WG14/N297

X3J11/93-043

••

From x.co.uk!clive Tue Nov 2 12:13:41 1993 remote from uunet
Received: by plauger.UUCP (UUL1.3#20134)
from uunet with UUCP; Tue, 2 Nov 93 13:15:10 EST
Received: from cambs.x.co.uk by relay1.UU.NET with SMTP
(5.61/UUNET-internet-primary) id AA26831; Tue, 2 Nov 93 12:13:41 -0500
Message-Id: <9311021713.AA26831@relay1.UU.NET>
Received: from hunts.x.co.uk by cambs.x.co.uk with v15.11; Tue, 2 Nov 93 17:14:28 gmt
Received: by hunts.x.co.uk with v1.37.109.4; Tue, 2 Nov 93 17:14:37 GMT
From: uunet!x.co.uk!clive (Clive Feather)
Subject: SC22WG14 mailing
To: plauger!pjp
Date: Tue, 2 Nov 93 17:14:34 GMT
Mailer: Elm (revision: 70.85)

Here is a copy of my proposed Defect Report, which I would like adding to the mailing. Currently it is being considered by BSI but, due to the additional layer of mechanism, it hasn't been formally approved yet. However, Derek says that it should be no problem unless the C panel wish to modify it at their next meeting (which I can't attend, unfortunately). Alternatively, either WG14 or yourself may wish to adopt it as is (I am assured that BSI wouldn't mind this in the slightest).

You will be glad to hear (I hope :-) that I will definitely be at the Kona meeting.

Clive

PROPOSED DEFECT REPORT
==========
Clive D.W. Feather

This is a collection of items intended to form a Defect Report on ISO C. Most of them have been collected from Usenet discussions in the past, which is why they will seem familiar to many. In some cases, suggested Technical Corrigenda have been given; these may imply particular answers to the questions, and it is accepted that different answers may render these moot.

In these items, identifiers lexically identical to those declared in standard headers refer to the identifiers declared in those standard headers, whether or not the header is explicitly mentioned.

This collection has been prepared with considerable help from Mark Brader, Jutta Degener, and a person whose employment conditions require anonymity. However, opinions expressed or implied should not be assumed to be those of any person other than myself.

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 (6.2.2.3). Since return acts as if by assignment (6.6.6.4) the function is strictly conforming.

If f2, the 0 is a null pointer constant. However, a constant expression cannot contain a comma operator (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 0, is not a null pointer constant but only of type int, thus violating a constraint (6.3.16.1).

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 ("")
          * cannot yield NULL.
          */
         printf ("Decimal point = '%s'\n",
                 localeconv ()->decimal_point);
         exit (0);
```

