WG14/N332

X3J11/94-017

Technical Corrigendum 1

This document presents all the normative corrections to date recommended by ISO committee JTC1/SC22/WG14 (Programming language C) to Defect Reports #001 through #059 for International Standard ISO/IEC 9899:1990. Only those Defect Report questions that result in normative corrections are reproduced here. A separate document, called a Record of Responses, lists all Defect Reports and all responses, including those that are non-normative.

The normative changes presented here were crafted by technical experts from a number of ISO member nations. In particular, WG14 solicited, and received, extensive assistance from the ANSI-authorized committee X3J11, which developed the ANSI C Standard that became the ISO C Standard. Technical experts from BSI (UK) also contributed extensively to these normative changes.

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

Neither this introduction, the reproduced Defect Report questions, any remarks labeled Response, nor the Summary of Issues are normative. Each normative correction to ISO/IEC 9899:1990 occurs in the subclauses labeled **Correction** that follow each of the reproduced Defect Report questions.

Defect Report #001

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

Question 1

Do functions return values by copying?

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

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

Correction

In subclause 6.6.6.4, page 80, lines 30-32, replace:

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

with:

If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment toan object having the return type of the function.*

[Footnote *: The `return` statement is not an assignment. The overlap restriction in subclause 6.3.16.1 does not apply to the case of function return.]

Add to subclause 6.6.6.4, page 80:

Example

In:

```
struct s {double i;} f(void);
union {struct {int f1;
                struct s f2;} u1;
       struct {struct s f3;
                int f4;} u2;
      } g;
struct s f(void)
     {
     return g.u1.f2;
     }
/* ... */
g.u2.f3 = f();
```

the behavior is defined.

144

Defect Report #009

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

Question 1

Use of typedef names in parameter declarations

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

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

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

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

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

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

> In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

Consider this declaration:

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

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

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

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

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

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

Correction

In subclause 6.5.4.3, page 68, lines 2-4, replace:

In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

with:

If, in a parameter declaration, an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name.

145

Defect Report #011

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

Question 1

Merging of declarations for linked identifier

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

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

In particular, consider the code:

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

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

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

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

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

Further variants of this question can be asked:

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

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

Correction

In subclause 6.1.2.6, page 25, lines 19-20, change:

146

For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

to:

For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible*, if the prior declaration specifies internal or external linkage, the type of the identifier at the latter declaration becomes the composite type. [Footnote *: As specified in 6.1.2.1, the latter declaration might hide the prior declaration.]

Question 2

Interpretation of **extern**

Consider the code:

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

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

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

Is this what is intended?

Correction

In subclause 6.1.2.2, page 21, change:

If the declaration of an identifier for an object or a function contains the storage-class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

to:

For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible*, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the latter declaration becomes the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage. [Footnote *: As specified in 6.1.2.1, the latter declaration might hide the prior declaration.]

Question 4

Tentative definition of externally-linked object with incomplete type

If one writes the file-scope declaration

```
int i[];
```

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

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

or equivalently

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

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

Is this what is intended?

Correction

Add to subclause 6.7.2, page 84, a second Example:

If at the end of the translation unit containing

```
int i[];
```

the array `i` still has incomplete type, the array is assumed to have one element. This element is initialized to zero on program startup.

Defect Report #013

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

Question 1

Compatible and composite function types

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

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

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

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

Correction

In subclause 6.5.4.3, page 68, lines 22-25, change:

(For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 6.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

to:

(In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the type that results from conversion to a pointer type, as in 6.7.1, and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

Question 4

When a structure is incomplete

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

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

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

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

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

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

Correction

In subclause 6.5.2.3, page 62, line 27, change:

occurs prior to the declaration that defines the content

to:

occurs prior to the `}` following the ***struct-declaration-list*** that defines the content

149

Question 5

Enumeration tag anomaly

Consider the following (bizarre) example:

```
enum strange1 {
     a = sizeof (enum strange1)     /* line [2] */
};
enum strange2 {
     b = sizeof (enum strange2 *)   /* line [5] */
};
```

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

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

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

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

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

Correction

Add to subclause 6.5.2.3, page 63, another Example:

An enumeration type is compatible with some integral type. An implementation may delay the choice of which integral type until all enumeration constants have been seen. Thus in:

```
enum f { c = sizeof(enum f) };
```

the behavior is undefined since the size of the respective enumeration type is not necessarily known when `sizeof` is encountered.

