

This program's behavior is undefined because of the restriction on inclusion of standard headers in subclause 7.1.2:

> The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion.

The Committee's intention was indeed to otherwise allow macros to mask keywords.

2/

Defect Report #109

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg16:

Does the C Standard draw any significant distinction between "undefined values" and "undefined behavior?" (It appears that it does, but if it does, that fact is not always apparent.)

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

Subclause 6.3.8 **Relational operators:**

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

(Emphasis added.)

Subclause 7.5.2.1 **The `acos` function:**

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

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

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

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

int foo()
  {
  i = (p1 > p2);  /* Must this be "successfully translated"? */
  1/0;            /* Must this be "successfully translated"? */
  return 0;
  }
```

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

Response

The C Standard uses the term "indeterminate value" not "undefined value." Use of an indeterminate valued object results in undefined behavior.

The footnote to subclause 5.1.1.3 points out that an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated.

The result of translating a translation unit that is not strictly conforming is subject to the quality of implementation of that translator.

211

Defect Report #110

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg17:

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

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

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

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

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

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

Footnote: I have heard rumors that the issue of the exact timing of the decay of a formal parameter's array type into a pointer type (relative to the timing of the necessary check that the type of the formal parameter is in fact a valid type) was determined *explicitly* to be undefined by the Committee, but there is no record of this in the CIB #1 document I have. [CIB #1 is X3J11's earlier attempt to respond to Defect Reports #001-#035, then called Requests for Interpretation #001-#035.]

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

Response

No diagnostics are required for any of the above declarations. Each of the function declarations and definitions would render the translation unit not strictly conforming. See also Defect Report #047.

212

Defect Report #111

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg18:

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

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

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

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

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

Subclause 6.2.2.3:

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

Subclause 6.3.16.1 (constraints):

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

The rule specified in subclause 6.2.2.3 (and quoted above) makes it unclear whether a value of some pointer-to-qualified-object type may be *first* implicitly converted to type `(void*)` and then assigned to a type `(void*)` variable, or whether such implicit conversion only takes place as an integral part of an otherwise valid assignment operation.

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

Response

The constraint in subclause 6.3.16.1 takes precedence over subclause 6.2.2.3, thus a diagnostic is required. Note that the above quote from 6.2.2.3 is incomplete and taken out of context.

213

Defect Report #112

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg19:

Subject: Null pointer constants and relational comparisons.

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

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

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

Background:

Subclause 6.2.2.3:

> An integral constant expression with the value 0, or such an expression cast to type `void *`, is called a null pointer constant. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type.

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

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

Response

The code does not require a diagnostic but is undefined behavior, so it renders the translation unit not strictly conforming. Subclause 6.3.8 makes clear that this behavior is undefined.

214

Defect Report #113

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect report #rfg20:

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

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

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

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

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

Background:

Subclause 6.6.6.4 (constraints):

> A `return` statement with an expression shall not appear in a function whose return type is void.

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

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

> A `return` statement with an expression shall not appear in a function whose return type is a void type.

Response

Question 1: Yes, a diagnostic is required.

Question 2: Yes, this renders the program not strictly conforming code.

A qualified `void` function return type is disallowed by the constraints of subclause 6.7.1:

> The return type from a function shall be `void` or an object type other than array.

The constraint does not say "a void type" and thus `void` must not be qualified when used as a function return type. Since a qualified `void` return type is already invalid, there is no need for the additional constraint on the return statement (subclause 6.6.6.4).

2/5

Defect Report #114

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg21:

Subject: Initialization of multi-dimensional `char` array objects.

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

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

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

Background:

Subclause 6.5.7 (constraints):

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

Subclause 6.5.7:

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

Subclause 6.5.7 (examples):

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

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

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

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

Response

Question 1: Yes, a diagnostic is required.

Question 2: Yes, this renders the program not strictly conforming code.

The phrases "two dimensional array" and "three-dimensional array" are merely used for convenience. The semantics section on array declarators (subclause 6.5.4.2) and the syntax specification in the section on declarations (subclause 6.5.4) clearly show that array types must be declared with one index. Thus, an array which has two indices must be considered an "array of array of type."

Since this is the case, the semantics description for initializing aggregates and sub-aggregates in subclause 6.5.7 apply. This description states

> If the initializer of a subaggregate or the first member of the contained union begins with a left brace, the initialiers enclosed by that brace and its matching right brace initialize the members of the subaggregate or the first member of the contained union.

Thus, in the example, the string must be applied only to the first element of the two-element array (which is an array of 5 characters). Since the initializer contains 6 characters, it violates the constraint of the same section which states:

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

2/6

Defect Report #115

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg22:

Subject: Member declarators as declarators.

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

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

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

Background:

Subclause 6.5 (constraints):

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

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

Response

The Committee agrees that the quoted constraint can be read either way. Hence, a diagnostic is not required, but a program that uses such a form has undefined behavior. In the case of an aggregate, the intent was to require a tag or declarator for the aggregate itself. Thus, it is not unreasonable to issue a diagnostic for the given example. However, since the constraint can be read either way, an implementation could actually compile such a case though it is marginally useful at best.

217

Defect Report #116

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg23:

Subject: Implicit unary & applied to register arrays.

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

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

