WG14/N388
X3J11/94-073

Defect Reports UK 002 to 025

BSI reference: IST/5/-/14 CP021.

These DRs were all accepted for forwarding to WG14 at the 1994-11-10 meeting of IST/5/-/14, who have not, however, examined the Suggested Technical Corrigenda.

--
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 1923 813541 | Hatters Lane, Watford | - Henry Spencer
Fax: +44 1923 813811 | WD1 8YN, United Kingdom |

Each Defect Report should be treated as if preceeded by the following boilerplate:

```
** Submitted to BSI by Clive D.W. Feather <clive@sco.com>

** In this Defect Report, 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 Defect Report has been prepared with considerable help from Mark
** Brader, Jutta Degener, Ronald Guilmette, 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.
```

Defect Report UK 002: consistency of implementation-defined values

The restrictions that apply to "implementation-defined" entities are not clear.

What restrictions apply to implementation-defined entities ? If the value of an expression is implementation-defined, need the implementation always produce the same result ?

For example, the value of the expressions "7/-3" and "8/-3" must each be either -3 or -2. Can an implementation make them different (that is, use a different implementation-defined choice for each), or must it make the same choice for all integral divisions involving a negative quantity ?

As another example, can the number of significant characters and the significance of case in an identifier with external linkage depend on the identifier itself, or must it be the same for all possible identifiers ?

Defect Report UK 003: zero sized allocations

The use of the word "unique" in subclause 7.10.3 is ambiguous, and the handling of zero size allocations is incomplete.

Part 1

7.10.3 reads:

> || If the size of the space requested is zero, the behavior is
> || implementation-defined; the value returned shall be either a null pointer
> || or a unique pointer.

Does the term "unique" mean "different every time", or does it mean "there is a single pointer returned by all calls with size zero" (as might be presumed from the ordinary dictionary definition of "unique") ?

In other words, if "malloc(0)" does not return a null pointer, is the following expression:

```
    malloc(0) == malloc(0)
```

003

always zero, always non-zero, or implementation-defined ?

Part 2

If unique means "there is a single pointer", what is the result of attempting to free that pointer ? How does the wording of 7.10.3 apply:
|| The value of a pointer that refers to freed space is indeterminate.
Possibly nothing happens, because the pointer does not really point to a block of memory. In that case, is the following code strictly conforming ?

```
#include <stdlib.h>
/* ... */
void *p = malloc (0);
if (p != NULL)
{
    free (p); /* Line A */
    free (p); /* Line B */
}
```

What is the behavior if each of lines A and B are reached ?

Part 3

If "unique" means "different every time", then each such call still consumes address space, even though no storage actually needs to be allocated, and therefore the call can fail due to exhaustion of memory. Thus malloc (0) can return a null pointer, while the Standard seems to suggest that an implementation can return either null pointers or unique pointers, but not both. This is a defect in the existing wording.

Suggested Technical Corrigendum

If "unique" means "there is a single pointer", then change the penultimate sentence of 7.10.3 from:

> If the size of the space requested is zero, the behavior is implementation-defined; the value returned shall be either a null pointer or a unique pointer.

to:

> If the size of the space requested is zero, the behavior is implementation-defined; the value returned shall be either a null pointer or a unique pointer. The values returned by two zero-length allocations shall compare equal. Freeing the value returned by a zero-length allocation shall have no effect. If that value is used as an operand of the unary * operator, or of a + or - operator except one whose other operand has integral type and value zero, the behavior is undefined.

If "unique" means "different every time", then change it to:

> If the size of the space requested is zero, the behavior is implementation-defined; either a null pointer is always returned, or the behavior is as if the size were some unspecified non-zero value. In the latter case, if the returned pointer is not a null pointer and is used as an operand of the unary * operator, or of a + or - operator except one whose other operand has integral type and value zero, the behavior is undefined.

[See also Defect Report UK 006.]

Defect Report UK 004: closed streams
====================================
Calls to fsetpos with positions in closed and reopened streams are permitted, but should be undefined.

The definition of fsetpos (subclause 7.9.9.3) requires the fpos_t argument to have a value generated by a successful call to fgetpos on the same stream. However, it does not require the stream to refer to the same file. If the stream does not so refer, the effect should be explicitly undefined.

Suggested Technical Corrigendum

004

In 7.9.9.3, change:

    ... an earlier call to the fgetpos function on the same stream.

to:

    ... an earlier call to the fgetpos function on the same stream; there
    shall not have been an intervening call to the fclose or freopen
    function with that stream.