150

Defect Report #014

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

Question 2

X/Open Reference Number KRT3.159.2

Subclause 7.9.6.2 The **fscanf** function states:

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

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

Correction

Add to subclause 7.9.6.2, page 137, line 4 (the n conversion specifier):

No argument is converted, but one is consumed. If the conversion specification with this conversion specifier is not one of **%n**, **%ln**, or **%hn**, the behavior is undefined.

Add to subclause 7.9.6.2, page 138, another Example:

In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to **d1** and the value 3 to **n1**. Because **%n** can never get an input failure the value of 3 is also assigned to **n2**. The value of **d2** is not affected. The value 3 is assigned to **i**.

151

Defect Report #016

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

Question 2

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

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

> If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

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

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

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

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

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

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

Correction

In subclause 6.5.7, page 71, line 41 through page 72, line 2, change:

If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

to:

If an object that has static storage duration is not initialized explicitly, then:

— if it has pointer type, it is initialized to a null pointer;

— if it has arithmetic type, it is initialized to zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules;

— if it is a union, the first named member is initialized (recursively) according to these rules.

152

Defect Report #017

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

Question 1

New-line in preprocessor directives

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

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

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

These three statements are not sufficient to categorize the following:

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

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

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

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

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

Correction

***Add to subclause 6.8, page 86, line 5,* (Description):**

A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

Question 2

Behavior if no function called `main` exists

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

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

Response

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

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

Correction

Add to subclause G.2, page 200:

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

Question 3

Precedence of behaviors

153

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

Correction

In subclause 5.1.1.3, page 6, lines 15-17, change:

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint.

to:

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined.

Add to subclause 5.1.1.3, page 6:

Example

An implementation shall issue a diagnostic for the translation unit:

```
char i;
int i;
```

because in those cases where wording in this International Standard describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error shall be diagnosed.

Question 6

`register` on aggregates

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

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

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

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

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

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

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

Correction

Add to subclause 6.5.1, page 58 (Semantics)***:***

If an aggregate or union object is declared with a storage-class specifier other than `typedef`, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

Question 9

Syntax of assignment expression

154

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

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

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

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

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

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

Correction

Add to subclause 6.3.16.1, page 54, another Example:

In the fragment:

```
    char c;
    int i;
    long l;

    l = ( c = i );
```

the value of `i` is converted to the type of the assignment-expression `c = i`, that is, `char` type. The value of the expression enclosed in parenthesis is then converted to the type of the outer assignment-expression, that is, `long` type.

Question 14

`const void` type as a parameter

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

Correction

Add to subclause G.2, page 201:

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

Question 16

Pointer to multidimensional array

Given the declaration:

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

Does the behavior become undefined when:

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

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

This case should be explicitly stated.

Arguments for/against:

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

Response

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

155

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

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

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

Correction

Add to subclause G.2, page 201:

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.3.6).

Question 17

Initialization of unions with unnamed members

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

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

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

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

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

Correction

In subclause 6.5.7, page 71, line 39, change:

All unnamed structure or union members are ignored during initialization.

to:

Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization. A union object containing only unnamed members has indeterminate value even after initialization.

In subclause 6.5.7, page 72, line 11, change:

The initial value of the object is that of the expression.

to:

The initial value of the object, including unnamed members, is that of the expression.

Question 19

Order of evaluation of macros

Refer to subclause 6.8.3, page 89. In:

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

it should be defined whether this results in:

1) `2*f(9)`

or

2) `2*9*g`

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

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

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

156

Response

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

For example, given the definitions:

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

the invocation:

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

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

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

and

```
    2*9*g
```

Correction

Add to subclause G.2, page 202:

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

Question 22

Gluing during rescan

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

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

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

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

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

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

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

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

Correction

Add to subclause 6.8.3.3, page 90:

Example

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y); /* equivalent to char p[] = "x ## y"; */
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)
```

157

```
in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding **hash_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the catenation operator.