```
void example ()
        {
        register int array[5] = 0;

        array;          /* ? */
        array[3]; /* ? */
        array+3;        /* ? */
        }
```

Background:

Subclause 6.5.1 (footnotes):

> The implementation may treat the register declaration simply as an auto declaration. However whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier `register` may not be computed, either explicitly (by use of the unary & operator as discussed in 6.3.3.2) or implicitly (by converting an array name to a pointer as discussed in subclause 6.2.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier `register` is `sizeof`.

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

(Reference: CIB #1, RFI #17, question #6.)

Response

Question 1: No, a diagnostic is not required.

Question 2: Yes, this renders the program not strictly conforming code.

218

Defect Report #117

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg24:

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

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

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

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

Background:

Subclause 5.1.2.3:

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

Subclause 6.3:

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

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

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

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

This fact makes it inherently impossible to precisely determine even just the *relative* timings of various events (including the "occurence" of or the "execution" of or the "evaluation" of sequence points) which may (or must) occur sometime during the evaluation of a larger containing expression (except in a few cases involving `||` or `&&` or `?:` or `,` operators).

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

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

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

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

In this expression there are two "comma" sequence points, however nothing in the C Standard gives any indication as to which of these two may be (or must be) "evaluated" or "reached" first. (Indeed, it would seem that on a parallel machine of some sort, *both* points could perhaps be reached simultaneously.) It is

219

fairly clear however that each of the references to the stored values of **x** must not be evaluated until their respective preceeding "comma sequence points" have been "reached" or "evaluated." Thus, a partial (but very incomplete) ordering is imposed upon the sequence of events which must occur during the evaluation of this expression.

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

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

Note that in this (very realistic) example, the stored value of **x** is *never* modified more than once between any pair of sequence points. Given that the ordering described above is both a perfectly *plausible* and also a perfectly *permissible* ordering for the evaluation of the expression in question, and given that this particular permissible ordering of events does not violate the "modified at most once" rule (quoted earlier) it therefore appears that the expression in question may in fact be interpreted as being "strictly conforming," and that such expressions may appear within "strictly conforming" programs.

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

Response

The C Standard does not forbid an implementation from interleaving the subexpressions in the given example as specified above. Similarly, there is no requirement that an implementation use this particular interleaving. It is irrelevant that one particular interleaving yields code that properly delimits multiple modifications of the same object with sequence points. Any program that depends on this particular interleaving is depending on unspecified behavior, and is therefore not strictly conforming.

22

Defect Report #118

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg25:

Subject: Completion point for enum types.

Question: Are diagnostics required for the following code examples?

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

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

Background:

Subclause 6.3.3.4 (constraints):

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

Subclause 6.5.2.1 (semantics):

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

(Bracketed portions added for clarity.)

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

> For the example:
>
> ```
>         enum e { a = sizeof(enum e) };
> ```
>
> the relevant citations are subclause 6.1.2.1 starting on page 21, line 39, indicating that the scope of the first **e** begins at the {, and subclause 6.5.2.2, page 62, line 20, which attributes meaning to a later **enum e** *only if* this use appears in a *subsequent* declaration. By subsequent, we mean "after the }." Because in this case, the second **enum e** is not in a subsequent declaration, and no other wording in the C Standard addresses the meaning, the C Standard has left this example in the category of undefined behavior.

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

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

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

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

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

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

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

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

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

221

Even after such a clarification however, the status of:

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

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

Response

No, diagnostics are not required. This was corrected in response to Defect Report #013, Question 5.

222

Defect Report #119

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg26:

Subject: Initialization of multi-dimensional array objects.

Question A: Is a diagnostic required for the following declaration?

Question B: Is the following declaration "strictly conforming" or not?

```
static int array[][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Background:

Subclause 6.5.7 (semantics):

> If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements.

Subclause 6.5.7 (semantics):

> If the aggregate contains members that are aggregates or unions, or if the first member of a union is an aggregate or union, the rules apply recursively to the subaggregates or contained unions.

On the basis of the above quoted rules, one might conclude that the code example given above is strictly conforming. (Many existing implementations seem to disagree however.)

Response

A: No, a diagnostic is not required. It is a semantic requirement that array elements must be objects, not a constraint.