Defect Report UK 005: legitimacy of type synonyms
==========

The Standard does not clearly indicate when the spelling of a type name is or is not significant; in other words, when a type name may be replaced by another type name representing the same type.

Part 1

Subclause 6.5.4.3 reads in part:

|| The special case of void as the only item in the list specifies that
|| the function has no parameters.

Subclause 6.7.1 reads in part:

|| (except in the special case of a parameter list consisting of a single
|| parameter of type void, in which there shall not be an identifier).

In both cases, the word "void" is set in the typeface used to indicate C code.

In the code:

```
    typedef void Void;

    extern int f (Void);

    int f (Void) { return 0; }
```

is the declaration on line 2 strictly conforming, and is the external definition on line 3 strictly conforming ?

Part 2

Subclause 5.1.2.2.1 reads in part:

|| It can be defined with no parameters:
|| int main (void) { /* ... */ }

Is the following definition of main strictly conforming ?

```
    typedef int word;

    word main (void) { /* ... */ }
```

Part 3

Are there any circumstances in which a typedef name is not permitted instead of the type it is a synonym for ? If so, what are they ?

Defect Report UK 006: null pointer conversions
==========

The Standard does not define semantics for the explicit conversion of null pointer constants and for the implicit conversion of null pointers.

Subclause 6.2.2.3 reads in part:

|| 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. Such a
|| pointer, called a null pointer, is guaranteed to compare unequal to a
|| pointer to any object or function.
||
|| Two null pointers, converted through possibly different sequences of
|| casts to pointer types, shall compare equal.

Given the definitions:

```
        void * p = 0;
```

005

```
        int  * i = 0;

does the standard guarantee that the expression

        p == i

always evaluates to 1 ? The last quoted sentence only covers casts, and
not the implicit conversions of that comparison. Conversely, do the
expressions:

        (int *) 0
        1 ? 0 : (int *) 0

yield null pointers of type (int *) ? The quoted text does not cover the
case of a null pointer constant being converted other than by assignment
or in a test for equality, yet expressions such as these are widely used.

Suggested Technical Corrigendum
-------------------------------
In subclause 6.2.2.3, change:
    Two null pointers, converted through possibly different sequences of
    casts to pointer types, shall compare equal.
to:
    Conversion of a null pointer to another pointer type yields a null
    pointer of that type. Any two null pointers shall compare equal.

Alternatively, a common term could be introduced to more conveniently
describe the various forms of pointer that cannot be dereferenced. In
this case, replace the last two paragraphs of subclause 6.2.2.3 with:
    For each pointer type, there exist values which can participate in
    assignment and equality operations, but which cause undefined behavior
    if dereferenced. These are referred to as _undereferenceable_. An
    undereferenceable pointer compares unequal to any other value of the
    same pointer type. For each pointer type, one particular undereferenceable
    pointer value is called the _null pointer_.[*]

    [*] Since there is only one such value, all null pointers of the same
        type compare equal.

    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 with an object
    of pointer type, or cast to pointer type, then it is converted to the
    null pointer of that type. Conversion of a null pointer to another pointer
    type produces the null pointer of that type.

If the answer to Defect Report UK 003 is that "unique" means "different
each time", then replace the last two sentences of subclause 7.10.3 with:
    If the size of the space requested is zero, an undereferenceable pointer
    is returned. It is implementation-defined whether this is always a null
    pointer or whether the implementation attempts to produce a value distinct
    from any other undereferenceable pointer. Any pointer value returned by
    an allocation can be passed to the free function; if the value is not a
    null pointer, it becomes indeterminate[*]. The value of a pointer that
    refers to any part of a freed object is also indeterminate.

    [*] A subsequent allocation may return a pointer value with the same
        bit pattern, but a strictly conforming program can't detect this.


Defect Report UK 007: consistency of the Standard
=================================================
Defects exist in the way the Standard refers to itself.

Part 1
------
The introduction to the Standard reads in part:
|| The introduction, the examples, the footnotes, the references, and
|| the annexes are not part of this International Standard.

While it is not, strictly speaking, an inconsistency for text that is not
part of the Standard to specify which text is part of the Standard, it is
```

006

confusing for this to be the case when other text that *looks* like part of the Standard isn't - the examples and footnotes.

In particular, placing this information - necessary for interpreting the text of the Standard itself - outside that text causes a danger that, when some other document is produced that purports to contain the full text of the Standard, the Introduction will be omitted while the footnotes and examples are retained. A reader of such a document who is not aware of the text of the introduction will then be misled as to the Standard's contents. Whilst this is not the responsibility of ISO, it is another reason for regularising the situation.