Question 24

Improve English

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

Correction

In subclause 7.1.2, page 96, lines 32-33, change:

However, if the identifier is declared or defined in more than one header,

to:

However, if an identifier is declared or defined in more than one header,

Question 30

Successful call to **ftell** or **fgetpos**

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

Correction

In subclause 7.9.9.2, page 145, lines 39-40, change:

a value returned by an earlier call to the **ftell** function

to:

a value returned by an earlier successful call to the **ftell** function

In subclause 7.9.9.3, page 146, lines 10-11, change:

a value obtained from an earlier call to the **fgetpos** function

to:

a value obtained from an earlier successful call to the **fgetpos** function

Question 37

Function result type

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

Correction

In subclause 6.3.2.2, page 40, line 35, change:

The value of the function call expression is specified in 6.6.6.4.

to:

If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.6.6.4. Otherwise, the function call has type **void**.

Question 38

What is an iteration control structure or selection control structure?

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

Is it:

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

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

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

Correction

In subclause 5.2.4.1, page 13, lines 1-2, change:

— 15 nested levels of compound statements, iteration control structures, and selection control structures

to:

— 15 nested levels of compound statements, iteration statements, and selection statements

Question 39

Header name tokenization

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

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

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

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

Correction

Add to subclause 6.1, page 18 **(Semantics)*:***

A header name preprocessing token is only recognized within a **`#include`** preprocessing directive, and within such a directive, a sequence of characters that could be either a header name or a string literal is recognized as the former.

Add to subclause 6.1.2, page 20 **(Semantics)*:***

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword.

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

Constraint

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

Add to subclause 6.1.7, page 32 **(Semantics)*:***

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

159

Defect Report #021

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

Question 1

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

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

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

Correction

In subclause 7.9.6.1, page 132, lines 37-38, change:

For **o** conversion, it increases the precision to force the first digit of the result to be a zero.

to:

For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero.

160

Defect Report #022

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

Question 1

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

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

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

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

Correction

In subclause 7.9.6.2, page 135, lines 31-33, change:

An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence.

to:

An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.

In subclause 7.9.6.2, page 137, delete:

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

Add to subclause 7.9.6.2, page 137:

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.* [Footnote *: `fscanf` pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to `strtod`, `strtol`, or `strtoul` are unacceptable to `fscanf`.]

Defect Report #027

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

Question 1

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

Subclause G.5.2 states:

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

Response

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

Answer: Yes.

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

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

```
#define abc$ x
```

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

Correction

Add to subclause 6.8, page 86 (Constraints)*:*

In the definition of an object-like macro, if the first character of a replacement list is not a character required by subclause 5.2.1, then there shall be white-space separation between the identifier and the replacement list.*

[Footnote *: This allows an implementation to choose to interpret the directive:

```
#define THIS$AND$THAT(a, b)    ((a) + (b))
```

as defining a function-like macro `THIS$AND$THAT`, rather than an object-like macro `THIS`. Whichever choice it makes, it must also issue a diagnostic.]

162

Defect Report #040

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

Question 2

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

Correction

Add to subclause G.2, page 203:

— A call to a library function exceeds an environmental limit (7.9.2, 7.9.3, 7.9.4.4, 7.9.6.1, 7.10.2.1).

163

Defect Report #043

Submission Date: 10 Dec 92
Submittor: WG14
Source: X3J11/92-004 (Robert Paul Corbett)

Question 1

Defining **NULL**

Subclause 7.1.6 defines **NULL** to be a macro "which expands to an implementation-defined null pointer constant." Subclause 6.2.2.3 defines a null pointer constant to be "an integral constant expression with the value 0, or such an expression cast to type `void *`." The expression `4-4` is an integral constant expression with the value 0. Therefore, Standard C appears to permit

```
#define NULL    4 - 4
```

as one of the ways **NULL** can be defined in the standard headers. By allowing such a definition, Standard C forces programmers to parenthesize **NULL** in several contexts if they wish to ensure portability. For example, when **NULL** is cast to a pointer type, **NULL** must be parenthesized in the cast expression.