```
        }
    }
```

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

Item 3 - locales

In a conforming implementation, can the value of any of the following expressions (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 ?

Suggested Technical Corrigendum: add to the end of the definitions of each of these four fields: "The string shall contain at most one multibyte character, and shall start and end in the initial shift state." or "The string shall start and end in the initial shift state." depending on the answer.

Item 4 - definitions of types

The terms "signed integer type", "unsigned integer type", and "integral type" are defined in 6.1.2.5. The 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 6 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 Standard to any of these 6 categories ?

(d) Can an implementation define other types (e.g. "__very long") which belong to any of these 6 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 ?

Suggested Technical Corrigendum: add to the end of the first paragraph on p23 (6.1.2.5): "The signed and unsigned integer types are collectively called the *integer types*.". In the second paragraph of p24 (6.1.2.5), add after the first sentence: "The signed integer types, the type char (if and only if char values are treated as signed), and any enumerated types that are compatible with signed integer types are collectively called the *signed integral types*. The remaining integral types are collectively called the *unsigned integral types*.

Item 5 - handling of char values

Values of the type char must be treated as either "signed" or "nonnegative" integers (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 ?

Suggested Technical Corrigendum: add to the end of the third or fourth paragraph of page 23 (6.1.2.5): "The type char behaves in all respects identically either to the type signed char or to the type unsigned char, although it is distinct from both."

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:

(In this transcription, words in {...} are in bold in the original, words in <...> are in italics in the original, and 2^4 means 2 with a superscript of 4.)

| | 05.03.15
| | {binary (numeration) system}
| | The <fixed radix numeration system> that uses the <digits> 0 and 1 and
| | the <radix> two.
| |
| | Example: In this <numeration system>, the numeral 110,01 represents the
| | number "6,25"; that is 1 x 2^2 + 1 x 2^1 + 1 x 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} (depreciated 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 depreciated in this sense because of its
| | mathematical use (see definition in 05.02.01).
| |
| | 05.03.07
| | {radix (numeration) system}
| | {radix 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.
| |
| | 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 Standard require that an implementation appear to use only one representation for each value of a given type ?

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

(e) Does the 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 behaviour, 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 524287 and INT_MIN is -79999.

(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 3 bits of the value are ignored (and each number has 8 representations). For signed values, the sign bit is the fourth bit from the top.

Furthermore:

(m) Does the Standard require that the values of SCHAR_MAX, SHRT_MAX, INT_MAX, and LONG_MAX in <limits.h> (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 Standard require that the values of UCHAR_MAX, USHRT_MAX, UINT_MAX, and ULONG_MAX in <limits.h> (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 ?

(r) Does the Standard require that the absolute values of SCHAR_MIN, SHRT_MIN, INT_MIN, and LONG_MIN in <limits.h> (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 Standard for other reasons, and if so, what are

they ?

(u) Does the 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 ?

Suggested Technical Corrigendum: change the second sentence of the second paragraph of p24 (6.1.2.5) to read: "The representation of integral types is described below." and delete footnote 18. Add a new subclause 6.1.2.5A as follows (in this text, alternative wording is shown by the notation "#choose", "#else", "#endif#"; (n) refers to footnotes given at the end of the text):

6.1.2.5A Representation of integral types

Each integral type is represented as a sequence of bits. For bit-fields, the number of bits is specified in the declaration of the bit-field. For each other integral type T, the number of bits in the representation is
#choose
the number of bits in an object of type T (1).
#else
less than or equal to the number of bits in an object of type T (1). If it is less than, then the values of any other bits in the object shall not affect the value represented.
#endif
Which bits are contained in which bytes of the object is unspecified.

For unsigned integral types, each bit shall have a weight. The weights of the bits shall be consecutive powers of 2, from 1 to 2^(N-1), where N is the number of bits in (the representation of) the type. The value represented by a number is the sum of the weights of the bits with value 1.

For signed integral types, one bit shall be designated the *sign bit*, and all other bits shall have a weight. The weights of the other bits shall be consecutive powers of 2, from 1 to 2^(N-2), where N is the number of bits in the representation of the type). If the sign bit is 0, the value represented by the number is the sum of the weights of the bits with value 1. If the sign bit is 1, then either:
- the value shall be the negative of the value represented if the sign bit had been 0, or
- the sign bit shall also have a weight,
#choose
which is either -2^(N-1) or 1-2^(N-1).
#else
which is negative and greater in magnitude than the weight of any other bit.
#endif

#choose
There shall be only one representation in a particular type for a

particular value. If a value can be represented in both an unsigned integral type and the corresponding signed integral type, the representation in both types shall be the same(2).
#else
It is permissible for a value to have more than one representation; all such representations shall compare equal.

If a value can be represented in both an unsigned integral type and the corresponding signed integral type, the representation in the unsigned type shall be a valid representation in the signed type.
#endif

(1) The number of bits in an object of type T is CHAR_BITS * sizeof(T).

(2) Thus a sign-and-complement machine must detect the bit pattern with the sign bit set representing the value 0, and convert it to the bit pattern with the sign bit clear before performing a bitwise operation.

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 (6.1.3.2), and they have the same representation (6.1.2.5). Footnote 16, which is not a part of the 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).

Item 8 - enumerated types

The Standard states (in effect) that an enumerated type is a set of integer constant values (6.1.2.5). It also states that an enumerated type must be compatible with an implementation-defined integer type (6.5.2.2). Finally, the integral promotions (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 0 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 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 ?

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 51 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 hack" ?

(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 hack" ?

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 malloced 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 ?

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" (6.3.6). For each of the marked lines, is the assignment strictly conforming ?

Item 11 - alignment and structure padding

The existence of structure padding (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 (3.1); for some type t, this is written A(t) in this item.

(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 6.3.3.4 ("the size is determined from the type of the operand") that "sizeof a" must equal "sizeof (struct fred)". Is this correct ?

(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)" ?

Suggested Technical Corrigendum: in two places in subclause 6.5.2.1, change "appropriate alignment" either to "appropriate alignment, or for any other purpose", or to "appropriate alignment, but not for any other purpose".

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 ?

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 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 12 ?

```
| | 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 Section ISO:6.2.2.1 above.  Therefore, as discussed in
| | the quote from Section ISO: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.
```

Suggested Technical Corrigendum: append to the fifth paragraph of the Semantics section of subclause 6.3.6: "and the result of that operator is not directly used as the operand of the unary & operator".

Note that, given "int x[5][5]", this change will explicitly permit "&x[4][5]" but not "&x[5][0]". If the latter is considered desirable, more complex wording will be needed.

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 ?

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 (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" format) 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.

(c) Can h ever return zero ? The library function memmove (7.11.2.2) completely meets the requirements for memcpy (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.

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;
}
```

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) ?