Note that this has definitely happened in the case of "The Annotated ANSI C Standard" by Herbert Schildt, and I have been informed (but have not confirmed) that it has also happened with the version of the Standard distributed by the Australian National Body.

Part 2

The introduction to the Standard reads in part:
|| The language clause (clause 7) ...
|| The library clause (clause 8) ...

These references are wrong.

Suggested Technical Corrigendum

In the introduction, change:
    The introduction, the examples, the footnotes, the references, and the annexes are not part of this International Standard.

    The language clause (clause 7) ...

    The library clause (clause 8) ...
to:
    As specified in the definitions and conventions clause (clause 3), this introduction, the examples, the footnotes, the references, and the annexes are not part of this International Standard.

    The language clause (clause 6) ...

    The library clause (clause 7) ...

Insert at the start of clause 3:
    The introduction, the examples, the footnotes, the references, and the annexes are not part of this International Standard.

Defect Report UK 008: reservation of identifiers
==
The Standard is unclear in its description of what applications can and cannot do with identifiers that are reserved to the implementation for certain uses.

Subclause 7.1.3 reads in part:
|| Each identifier with file scope listed in any of the following subclauses
|| (including the future library directions) is reserved for use as an
|| identifier with file scope in the same name space if any of its
|| associated headers is included.

Does this include reservation as macros ? In particular, is the following code:

```
    #include <stddef.h>
    #define size_t 42
```

strictly conforming, or could it cause a redefinition of the macro "size_t" ? Similarly, can another macro legitimately defined by <stddef.h> (such as offsetof) include size_t in its replacement list, so that:

```
    #include <stddef.h>
    #undef size_t
```

007

```
    #define size_t 42
    /* ... */
    offsetof (struct_type, field)

fails to expand correctly ? It is not clear how the wording of footnote
91 applies, and this is in any case not part of the Standard (except in
Australia :-).

Defect Report UK 009: details of reserved symbols
================================================
The wording of subclause 7.13 is unclear.

Does the term "any combination" in 7.13 include the empty combination ? In
other words, are names like E2, tom, LC_X, and memo reserved ?


Defect Report UK 010: gmtime and localtime
==========================================
The Standard's description of the static objects used by <time.h> functions
is misleading.

Subclause 7.12.3 reads in part:
|| these functions return values in one of two static objects: a broken-down
|| time structure and an array of char. Execution of any of the functions
|| may overwrite the information returned in either of these objects by any
|| of the other functions.

Does this mean that, for example, localtime and gmtime must share a single
broken-down time structure, and so the value returned from gmtime, if not
a null pointer, must equal the value returned from localtime (and this value
cannot change during execution of the program) ?

The wording "the other functions" also implies that a call to gmtime can
overwrite a previous call to localtime, but not a previous call to gmtime.
This is clearly ridiculous.

Suggested Technical Corrigendum
-------------------------------
In subclause 7.12.3, change:
    these functions return values in one of two static objects: a broken-down
    time structure and an array of char. Execution of any of the functions
    may overwrite the information returned in either of these objects by any
    of the other functions.
to:
    these functions each return a pointer to an object of static storage
    duration after assigning a value to it. Execution of any of these
    functions may overwrite the information returned in any of these objects
    by a previous call to any of these functions.


Defect Report UK 011: undeclared identifiers
============================================
The Standard is not clear on whether the use of an undeclared identifier as
a primary expression requires a diagnostic message.

Subclause 6.3.1 states that:
|| An identifier is a primary expression, provided it has been declared as
|| designating an object (in which case it is an lvalue) or a function (in
|| which case it is a function designator).

It has been suggested that if no declaration of some identifier is visible
in the current scope when that identifier appears in an expression, the
identifier is not a primary expression, and therefore the syntax of 6.3.1
is violated (in other words, there is no valid parse for the expression).
This would thus require a diagnostic for an undeclared identifier.

Is this interpretation correct ? If yes, then it needs to be made clear
that this does not prevent a previously undeclared function from being
called by a strictly conforming program (see 6.3.2.2).

If not, does an undeclared identifier require a diagnostic, and if so, why ?
If not, is this a deliberate policy, or is it a defect that needs correction ?
```