B: No, this is undefined behavior. Note that **`array`** does not have an array type because its element type is not an object type; hence subclause 6.5.7 does not apply. See subclause 6.1.2.5.

223

Defect Report #120

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg27:

Subject: Semantics of assignment to (and initialization of) bit-fields.

Question A: Is the following program "strictly conforming"?

Question B: Must a conforming implementation translate this code into an executable program which prints `3 3`?

```
#include <stdio.h>

struct S { unsigned bit:1; };
struct S object1 = { 3 };       /* ? */
struct S object2;

int main ()
      {
      object2.bit = 3;      /* ? */
      printf ("%u %u\n", object1.bit, object2.bit);
      return 0;
      }
```

Background:

Subclause 6.3.16.1 (semantics):

> In simple assignment (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

Subclause 6.2.1.2 (semantics):

> When a value with integral type is converted to another integral type, if the value can be represented by the new type, its value is unchanged.

Unless I'm mistaken, the type of the assignment expression:

```
      object2.bit = 3;
```

in the above example is type `unsigned int`. Thus, according to the rules quoted here, the value of 3 is converted to an `unsigned int` type value (during this assignment statement) and it is otherwise unchanged. Then, *that value of* 3 replaces the previous value of `object2.bit`.

I believe that the above examples illustrate the point that the C Standard currently fails to adequately describe the semantics of assignments to (and/or initializations of) bit-fields in cases where the value being assigned will not actually fit into the bit-field object.

In lieu of any description of the special semantics of assignments to bitfields, it appears to be currently *necessary* for both implementors and users to assume that the "normal" assignment semantics apply... but as you can see from the above examples, such assumptions lead to highly counterintuitive expectations (and to expectations which fly in the face of actual current common practice).

I believe that the Committee should rectify the current unfortunate situation by adding to subclause 6.3.16.1 (or maybe to subclause 6.2.1.2) some additional new verbage explicitly describing the special semantics of assignments to bit-fields.

Response

Subclause 6.5.2.1 states "A bit-field is interpreted as an integral type consisting of the specified number of bits." Thus the type of `object1.bit` and `object2.bit` can be described as `unsigned int : 1`. A larger integer is converted to this type according to the rules in subclause 6.2.1.2. Thus the value 3 is converted to the value 1.

The program is strictly conforming. It prints `1 1`.

224

Defect Report #121

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg28:

Subject: Conversions of pointer values to integral types.

Subclause 6.3.4 (semantics):

> A pointer may be converted to an integral type. The size of the integer required and the results are implementation-defined. If the space provided is not long enough, the behavior is undefined.

This passage is worded rather ambiguously.

In the first place, it talks about "The size of the integer required...." Required by whom? Required by what? I can't tell.

Also, I get the feeling that the way this passage reads, an implementation might permit conversions of pointers to types `char`, `short`, and `int` (with implementation defined semantics) while *disallowing* conversions of pointers to type `long`! (Of course that would be highly counterintuitive.)

Here is a suggested replacement for the above passage:

The value of any pointer expression whose `sizeof`, if computed, would be N, may be converted (via a cast) to any integral type whose `sizeof` is N or greater. The values resulting from such conversions are implementation-defined.

If an attempt is made to convert (via a cast) the value of a pointer expression whose `sizeof`, if computed, would be N, to some integral type whose `sizeof` is less than N, the behavior is undefined.

This is simply a more precise (and accurate) way of saying exactly what was (obviously) intended.

Response

The "size required" is that required by the implementation. The words "If the space provided is not long enough" make it clear that it is the size of the type that is relevant, and means that any type that is at least as long as the type of the "size required" is also acceptable. The size required need not be related to the result of `sizeof` applied to the expression.

2-2 5

Defect Report #122

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg29:

Subject: Conversion/widening of bit-fields.

Question: Must the following program print 1 or 0?

```
#include <stdio.h>

struct S { unsigned bit:1; } object = { 1 };

int main ()
      {
      printf ("%d\n", ((object.bit - 2) < 0));
      return 0;
      }
```

(At least one existing implementations prints 1 while another prints 0.)

Background:

Subclause 6.2.1.1:

> A `char`, a `short int`, or an `int` bit-field, or their signed or unsigned varieties or an object that has enumeration type 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 key phrase here is "the original type."

In effect, I am asking if the *type* of a bit-field is totally independent from its *width* for the purposes of the above rule.

If the answer to that question is "yes," then the value of `object.bit` must be considered to be an `unsigned int` (with a value of `1U`). In that case, the value 2 used in the above example must also be converted to type `unsigned int` and then the subtraction should be carried out on the two `unsigned int` values. The subtraction should then itself yield a value of type `unsigned int` which is itself (by definition) >= 0, so it would seem that the C Standard requires the above program to print 0.