Item 18 - left shift operator

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

The Standard defines a bit (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 in 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 ?

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)
{
    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" (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 examples 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.

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.

Suggested Technical Corrigendum: in 7.1.7, page 99, insert, after the words in parentheses in the second sentence, the text: or a type (after promotion) not expected by a function with variable number of arguments,".

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 (3.15) with no linkage (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 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 ?

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 (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.

Suggested Technical Corrigendum: add to the end of the first sentence of 5.1.2.2.3: "except that 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.".

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 7.11.6.3. Is such an implementation conforming with respect to the rules of 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.

Suggested Technical Corrigendum: on the eighth line of 7.1.7, change "macro" to "function-like macro".

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 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 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 5 atomic sequences:

(i) evaluate 10
(ii) assign 1 to g
(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 6.6 "Except as indicated, statements are executed in sequence.". It would have the effect of leaving line A undefined while line B 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 ?

Item 25 - compatibility of incomplete types

Consider each of the following programs separately. Each program consists of two translation units. For each program:

(a) Do all the declarations of struct s specify compatible types ?
(b) (Programs 1 to 4 and 7 to 9 only) do all the declarations of p specify compatible types ?
(c) (Programs 5 to 9 only) do all the declarations of f specify compatible types ?
(d) (Programs 8 and 9 only) is the call valid ?
(e) (Programs 7 to 9 only) is the type of p compatible with the type of q ?
(f) Is the program strictly conforming ?

Program 1:

```
struct s;                    int i;
struct s *p;
struct s { int a; }
struct s *p;
main () { return 0; }
```

Program 2:

```
struct s;                    struct s;
extern struct s *p;              struct s *p;
main () { return 0; }
```

Program 3:

```
struct s;                    struct s;
extern struct s *p;              struct s { int a; };
main () { return 0; }            struct s *p;
```

Program 4:

```
struct s;                       struct s { int a; };
extern struct s *p;               struct s *p;
main () { return 0; }
```

Program 5:

```
extern void f (struct s *);          struct s;
main () { return 0; }              struct s { int a; };
                                 void f (struct s *pp) {}
```

Program 6:

```
extern void f (struct s *);          struct s { int a; };
main () { return 0; }              void f (struct s *pp) {}
```

Program 7:

```
extern void f (struct s *);          struct s;
struct s;                        struct s { int a; };
extern struct s *p;                struct s *p;
main () { return 0; }              void f (struct s *q) {}
```

Program 8:

```
extern void f (struct s *);          struct s;
struct s;                        struct s { int a; };
extern struct s *p;                struct s *p;
main () { f (p); return 0; }         void f (struct s *q) {}
```

Program 9:

```
extern void f (struct s *);          struct s { int a; };
extern struct s *p;                struct s *p;
main () { f (p); return 0; }         void f (struct s *q) {}
```

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 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 6.8.3 is not violated, and a diagnostic is not required.

Suggested Technical Corrigendum: in 6.8.3, change, in two places, "may be redefined by another #define preprocessing directive provided" to "shall not be redefined by another #define preprocessing directive unless".

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 (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);
```

Suggested Technical Corrigendum: in the description sections of 7.9.6.1 and 7.9.6.2, add a footnote mark after "Each conversion specification is introduced by the character %.", and add a footnote: "A byte with the same value as the character % occuring in an alternate shift state, or as the second or subsequent byte of a multibyte character, is not the character % that introduces a conversion specification.".

Item 28 - multibyte encodings

Does a locale with the following encoding of multibyte characters conform to the 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 16256 (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.

Suggested Technical Corrigendum: add an additional bullet item to 5.2.1.1: "- No multibyte character (including the single-byte characters) may be a proper prefix of another multibyte character in the same shift state.".

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.

Suggested Technical Corrigendum: in the penultimate paragraph of 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,".

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.

Suggested Technical Corrigendum: in subclause 5.1.2.1, change "There are otherwise no reserved external identifiers" to "The only other reserved external identifiers are those described in subclause 7.1.3.". Furthermore, in subclause 7.1.3, in the first two bullet items, change "reserved" to "reserved (in both freestanding and hosted implementations)", and in the remaining bullet items, add: "In freestanding implementations, only identifiers mentioned in subclauses 7.1.5, 7.1.6, and 7.8 are reserved.".

--
Clive D.W. Feather | Santa Cruz Operation | If you lie to the compiler,
clive@sco.com | Croxley Centre | it will get its revenge.
Phone: +44 923 816 344 | Hatters Lane, Watford | - Henry Spencer
Fax: +44 923 817 688 | WD1 8YN, United Kingdom | <== * NOTE NEW INFORMATION *
••