008

```
433
434
435  Defect Report UK 012: bad declarations
436  ======================================
437  The Standard contains no constraint to prevent declarations involving types
438  not defined by subclause 6.1.2.5.
439
440  Subclause 6.5 states that:
441  || A declaration shall declare at least a declarator, a tag, or the members
442  || of an enumeration.
443  There seems to be no constraint that a declarator generate a well-formed
444  type. Consider the following code:
445
446      {
447          int a [][5];      /* Line A */
448          int x, b [][5];   /* Line B */
449      }
450
451  Neither a nor b has a well formed type. Does line A nevertheless
452  "declare a declarator", or does it violate the quoted constraint ?
453  If it violates the constraint, does line B ?
454
455  Is it the intent of the Standard that an ill-formed (but syntactically
456  correct) type generate a diagnostic ? If so, then is there one, or does one
457  need to be added ?
458
459
460  Defect Report UK 013: tags and incomplete types
461  ===============================================
462  The wording of subclause 6.5.2.3 concerning tags is defective in a number
463  of ways.
464
465  Part 1
466  ------
467  The first paragraph states that:
468      If this declaration of the tag is visible, a subsequent declaration that
469      uses the tag and that omits the bracketed list specifies the declared
470      structure, union, or enumerated type.
471  This neither handles the case of a type name (for example, in the operand
472  of the sizeof operator), nor does it make it clear whether or not the rule
473  applies within the braces of the first declaration (the tag is in scope from
474  the open brace).
475
476  In other words, it fails to address either occurrence of "struct tag *"
477  in the following code:
478
479      {
480          struct tag { int i [sizeof (struct tag *)]; };
481          int j [sizeof (struct tag *)];
482          /* ... */
483      }
484
485  Part 2
486  ------
487  The second paragraph does not adequately distinguish between type specifiers
488  which refer to an incomplete type and those which refer to a type in an outer
489  scope. For example, in the following code, it fails to indicate whether or
490  not all the uses of the tag refer to the same type:
491
492      struct tag;
493      struct tag *p;
494      {
495          struct tag *q;
496          /* ... */
497      }
498      struct tag { int member; };
499
500  Part 3
501  ------
502  The handling of enumerated types before their content is defined is also
503  unclear; this was covered to some extent in DR013Q5 and the subsequent
504  discussion on the WG14 mailing list.
```

009

```
For example, what is the status of the following code:

    enum tag { e = sizeof (enum tag ******) };

or of:

    enum tag { e0, e1, e2, e3 };
    {
        enum tag2 { e4 = sizeof (enum tag); };
        enum tag  { e5 = sizeof (enum tag); };
    }

If an enumeration tag cannot be used before the end of the list defining its
contents, a diagnostic ought to be required.

Part 4
------
If the same tag is used in a type specifier with a contents list twice in the
same scope, it is unclear whether or not a diagnostic is required. It could
be argued that, since this is forbidden by the semantics in 6.5.2.3, it is
not excluded from the second constraint of 6.5, and so a diagnostic is
required by that constraint. However, this may be viewed as clutching at
straws. An explicit constraint should be added.

Suggested Technical Corrigendum
-------------------------------
Rather than making piecemeal changes to address each issue separately,
the whole subclause should be rewritten. Footnote numbers have been chosen
to match the present footnotes.

  Constraints
    A specific type shall have its content defined at most once.

    A type specifier of the form
        enum identifier
    without an enumerator list shall only appear when the type it specifies
    is complete.

  Semantics
    All declarations of structure, union, or enumerated types that have
    the same scope and use the same tag declare the same type. The type is
    incomplete [63] until the closing brace of the list defining the content,
    and complete thereafter.

    [63] An incomplete type may only be used when the size of an object of
         that type is not needed. [Append the present wording, or see
         Defect Report CA-2-09 - submitted independently - for alternative
         wording.]

    Two declarations of structure, union, or enumerated types which are in
    different scopes or use different tags declare distinct types. Each
    declaration of a structure, union, or enumerated type which does not
    include a tag declares a distinct type.

    A type specifier of the form
        struct-or-union identifier    { struct-declaration-list }
                                  opt
    or
        enum identifier    { enumerator-list }
                       opt
    declares a structure, union, or enumerated type. The list defines the
    *structure content*, *union content*, or *enumeration content*. If an
    identifier is provided[64], the type specifier also declares the
    identifier to be the tag of that type.

    [64] If there is no identifier, the type can, within the translation
         unit, only be referred to by the declaration of which it is a part.
         Of course, when the declaration is of a typedef name, subsequent
         declarations can make use of that typedef name to declare objects
         having the specified structure, union, or enumerated type.
```

010

A declaration of the form
    struct-or-union identifier ;
specifies a structure or union type and declares the identifier as the tag of that type[62].

[62] A similar construction with enum does not exist.

If a type specifier of the form
    struct-or-union identifier
occurs other than as part of one of the above constructions, and no other declaration of the identifier as a tag is visible, then it declares a structure or union type which is incomplete at this point, and declares the identifer as the tag of that type[62].