At least one book about Standard C suggests defining **NULL** as

```
#define NULL    (void *) 0
```

That definition leads to a subtler version of the problem described above. Consider the expression `NULL[p]`, where `p` is an array of pointers. The expression expands to `(void *)0[p]` which is equivalent to `(void *)(p[0])`. I doubt many users would expect such a result.

Have I correctly understood Standard C's requirements regarding **NULL**? If not, what are those requirements?

Correction

Add to subclause 7.1.2, page 96 (before* Forward references*):

Any definition of an object-like macro described in this clause shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

164

Defect Report #052

Submission Date: 21 Mar 93
Submittor: Project Editor (P.J. Plauger)
Source: Paul Edwards

Question 1

In subclause 7.12.2.3, page 172, the example is not strictly conforming. The `mktime` return is compared against `-1` instead of `(time_t)-1`, which could cause a problem with a strictly conforming implementation.

Correction

In subclause 7.12.2.3, page 172, line 16, change:

```
        if (mktime(&time_str) == -1)
```

to:

```
        if (mktime(&time_str) == (time_t)-1)
```

Question 2

Index entry for `static` lists subclause 3.1.2.2 instead of subclause 6.1.2.2.

Correction

In the index, page 217, change:

`static` storage-class specifier, 3.1.2.2, 6.1.2.4, 6.5.1, 6.7

to:

`static` storage-class specifier, 6.1.2.2, 6.1.2.4, 6.5.1, 6.7

165

Defect Report #053

Submission Date: 25 Mar 93
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

There's been a discussion on `comp.std.c` recently about accessing a pointer to a function with parameter type information through a pointer to a pointer to a function without parameter type information. For example:

```
int f(int);
int (*fp1)(int);
int (*fp2)();
int (**fpp)();

fp1 = f;
fp2 = fp1;        /* pointers to compatible types, assignment ok */
(*fp2)(3);        /* function types are compatible, call is ok */
fpp = &fp1;       /* pointer to compatible types, assignment ok */
(**fpp)(3);       /* valid? */
```

The final call itself should be valid since the resulting function type is compatible with the type of the function being called, but there's still a problem: Subclause 6.3 **Expressions**, page 38, says:

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[36]
>
> — the declared type of the object,
>
> — a qualified version of the declared type of the object,
>
> — a type that is the signed or unsigned type corresponding to the declared type of the object,
>
> — a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
>
> — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
>
> — a character type.
>
> [Footnote 36: The intent of this list is to specify those circumstances in which an object may or may not be aliased.]

This would appear to render the final call undefined since the stored value of `fp1` is being accessed by an lvalue that does not match its declared type: `(int (*)())` vs. `(int (*)(int))`.

I think that this example should be valid and that the above limitation is too strict. I think what we meant to say was "*a type compatible with* the declared type of the object," which would allow "reasonable" type mismatches without allowing aliasing between wildly different types.

Correction

In subclause 6.3, page 38, lines 18-21, change:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[36]

— the declared type of the object,

— a qualified version of the declared type of the object,

to:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[36]

— a type compatible with the declared type of the object,

— a qualified version of a type compatible with the declared type of the object,

166

Defect Report #054

Submission Date: 01 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Larry Jones

Question 1

Are the string handling functions defined in subclause 7.11 that have an explicit length specification (**`memcpy`**, **`memmove`**, **`strncpy`**, **`strncat`**, **`memcmp`**, **`strncmp`**, **`strxfrm`**, **`memchr`**, and **`memset`**) well-defined when the length is specified as zero?

Taking **`memcpy`** as an example, the description in subclause 7.11.2.1 states:

> The **`memcpy`** function copies **`n`** characters from the object pointed to by **`s2`** into the object pointed to by **`s1`**. If copying takes place between objects that overlap, the behavior is undefined.

The response to Defect Report #042 Question 1 indicates that:

> ... the "objects" referred to by subclause 7.11.2.1 are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of **`N`** bytes in length (i.e. treated as an array of **`N`** elements of character type).

Since, by definition, objects consist of at least one byte, this would imply that **`s1`** and **`s2`** are not pointing to objects when **`N`** is zero and thus are outside the domain of the function leading to undefined behavior.

I do not recall whether this was the Committee's intent or not, but it would seem that some clarification is in order.