Is that correct? If so, perhaps the wording of the above paragraph needs to be improved so as to make the correct interpretation of these rules more apparent to implementors.

Response

See Defect Report #015. "The original type" applies to both width and signedness. `object.bit` promotes to `int`, and the program prints `1`.

Defect Report #123

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg30:

Subject: "Type categories" and qualified types.

Question A: Is the following code "strictly conforming"?

Question B: Must a conforming implementation "correctly translate" the following code?

```
enum E1 { enumerator1 = (const int) 9 };          /* ? */
enum E2 { enumerator2 = (volatile int) 9 };       /* ? */
```

Background:

Subclause 6.5.2.2 (constraints):

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

Subclause 6.4 (semantics):

> Cast operators in an integral constant expression shall only convert arithmetic types to integral types...

Subclause 6.1.2.5:

> The type `char`, the signed and unsigned integer types, and the enumerated types are collectively called *integral types.*

Subclause 6.1.2.5:

> Any type mentioned so far is an unqualified type. Each unqualified type has three corresponding qualified versions of its type... The qualified or unqualified versions of a type are distinct types that belong to the same type category...

The problem is with the term "type category." I have been unable to find any actual definition of this term in the C Standard. My assumption is that *integral types* constitute one such "type category," but it would be nice to have the Committee's assurances about this. More specifically, I think that it would be advisable to add a statement somewhat like the following one just after the first paragraph in subclause 6.1.2.5:

> In addition to the partitioning of types into *object types, function types,* and *incomplete types,* each type is also said to belong to some *type category*. The *type categories* are *integral types, floating types, pointer types, structure types, union types, array types, void types,* and *function types.*

Response

A: Yes.

B: Yes.

As stated in subclause 6.5.3, "The properties associated with qualified types are meaningful only for expressions that are lvalues." The definition of "type category" is given in subclause 6.1.2.5, page 24.

227

Defect Report #124

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg31:

Subject: Casts to "a void type" versus casts to "the void type."

Question: Must a conforming implementation issue a diagnostic for the following code?

```
void example ()
        {
        (const volatile void) 0; /* diagnostic required? */
        }
```

Background:

Subclause 6.3.4 (constraints):

> Unless the type name specifies void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

Note that this constraint is *not* specific about whether a qualified void type is permitted in a cast or not; i.e. it should say either "a void type" or else say "the void type."

A quick check of several existing implementations seems to indicate that a majority of implementors have assumed that any void type (however qualified) is acceptable in a cast. Therefore it would seem prudent for the Committee to clarify the above quoted rule by changing "void type" to "a void type."

Correction

Change subclause 6.3.4 **Constraints** ***from:***

Unless the type name specifies `void` type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

to:

Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

228

Defect Report #125

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg32:

Subject: Using things declared as "extern (qualified) void."

Question: May a standard conforming implementation fail to correctly translate a translation unit containing the following declarations:

```
extern const void etext;
const void *vp = &etext;
```

Background:

RFI #12 in CIB #1 discusses at length the issue of applying unary `&` to an expression whose type is some void type. The conclusion of that discussion seem to be that although unary `&` *may not* be applied to an expression having *the* void type (because such expressions are not lvalues) it *is* permissible to apply unary `&` to an expression whose type is some qualified version of void. The text of the interpretation for RFI #12 even goes so far as to actively recommend the practice of declaring things to be **`extern`** and to have some qualified void type (so that the address may then be taken).

The question raised herein is a different one. Tom Pennello has pointed out the following rule from the second semantics paragraph of subclause 6.7:

> If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **`sizeof`** operator), somewhere in the entire program there shall be exactly one external definition for the identifier.

Thus, as Tom has noted, applying unary `&` to an entity declared to be both extern and of some qualified void type is a "use" of that entity which would necessarily force you to supply a definition of that entity, somewhere in the program. But as Tom has further noted, there is simply no way to accomplish that (in a strictly conforming program) because of the following rule (given in subclause 6.5):

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

Thus, if you either define or fail to define **`etext`**, it would appear that the behavior is undefined. Is this a correct interpretation?

(Footnote: It would appear that a strictly conforming program may contain a mere declaration of an extern entity whose type is any qualified or unqualified void type, but that any use of such an entity within an expression, other than within a **`sizeof`** expression, renders the program "not strictly conforming.")

Response

Applying `&` to an identifier of type `const void` is undefined behavior, as explained in the response to Defect Report #012, Question 1. Thus an implementation can define any semantics it wishes. A strictly conforming program cannot contain such a construct.

22

Defect Report #126

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg33:

Subject: What does "synonym" mean with respect to typedef names?

Question: Given the declarations:

```
typedef int *IP;
const IP object;
```

what is the type of `object`?