If a type specifier of the form
    struct-or-union identifier
or
    enum identifier
occurs other than as part of one of the above constructions, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

Defect Report UK 014: meaning of lvalue
=======================================
Constraints that require something to be an lvalue place an unacceptable burden on the implementation.

Subclasue 6.2.2.1 states in part:
|| An lvalue is an expression (with an object type or an incomplete type
|| other than void) that designates an object.

Given the declaration "int a [10], i;", the expression "a [i]" designates an object, and is thus an lvalue, if and only if "i" has a value between 0 and 9 inclusive (see Defect Report 076 for further details). Now consider the Constraint in subclause 6.3.3.2:
|| The operand of the unary & operator shall be either a function
|| designator or an lvalue that designates an object ...

This means that the expression "&a[i]" is a constraint violation whenever "i" has a value outside the range 0 to 9 inclusive, and that therefore a diagnostic is required, at run-time !

The defect is that the operand of the unary & operator does not need to be an lvalue that designates an object, but rather an lvalue which, if evaluated with its operands having suitable values, could designate an object.

There are probably other parts of the Standard with the same problem, such as 6.3.2.4, 6.3.3.1, and 6.3.16.

Defect Report UK 015: consistency of the Standard
===
The change to the n conversion specifier in subclause 7.9.6.2 made by TC1, DR014Q2, should also be applied to subclause 7.9.6.1. Change:
    No argument is converted.
to:
    No argument is converted, but one is consumed. If the conversion specification with this conversion specifier is not one of %n, %ln, or %hn, the behavior is undefined.

In addition, an entry something like:
    A %n conversion specification for the fprintf or fscanf functions is not one of %n, %ln, or %hn (7.9.6.1, 7.9.6.2).
should be added to Annex G.2.

Defect Report UK 016: consistency of the Standard
===
The change to subclause 6.3 made by TC1, DR053Q1, should also be applied in Annex G.2 (page 200).

```
Line
649
650  Defect Report UK 017: Trigraphs
651  ================================
652  The standard's description of the replacement of trigraphs is contradictory.
653
654  Subclause 5.2.1.1 reads in part:
655  || All occurrences in a source file of the following sequences of three
656  || characters (called trigraph sequences [7]) are replaced with the
657  || corresponding single character.
658  [...]
659  || Each ? that does not begin one of the trigraphs listed above is not
660  || changed.
661  Since the second character in each trigraph is a ? that does not begin
662  the trigraph, this is a direct contradiction.
663
664  Suggested Technical Corrigendum
665  -------------------------------
666  Change the last sentence of the cited text to:
667      Each ? that is not part of one of the trigraphs listed above is not
668      changed.
669
670
671  Defect Report UK 018: Operators and Punctuators
672  ================================================
673  The description of operators and punctuators is confusing, and the
674  constraints are contradictory.
675
676  Subclause 6.1.5 Constraints reads:
677  || The operators [ ], ( ), and ? : shall occur in pairs, possibly
678  || separated by expressions. The operators # and ## shall occur in
679  || macro-defining preprocessing directives only.
680
681  Subclause 6.1.6 Constraints reads:
682  || The punctuators [ ], ( ), and { } shall occur (after translation phase
683  || 4) in pairs, possibly separated by expressions, declarations, or
684  || statements. The punctuator # shall occur in preprocessing directives
685  || only.
686
687  Consider the code:
688
689      #define STR(x)  #x
690      STR ({)     /* Line A */
691      STR (:)     /* Line B */
692      STR ([)     /* Line C */
693      STR (#)     /* Line D */
694
695  Line A appears to be strictly conforming, since the first sentence of
696  the constraint of 6.1.6 does not apply during translation phase 4. Line B
697  violates the constraint of 6.1.5. The interpretation of line C depends
698  on whether the [ is an operator or a punctuator !
699
700  Line D violates both constraints, but again which one depends on whether
701  it is an operator or a punctuator, something which is not made clear in
702  the Standard.
703
704  Assuming that the intent was for line B to be strictly conforming, and
705  that "(after translation phase 4)" was inadvertently omitted from 6.1.5,
706  the first sentence of each of these Constraints is nugatory, as any
707  program which violates these constraints also violates a syntax rule
708  elsewhere in clause 6. The remaining sentences would be better expressed
709  as part of subclause 6.8. It is also arguable that the concepts of operator
710  and punctuator are better merged at the syntactic level, and separated out
711  only at the semantic level.
712
713  Suggested Technical Corrigendum
714  -------------------------------
715  Delete the Constraints of subclauses 6.1.5 and 6.1.6. Add the following
716  constraint to 6.8:
717      A # preprocessing token shall only occur within a replacement-list
718      or when permitted by the syntax rules of this subclause. A ##
719      preprocessing token shall only occur within a replacement-list.
720
```

012

Add to the end of the Constraints of subclause 6.1, just before the full stop:

, and shall not be # or ##

Alternative Suggested Technical Corrigendum

In subclause 6.1 syntax, delete both occurences of "operator" and replace the second occurence of "punctuator" by "pp-punctuator".