Correction

Add to subclause 7.11.1, page 162:

Where an argument declared as **`size_t n`** specifies the length of the array for a function, **`n`** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call must still have valid values, as described in subclause 7.1.7. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

Defect Report #055

Submission Date: 14 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Loren Schall

Question 1

It has been suggested that the six macros **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** must have distinct values. Here is the relevant portion of subclause 7.7:

"The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to and the return value of the **signal** function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

...

An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function."

On the one hand, the reference to "the signal number corresponding to the specified condition" might be assumed to imply different numbers for each signal. On the other hand, the words "distinct values" were explicitly used for the three **SIG_*** macros and are conspicuously missing for the others.

Also, I think it's worth noting that the standard expects **raise** to work meaningfully (i.e. to be able to tell them apart).

Summary: must **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** have distinct values?

Correction

In subclause 7.7, page 120, lines 14-16, change:

and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

to:

and the following, which expand to positive integral constant expressions with distinct values that are the signal numbers, each corresponding to the specified condition:

168

Summary of Issues

The list that follows provides a brief summary of all issues raised as separate questions within Defect Reports #001 through #059. Please note that the one-sentence summaries that follow seldom do justice to the issues, which are often subtle or complex. Read them to get a sense of the area of the C Standard requiring interpretation or correction. Be warned that they may well fail to properly characterize the precise concern.

#001 10 Dec 92 X3J11/90-009 (Paul Eggert)

Q1: Do functions return values by copying?

#002 10 Dec 92 X3J11/90-010 (Terence David Carroll)

Q1: Should \ be escaped within macro actual parameters?

#003 10 Dec 92 X3J11/90-011 (Terence David Carroll)

Q1: Are preprocessing numbers too inclusive?

Q2: Should white space surround macro substitutions?

Q3: Is an empty macro argument a constraint violation?

Q4: Should preprocessing directives be permitted within macro invocations?

#004 10 Dec 92 X3J11/90-012 (Paul Eggert)

Q1: Are multiple definitions of unused identifiers with external linkage permitted?

#005 10 Dec 92 X3J11/90-020 (Walter J. Murray)

Q1: May a conforming implementation support a pragma which changes the semantics of the language?

#006 10 Dec 92 X3J11/90-020 (Walter J. Murray)

Q1: How does `strtoul` behave when presented with a subject sequence that begins with a minus sign?

#007 10 Dec 92 X3J11/90-043 (Paul Eggert)

Q1: Are declarations of the form `struct tag` permitted after `tag` has already been declared?

#008 10 Dec 92 X3J11/90-021 (Otto R. Newman)

Q1: Is dead-store elimination permitted near `setjmp`?

Q2: Should volatile-qualified functions be added?

#009 10 Dec 92 X3J11/90-023 (Bruce Blodgett)

Q1: Are typedef names sometimes ambiguous in parameter declarations?

#010 10 Dec 92 X3J1190-044 (Michael S. Ball)

Q1: Is typedef to an incomplete type valid?

#011 10 Dec 92 X3J11/90-008 (Rich Peterson)

Q1: When do the types of multiple external declarations get formed into a composite type?

Q2: Does `extern` link to a static declaration that is not visible?

Q3: Are implicit initializers for tentative array definitions syntactically valid?

Q4: Does an incomplete array get completed as a tentative definition?

#012 10 Dec 92 X3J11/90-046 (David F. Prosser)

Q1: Can one take the address of a void expression?

#013 10 Dec 92 X3J11/90-047 (Sam Kendall)

Q1: How does one form the composite type of mixed array and pointer parameter types?

Q2: Is compatability properly defined for recursive types?

Q3: What is the composite type of an enumeration and an integer?

Q4: When is a structure type complete?

Q5: When is the size of an enumeration type known?

#014 10 Dec 92 X3J11/90-049 (Max K. Goff)

Q1: Is `setjmp` a macro or a function?

Q2: How does `fscanf("%n")` behave on end-of-file?

169

Defect Report #055

Submission Date: 14 Apr 93
Submittor: Project Editor (P.J. Plauger)
Source: Loren Schall

Question 1

It has been suggested that the six macros **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** must have distinct values. Here is the relevant portion of subclause 7.7:

"The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to and the return value of the **signal** function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

...

An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function."