Background:

Subclause 6.5.6 says:

> A `typedef` declaration does not introduce a new type, only a synonym for the type so specified.

At least one person has wondered aloud about the true meaning of this rule.

Note that if the name `IP` in the above example is expanded as if it were a mere macro, then the type of `object` would be `(const int *)`. But essentially all existing implementations act as if there were some sort of magical parsing precedence (or extra parenthesization) which causes the `IP` (when used in the second line of the example above) to be treated as a single type, to which the `const` qualifier is applied (after the fact) thus resulting in `object` having type `(int * const)` rather than `(const int *)`.

While this treatment is well known to experienced implementors and users, it appears that the C Standard doesn't really explain it very well (or very precisely). I consider this to be a defect in the C Standard, worthy of the Committee's attention.

Response

A `typedef` introduces a name for a type. This is not a macro, and the type must indeed be "magically parenthesized." In

```
typedef int *ip;
ip x;
const ip y;
```

the type of `x` is *pointer to* `int`, and the type of `y` is `const` *pointer to* `int`. This is exactly analogous to the fact that

```
ip x1, x2;
```

declares both `x1` and `x2` as having the type *pointer to* `int`, and is not to be read as

```
int *x1, x2;
```

23

Defect Report #127

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg34:

Subject: Composite type of an enum type and an integral type.

Question: Given the declarations:

```
enum E { red, green, blue } object;
int object;
```

and given an implementation for which the type `int` is considered to be compatible with the type `enum E`, what is the composite type of `object` at the end of the translation unit which contains the above declarations?

Background:

Subclause 6.5.2.2 says:

> Each enumerated type shall be compatible with an integer type; the choice of type implementation-defined.

Subclause 6.1.2.6 says:

> A composite type can be constructed from two types that are compatible...
>
> ... 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.

Response

See Defect Report #013, Question 3. There is no requirement that the composite type be unique, and either of the types could be chosen as the composite type.

231

Defect Report #128

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg35:

Subject: Editorial issue relating to tag declarations in type specifiers.

Question: Given the code:

```
void example ()
      {
            {
            struct TAG {int i};
            }
            {
            struct TAG object;   /* line 7 */
            }
      }
```

Does line 7 violate the semantic rule given at the very end of the semantics sub-part of subclause 6.5, i.e., "If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator...?"

In other words, does `struct TAG` represent an incomplete type on line 7? (I believe that the answer is "yes," but the C Standard fails to make that entirely clear.)

Background:

Subclause 6.5.3.2 says:

> 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. It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.

These statements fail to take full account of scoping issues. The statements quoted above should be rephrased to take scope issues into account, perhaps as follows:

> If a type specifier of the form
>
> `struct-or-union identifier`
>
> occurs within a given scope prior to another declaration (in the same scope) of the same identifier (which also declares the identifier to be a struct or union tag) or if such a type specifier occurs at some point within a given scope where no prior declaration of the same tag identifier is visible, then the type specifier declares the identifier to be a structure or union tag for an incomplete structure or union type (respectively). The type so declared may only be used when the size of an object of the specified type is not needed.

Response

Yes, line 7 violates the semantic rule cited. Yes, `struct TAG` represents an incomplete type. The application of rules such as scope rules need not be restated at each relevant point in the C Standard.

232

Defect Report #129

Submission Date: 03 Dec 93
Submittor: WG14
Source: Ron Guilmette

Question 1

ANSI/ISO C Defect Report #rfg36:

Subject: Tags and name spaces.

Question: Should (or must) a conforming implementation "correctly translate" the following code?

```
void *vp;
struct TAG { int i; };

void f ()
        {
        enum TAG { enumerator };
        (struct TAG *) vp;
        }
```

Background:

Subclause 6.1.2.3 says:

> Thus, there are separate name spaces for various categories of identifiers, as follows:
>
> ...
>
> — the tags of structures, unions, and enumerations (disambiguated by following any of the keywords `struct`, `union`, or `enum`);...

A footnote for this subclause states that "There is only one name space for tags even though three are possible."

Given that this statement is only a footnote, and given that there are neither any specific constraints nor any specific semantic rules violated by the code shown above, it appears that a standard conforming implementation is actually *required* (by the C Standard, as now written) to accept the code shown above (even though this was probably not the intent of the Committee). It also seems that the code shown above is strictly conforming.

If the Committee actually intended that such code should be considered to be invalid, then it seems necessary to amend the C Standard to make it say that. (Actually, I think that a new constraint is in order here.)

Response

No change is necessary, because subclause 6.1.2.3 (second bullet) states that namespaces of tags are shared. Therefore the inner `enum TAG` hides the outer `struct TAG`, and therefore the cast `(struct TAG *)` attempts to declare a new `struct TAG` thus violating a constraint in subclause 6.5.