Delete subclauses 6.1.5 and 6.1.6, and replace them by the following:

6.1.5 Punctuators

Syntax:

```
pp-punctuator:
    punctuator
    pp-only-punctuator
pp-only-punctuator: one of
    # ## defined
punctuator:
    [ ] ( ) { } . ->
    ++ -- & * + - ~ ! sizeof
    / % << >> < > <= >= == != ^ | && ||
    ? : , : ; ...
    = *= /= %= += -= <<= >>= &= ^= |=
```

Semantics:

A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, some punctuators may specify an operation to be performed (an /evaluation/) that yields a value, or yields a designator, or produces a side-effect, or a combination thereof; in that context, the punctuator is known as an /operator/. An /operand/ is an entity on which an operator acts.

Add the following constraint to 6.8:

A # preprocessing token shall only occur within a replacement-list or when permitted by the syntax rules of this subclause. A ## preprocessing token shall only occur within a replacement-list.

Defect Report UK 019: ranges of integral types

It appears to be possible to create implementations with unreasonable arrangements of integral types.

Subclause 6.1.2.5 states various rules which allow the following deductions to be made:

```
SCHAR_MAX <= SHRT_MAX
SHRT_MAX  <= INT_MAX
INT_MAX   <= LONG_MAX
SCHAR_MIN >= SHRT_MIN
SHRT_MIN  >= INT_MIN
INT_MIN   >= LONG_MIN
SCHAR_MAX <= UCHAR_MAX
SHRT_MAX  <= USHRT_MAX
INT_MAX   <= UINT_MAX
LONG_MAX  <= ULONG_MAX
```

and, depending on the interpretation of the term "the same amount of storage":

```
sizeof (unsigned short) == sizeof (short)
sizeof (unsigned int)   == sizeof (int)
sizeof (unsigned long)  == sizeof (long)
```

However, (based on the preliminary discussions of DR 069, which allow padding bits in integral types) there does not appear to be any requirement for the following:

```
UCHAR_MAX <= USHRT_MAX
USHRT_MAX <= UINT_MAX
UINT_MAX  <= ULONG_MAX
```

013

```
793     sizeof (short) <= sizeof (int)
794     sizeof (int)   <= sizeof (long)
795     UCHAR_MAX <= INT_MAX
796
797 The first five of these are necessary to allow reasonable deductions to
798 be made about the behavior of types in the presence of padding bits (for
799 example, that unsigned long can hold any value representable in any
800 integral type). The sixth is necessary to allow the <ctype.h> functions
801 to behave sensibly (it is also assumed by example 2 of subclause 5.1.2.3).
802
803 Suggested Technical Corrigendum
804 -------------------------------
805 In subclause 6.1.2.5, change in the fourth paragraph:
806     In the list of signed integer types above, the range of values
807     of each type is a subrange of the values of the next type in the
808     list.
809 to:
810     In the list of signed integer types above, the range of values
811     of each type is a subrange of the values of the next type in the
812     list, and the size of an object of each type is not greater than
813     the size of an object of the next type in the list.
814
815 Add to the fifth paragraph:
816     The range of values of each unsigned integer type is a subrange of
817     the next type (in the list unsigned char, unsigned short, unsigned
818     int, unsigned long).
819
820 Add to the fifth or eighth paragraph:
821     The range of values of the type unsigned char is a subrange of
822     the values of the type int.
823
824
825 Defect Report UK 020: Relational and Equality operators
826 ======================================================
827 The descriptions of these operators with pointer operands contain several
828 defects.
829
830 Part 1
831 ------
832 Consider the following code:
833     char *s = "a string";
834     if (s >= NULL)
835         /* ... */
836
837 Subclause 6.3.8 Semantics reads in part:
838 || If the objects pointed to are not members of the same aggregate or union
839 || object, the result is undefined
840
841 This implies that the comparison causes undefined behavior.
842
843 Subclause 6.2.2.1 reads in part:
844 || Such a pointer, called a null pointer, is guaranteed to compare unequal
845 || to a pointer to any object or function.
846
847 This implies that the comparison is guaranteed to yield "false".
848
849 This is a direct contradiction.
850
851 Part 2
852 ------
853 Subclause 6.3.9 Semantics reads in part:
854 || Where the operands have types and values suitable for the relational
855 || operators, the semantics detailed in 6.3.8 apply.
856
857 This can reasonably be read as meaning that, whenever the constraints of
858 6.3.8 apply, its definitions should be used, even if that would result in
859 undefined behavior. [The phrase "and values" can reasonably be read as
860 requiring only that the pointers both be to objects; it does not necessarily
861 mean that the result of the comparison must be defined.]
862
863 It further reads:
864 || If two pointers to object or incomplete types are both null pointers,
```