A conforming implementation need not translate the given code.

2 J J

Defect Report #130

Submission Date: 03 Dec 93
Submittor: WG14
Source: Sheng Yu

Question 1

Under subclause 7.9.2 **Streams**, page 125, lines 26-28:

> Data read in from a text stream will necessarily compare equal to the data that were earlier written out to the stream only if the data consist only of printable characters and the control characters horizontal tab and newline; ...

Writing on a text stream might not cause characters to be overwritten exactly one for one, especially on fixed-length record based file systems. If the file is not truncated beyond the point where the data is written, there is no sure way to predict what will be read in after writing in the middle of a text stream because the data might just replace a character, a line, etc. Consider the following example:

```
#include <stdio.h>
#include <string.h>
int buf[99];
unsigned int lex;
int main()
        {
        FILE *f = fopen("test data", "w");
        fwrite("abc\ndef\n", 8, 1, f);
        fseek(f, 0, SEEK_SET);
        fwrite("UWXYZ", 5, 1, f);
        fseek(f, 0, SEEK_SET);
        len = fread(buf, 1, 10, f);
        if (len == 8 && !memcmp(buf, "UWXYZef\n"))
                ;      /* Case 1: OK, acts like binary */
        else if (len == 5 && !memcmp(buf, "UWXYZ", 5))
                ;      /* Case 2: OK to truncate after write */
        else if (len > 5 && !memcmp(buf, "UWXYZ", 5))
                printf("len = %u, buf = %s", len buf);
                       /* Case 3: Is this nonstandard? */
        else
                printf("This is obviously nonstandard.");
        }
```

Is it conforming to the C Standard for a compiler that compiles the above program and produces the following output (Case 3):

```
len = 9, buf = UWXYZdef
```

Response

Yes, a conforming implementation may produce the "Case 3" output. However, there may be cases in some conforming implementations in addition to those shown in your example, so the printout "obviously nonstandard" may be inappropriate.

234

Defect Report #131

Submission Date: 03 Dec 93
Submittor: WG14
Source: Douglas Gwyn

Question 1

I've discovered an apparent bug in the C Standard. The code snippet:

```
struct {const int a[5]; } s1, s2;
void f(void) {s1 = s2; }
```

can be contained in a strictly conforming program, which runs counter to my understanding of the meaning of "const-qualification." That occurs because, according to subclause 6.5.3, the member `s1.a` is *not* const-qualified and thus slips past the modifiable-lvalue definition in subclause 6.2.2.1. Subclause 6.5.3 says that the *elements* of the array `s1.a` are const-qualified, not the array itself, and I can find no reasonable way to construe `s1.a[3]`, for example, as a "member" of `s1`; its only member is `s1.a`, as I see it. Apparently, the C Standard does not define the term "member," except implicitly through its use in subclause 6.3.2.3 **Semantics**, which says that `s1.a` is the member (on which the subscripting operator can operate to extract an element, but the element is not a member of the structure.)

What I think is desirable would be a required diagnostic for this example, as it *should* be considered to violate the constraint in subclause 6.3.16 that requires the left operand of an assignment operator to be a modifiable lvalue.

Relevant citations:

Subclause 6.2.2.1 **Lvalues and function designators:**

> A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

Subclause 6.3.16 **Assignment operators:**

> **Constraints:**
>
> An assignment operator shall have a modifiable lvalue as its left operand.

Subclause 6.5.3 **Type qualifiers:**

> If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.

Response

The example code is *not* strictly conforming, because some objects (the elements of the array `s1.a`) are being modified through use of an lvalue (`s1`) with non-const-qualified type, which according to subclause 6.5.3 results in undefined behavior.

However, a diagnostic is indeed desired here.

Correction

In the final sentence of the first paragraph of subclause 6.2.2.1, change the parenthetic remark from:

(including, recursively, any member of all contained structures or unions)

to:

(including, recursively, any member or element of all contained aggregates or unions)

235

Defect Report #132

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Can undefined behavior occur at translation time, or only at run time? If the former, then how does one distinguish the two cases in the C Standard?

Consider the translation unit:

```
/* No headers included */
int checkup()
    {
    /* Case 1 */
    if (0)
        printf("Printing.\n");
    /* Case 2 */
    return 2 || 1 / 0;
    }
```

Case 1 calls a function with a variable number of arguments without a prototype in scope. But the call is never actually executed. Now, subclause 6.3.2.2, in the first paragraph of page 41, states that this is undefined. Is it undefined to *translate* the code, or to *execute* it? The definition of undefined behavior (subclause 3.16) clearly allows the former, and subclause 5.3.2.2 does *not* say that the undefined behavior occurs only if the call is actually executed.

On the other hand, while subclause 6.3.5 uses similar wording about division by zero, "we all know" that my Case 2 is strictly conforming.

So what is the answer? If undefined behavior cannot occur at translation time, why the wording in subclause 3.16? If it can, how do I distinguish the possibilities? And, by the way, what is the answer for my Case 1?

Response

The translation unit must be successfully translated.

23

Defect Report #133

Submission Date: 03 Dec 93
Submittor: WG14
Source: Clive Feather

Question 1

Undefined behavior not previously listed in subclause G2:

1. Use of `sizeof` on `enum`, as in
`enum f(c = sizeof (enum f))`

2. A program containing no function called `main`

3. A storage class specifier or type-qualifier modifies the keyword `void` as a function parameter-type-list

4. Indexing of arrays exceeding specified sizes, as in:

```
int a[4][5];
a[1][7] = 0;
```

5. If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined.

6. In pointer-integer conversion, the size of integer required and the result are implementation-defined. If the space provided is not long enough, the behavior is undefined.

7. The result of the `%` operator is the remainder. In both this and the divide operations, if the value of the second operand is zero, the behavior is undefined.

8. As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.

9. If a file with the same name as a standard header, not provided as part of the implementation, is placed in any of the standard places for a source file to be included, the behavior is undefined.

10. If the signal handler `func(int sig)` executes a `return` statement and the value of `sig` was `SIGFPE` or any other implementation-defined value corresponding to a computational exception, the behavior is undefined.

11. If any signal is generated by an asynchronous signal handler, the behavior is undefined.

12. If copying takes place between objects that overlap, the behavior is undefined.

13. 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:

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

the invocation `f(2) (9)` results in undefined behavior.

14. A call to a library function exceeds an Environmental limit.

Response

The C Standard is sufficiently clear that the described behaviors are undefined. The next revision of the C Standard can include a more comprehensive list.

Defect Report #134

Submission Date: 31 Jan 94
Submittor: Project Editor (P.J. Plauger)
Source: Clive Feather

Question 1

Subclause 7.11.6.2 The `strerror` function, page 168, reads:

> The `strerror` function maps the error number in `errnum` to an error message string.

However, "error number" is an undefined term. Must `strerror` provide a valid message for every value of type `int`, or can some values be a domain error, allowing it to return garbage or a null pointer? If the latter, then what are the values that must generate a valid string? Must the following generate a valid string:

zero

`EDOM` and `ERANGE`

the value of any other symbol defined in `<errno.h>`

any value that a library routine might set `errno` to

Response

The `strerror` function must provide a valid message for the error numbers `EDOM`, `ERANGE`, and any other value a library function might store in `errno`. For all other values, the behavior is undefined.

228

Defect Report #135

Submission Date: 31 Jan 94
Submittor: Project Editor (P.J. Plauger)
Source: Per Bothner

Question 1

H.J. Lu points out that the SVR4 manual explicitly says that `fwrite(ptr, 0, 1, stream)` returns 0, not 1. I don't know what the SVID states.

I think it is more mathematically consistent to return 1 in this case. But in that case `fread(ptr, 0, 1, stream)` should also return 1, but ANSI explicitly states that it should return 0. I don't see any reason why these should be different, so I think it is best to follow existing practice. I think the ANSI specification for `fwrite` is a mistake; perhaps it should be fixed in the revision.

Response

There are no zero-length objects in C. Therefore, if the size argument to `fwrite` is zero, it is outside the domain of the function and (by subclause 7.1.7), the result is undefined. The C standard is not in conflict with the cited behavior of SVR4.

209

Defect Report #136

Submission Date: 31 Mar 94
Submittor: Project Editor (P.J. Plauger)
Source: Paul Eggert

Question 1

Suppose I run the following program in a US environment, where the clocks will jump forward from 01:59:59 to 03:00:00 on April 3, 1994. This program attempts to invoke `mktime` on a `struct tm` that represents 02:30:00 on that date. Does the C Standard let `mktime` return –1 in this case?

```
#include <stdio.h>
#include <time.h>
int main()
      {
      struct tm t;
      time_t r;

      /* 1994-04-03 02:30:00 */
      t.tm_year = 1994 - 1900; t.tm_mon = 3; t.tm_mday = 3;
      t.tm_hour = 2; t.tm_min = 30; t.tm_sec = 0;

      t.tm_isdst = -1; /* i.e. unknown */

      r = mktime(&t);
      if (r == -1)
            printf("mktime failed\n");
      else
            printf("%s", ctime(&r));
      return 0;
      }
```

The ANSI C Rationale (corresponding to subclause 7.12.2.3) clearly lets `mktime` yield –1 in the "fall-backward fold" that will occur when the clock is turned back from 01:59:59 to 01:00:00 on October 30, 1994. The question is whether `mktime` is also allowed to yield –1 in the "spring-forward gap" when the clock is advanced from 01:59:59 to 03:00:00.