014

|| they compare equal. If two pointers to object or incomplete types compare
|| equal, they both are null pointers, or both point to the same object, or
|| both point one past the last element of the same array object.

This says nothing about the comparison of any other pointers. Now, subclause
3.16 reads in part:
|| Undefined behavior is otherwise indicated [...] by the omission of any
|| explicit definition of behavior.

Thus, in:

```
    int a, b;
    &a == &b
```

the comparison causes undefined behavior !

Part 3

The above citation does not allow for the case where one pointer is to an
object, and the other is one past the last element of an array object. If
an implementation places two independent objects in adjacent memory
locations, a pointer to one would equal a pointer to just past the other
on many common implementations.

If these pointers are not to be viewed as identical, then the wording is
defective.

Suggested Technical Corrigendum

In subclause 6.2.2.1, replace the cited text by:
    Such a pointer is called a null pointer.

In subclause 6.3.9, replace the first paragraph of the semantics by:
    The operators == (equal to) and != (not equal to) shall yield 1 if the
    specified relation is true and 0 if it is false. If the operands have
    types suitable for those of a relational operator and values that would
    not cause undefined behavior if used with a relational operator, then
    the result of the comparison, either greater than or less than (both
    implying not equal to) or equal to, is the same as with a relational
    operator.
insert at the start of the second paragraph:
    Otherwise the operands are pointers, and they shall compare either
    equal or not equal.

If part 3 is viewed as an issue, then in the same paragraph change:
    or both point one past the last element of the same array object.
to:
    both point one past the last element of the same array object, or one
    points one past the last element of some array object and the other
    points to the first element of a different array object.

Defect Report UK 021: Line numbers
==================================
The concept of "line number" is not clearly defined when a token is split
over more than one physical source line.

Subclause 6.8.4 reads in part:
|| The line number of the current source line is one greater than the number
|| of new-line characters read or introduced in translation phase 1 (5.1.1.2)
|| while processing the source file to the current token.

Subclause 6.8.8 reads in part:
|| __LINE__ The line number of the current source line (a decimal constant).

Consider the program:

```
    #include <stdio.h>

    #define LNER __LINE__

    /* The next statement is on physical source lines 6 to 8 */
```

015

```
    int east_coast = __\
    LINE\
    __;
    /* The next statement is on physical source lines 10 to 13 */
    int main_line = L\
    N\
    E\
    R;

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

In each of the two substitutions, it is unclear whether the line number is the number of new-lines read to the *start of* the current token, or to the *end of* the current token, or to a specified point within the current token.

What is the output of this program ?

Defect Report UK 022: Implicit conversions

The wording dealing with the usual arithmetic conversions contains a number of errors; while the correct meaning is usually clear, a strict reading of the Standard shows some contradictions and/or unwanted side-effects.

Subclause 6.2.1.5 reads in part:

> || Many binary operators that expect operands of arithmetic type cause
> || conversions and yield result types in a similar way. The purpose is to
> || yield a common type, which is also the type of the result.

Subclause 6.3.15 reads in part:

> || The second operand is evaluated only if the first compares unequal to
> || 0; the third operand is evaluated only if the first compares equal to 0;
> || the value of the second or third operand (whichever is evaluated) is
> || the result.
> ||
> || If both the second and third operands have arithmetic type, the usual
> || arithmetic conversions are performed to bring them to a common type
> || and the result has that type.
> [...]
> || in which case the other operand is converted to type pointer to void,
> || and the result has that type.

These citations have several defects:

* The relational and equality operators apply the usual arithmetic conversions, but not to yield the type of result.
* The conditional operator ?: is not a binary operator, but is specified as performing the usual arithmetic conversions.
* The concept of conversions applies only to a value; 6.3.15 is therefore contradicting itself when it calls for both the second and third operands to be subject to conversion when only one of them is evaluated.
* The value of the result of the ?: is not necessarily that of the second or third operand, as the value may have been converted (possibly yielding a different value).

Suggested Technical Corrigendum

In 6.2.1.5, change the cited sentences to:

> Many operators cause the same pattern of conversions to be applied to two operands of arithmetic type. The purpose is to yield a common type, which, unless explicitly stated otherwise, is also the type of the operator's result.

In 6.3.15, change the cited wording to:

> The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result of the operator is the value of the second or third operand (whichever is evaluated), converted to the type described below.

016