This question arose when Arthur David Olson's popular "tz" time zone software was tested using NIST-PCTS:151-2, Version 1.4, (1993-12-03) a test suite put out by the National Institute of Standards and Technology that attempts to test C and Posix conformance. The PCTS package insists that in the above case, `mktime` must yield a `time_t` corresponding to either 01:30:00 or 03:30:00; i.e. PCTS rejects Olson's `mktime`, which yields –1.

This test case differs in an important way from the common practical use of `mktime` to "add 1" to the output of `localtime` or `gmtime`, since those functions normally set `tm_isdst` to a nonnegative value, whereas `tm_isdst` is –1 in the case under question.

I suggest that the Committee issue a clarification which makes it clear that `mktime` can yield –1 in the spring-forward gap when `tm_isdst` is –1.

Response

The Standard does not specify the behavior precisely enough to preclude `mktime` from returning a value of `(time_t)-1` and leaving the `tm_isdst` member set to –1 in such situations.

240

Defect Report #137

Submission Date: 30 Apr 94
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

Is `printf("%.1f", -0.01)` required to produce `0.0`, `-0.0`, or are both acceptable?

Subclause 7.9.6.1 says that when the + flag is not specified, the result begins with a sign only when a negative value is converted. The description of the `f` conversion (also `e` and `E`) says that the value is rounded to the appropriate number of digits. Is the value used to determine the sign of the result the value before or after rounding?

Response

As specified in subclause 7.9.6.1 for the + flag, a negative value is being converted, so a minus sign is required. The intent is that the sign is determined prior to conversion.

Defect Report #138

Submission Date: 02 Jun 94
Submittor: Project Editor (P.J. Plauger)
Source: John Max Skaller

Question 1

Subclause 6.1.2.4 says:

> An object has storage duration that determines its lifetime. There are two storage durations: static and automatic.

To me that clearly excludes heap objects. Is there a Defect Report on that? If so which one? (I have responses to Defect Report #001 through Defect Report #059) If not and something else in the Standard fixes it, can you point out where?

Correction

Change Subclause 6.1.2.4, page 22, lines 7-8 from:

There are two storage durations: static and automatic.

to:

There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.10.3.

242

Defect Report #139

Submission Date: 13 Jun 94
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

Compatibility of complete and incomplete types

The Committee has already endorsed the concept of using incomplete types which are completed in some translation units and left incomplete in others for encapsulation and data hiding (c.f. Defect Report #059). However, I can find nothing in the Standard which allows the incomplete type to be compatible with the completed type, which causes such usage to be not strictly conforming. I believe this to be an oversight.

Response

The behavior is currently undefined, and the Committee does not currently intend to alter the C Standard to define it.

2 4/3

Defect Report #140

Submission Date: 27 Jul 94
Submittor: BSI
Source: Andy Pepperdine

Question 1

Subclause 7.9.5.6 says:

> The **`setvbuf`** function may be used only after the stream ... has been associated with an open file and before any other operation is performed on the stream.

There are two related questions associated with this statement.

1. What does "performed" mean?

a) Does it include attempts that failed (such as **`fread`** on output file, etc.)?

b) In particular, does it include a failed attempt to **`setvbuf`**?

c) What about **`fprintf(f, "")`**?

2. What does "other operation" mean?

a) Does it include **`setvbuf`** itself?

b) Are **`ferror`** and **`feof`** operations?

c) What about **`clearerr`**?

Reasons for asking:

It would seem reasonable to try to get a very large buffer in some applications by attempting to do a **`setvbuf`** with, say, 1 MB of buffer space. If that fails, try again with 0.5 MB, etc. Is this allowed?

My *guess* as to the interpretation is as follows:

1. An operation is "performed" even if it fails for whatever reason.

2. All functions defined in subclause 7.9 are to be treated as "operations."

This is unsatisfactory, as the above approach of attempting to find a good buffer size would fail.

In the rationale, it states "The general principle is to provide portable code with a means of requesting the most appropriate popular buffering style, but not to *require* an implementation to support these styles." [Emphasis added.]

I interpret this as saying that **`setvbuf`** is an advisory call and need not be acted on. However, my questions above still stand as there seems to be no way of negotiating an agreement on good acceptable buffer sizes.

I believe that a clarification is required.

Response

As you say, "**`setvbuf`** is an advisory call and need not be acted on." That is to say, the C Standard allows it to fail. Therefore, discussions of detailed constraints such as you describe could only constitute non-normative advice to programmers or implementers. The Committee do not have any specific advice to give in this regard.