```
1009      If both the second and third operands have arithmetic type, the type
1010      that the usual arithmetic conversions would yield if applied to those
1011      two operands is the type of the result.
1012      [...]
1013      in which case the type of the result is pointer to void.
1014
1015
1016  Defect Report UK 023: Correction to Technical Corrigendum number 1
1017  ==================================================================
1018  An example added by TC1 is wrong.
1019
1020  TC1 added the following example to subclause 7.9.6.2:
1021  || Add to subclause 7.9.6.2, page 138, another Example:
1022  ||   In:
1023  ||
1024  ||       #include <stdio.h>
1025  ||       /* ... */
1026  ||       int d1, d2, n1, n2, i;
1027  ||       i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
1028  ||
1029  ||   the value 123 is assigned to d1 and the value 3 to n1. Because %n can
1030  ||   never get an input failure the value of 3 is also assigned to n2. The
1031  ||   value of d2 is not affected. The value 3 is assigned to i.
1032
1033  This should set i to 1, not 3, as %n does not affect the returned assignment
1034  count.
1035
1036  Suggested Technical Corrigendum
1037  -------------------------------
1038  In the example, change:
1039      The value 3 is assigned to i.
1040  to:
1041      The value 1 is assigned to i.
1042
1043
1044  Defect Report UK 024: diagnostics for #error
1045  ============================================
1046  The rules concerning whether #error generates a diagnostic are contradictory.
1047
1048  Subclause 5.1.1.3 reads:
1049  ||  A conforming implementation shall produce at least one diagnostic message
1050  || (identified in an implementation-defined manner) for every translation
1051  || unit that contains a violation of any syntax rule or constraint. Diagnostic
1052  || messages need not be produced in other circumstances.
1053
1054  Subclause 6.8.5 reads:
1055  || Semantics
1056  || A preprocessing directive of the form
1057  ||      # error pp-tokens    new-line
1058  ||                      opt
1059  || causes the implementation to produce a diagnostic message that includes
1060  || the specified sequence of preprocessing tokens.
1061
1062  Since this is not in a Constraints section, these two statements directly
1063  contradict one another. Furthermore, the second statement can be read as
1064  applying to a #error directive that is excluded by a false #if condition.
1065
1066  Suggested Technical Corrigendum
1067  -------------------------------
1068  In 6.8.5, replace the entire subclause with:
1069      Constraints
1070      A #error preprocessing directive shall not occur in a translation unit.
1071      Any diagnostic message generated because of the violation of this
1072      constraint [*] shall include the sequence of preprocessing tokens in the
1073      directive.
1074      [*] The intent of this subclause is that #error indicates that translation
1075      should fail. As stated in 5.1.1.3, a translation unit excludes lines
1076      within the "false" side of #if...#else...#endif groups.
1077
1078
1079  Defect Report UK 025: preprocessing directives
1080  ==============================================
```

017

Preprocessing directives are not removed from the translation unit at
any point during or after translation phase 4, and thus wreck the syntax
analysis in translation phase 7.

Subclause 5.1.1.1 reads in part:
|| A source file together with all the headers and source files included
|| via the preprocessing directive #include, less any source lines skipped
|| by any of the conditional inclusion preprocessing directives, is called
|| a /translation unit/.

Nothing here, in the description of translation phase 4, or in subclause 6.8,
states that any preprocessing directive is removed (except for #include,
which is "replaced").

Consider the source file:

```
    #define QUIT return 0
    #if 0
    This is some junk
    #else
    int main (void)
    {
        puts ("Hello world\n");
    #endif
        QUIT;
    }
```

The translation unit resulting at the end of translation phase 4 is thus:

```
    #define QUIT return 0
    #if 0
    #else
    int main (void)
    {
        puts ("Hello world\n");
    #endif
        return 0;
    }
```

and this clearly does not match the syntax of "translation-unit" in
subclause 6.7.

Suggested Technical Corrigendum

In subclause 5.1.1.2, add at the end of the description of translation
phase 4:

    All preprocessing directives are then removed from the translation unit.