On the one hand, the reference to "the signal number corresponding to the specified condition" might be assumed to imply different numbers for each signal. On the other hand, the words "distinct values" were explicitly used for the three **SIG_*** macros and are conspicuously missing for the others.

Also, I think it's worth noting that the standard expects **raise** to work meaningfully (i.e. to be able to tell them apart).

Summary: must **SIGABRT**, **SIGFPE**, **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM** have distinct values?

Correction

In subclause 7.7, page 120, lines 14-16, change:

and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition:

to:

and the following, which expand to positive integral constant expressions with distinct values that are the signal numbers, each corresponding to the specified condition:

168

#018 10 Dec 92 X3J11/90-066 (Yasushi Nakahara)

Q1: Does `fscanf` recognize literal multibyte characters properly?

#019 10 Dec 92 X3J11/91-014 (Richard Wiersma)

Q1: Are printing characters implementation defined?

#020 10 Dec 92 X3J11/91-006 (Bruce Lambert)

Q1: Is the Relaxed Ref/Def linkage model conforming?

#021 10 Dec 92 X3J11/91-001 (Fred Tydeman)

Q1: What is the result of `printf("%#.4o", 345)`?

#022 10 Dec 92 X3J11/91-002 (Fred Tydeman)

Q1: What is the result of `strtod("100ergs", &ptr)`?

#023 10 Dec 92 X3J11/91-003 (Fred Tydeman)

Q1: If `99999 > DBL_MAX_10_EXP`, what is the result of `strtod("0.0e99999", &ptr)`?

#024 10 Dec 92 X3J11/91-004 (Fred Tydeman)

Q1: For `strtod`, what does `"C"` locale mean?

#025 10 Dec 92 X3J11/91-005 (Fred Tydeman)

Q1: What is meant by "representable floating-point value?"

#026 10 Dec 92 X3J11/91-007 (Randall Meyers)

Q1: Can one use other than the basic C character set in a strictly conforming program?

#027 10 Dec 92 X3J11/91-008 (Randall Meyers)

Q1: May a standard conforming implementation add identifier characters?

#028 10 Dec 92 X3J11/91-009 (Randall Meyers)

Q1: What are the aliasing rules for dynamically allocated objects?

#029 10 Dec 92 X3J11/91-016 (Sam Kendall)

Q1: Must compatible structures have the same tag in different translation units?

#030 10 Dec 92 X3J11/91-017 (Pawel Molenda)

Q1: May `sin(DBL_MAX)` set `errno` to `EDOM`?

#031 10 Dec 92 X3J11/91-018 (Pawel Molenda)

Q1: How are exceptions handled in constant expressions?

#032 10 Dec 92 X3J11/91-036 (Stephen D. Clamage)

Q1: Can an implementation permit a comma operator in a constant expression?

#033 10 Dec 92 X3J11/91-037 (Mike Vermeulen)

Q1: Must a conforming implementation diagnose "shall" violations outside **Constraints**?

#034 10 Dec 92 X3J11/91-038 (Stephen D. Clamage)

Q1: Is size information lost when a declaration goes out of scope, for objects with external linkage?

Q2: If so, can one then write conflicting declarations in disjoint scopes?

#035 10 Dec 92 X3J11/91-039 (Derek M. Jones)

Q1: Can one declare an enumeration or structure tag as part of an old-style parameter declaration?

Q2: If so, what is the scope of enumeration tags and constants declared in old-style parameter declarations?

#036 10 Dec 92 X3J11/91-040 (Fred Tydeman)

Q1: May a floating-point constant be represented with more precision than implied by its type?

#037 10 Dec 92 X3J11/91-043 (Isai Scheinberg)

Q1: Can UNICODE or ISO 10646 be used as a multibyte code?

#038 10 Dec 92 X3J11/91-046 (Kuo-Wei Lee)

Q1: What happens when macro replacement creates adjacent tokens that can be taken as a single token?

171

#039 10 Dec 92 X3J11/91-061 (Vania Joloboff)

Q1: Must `MB_CUR_MAX` be one in the `"C"` locale?

Q2: Should `setlocale(LC_ALL, NULL)` return `"C"` in the `"C"` locale?

#040 10 Dec 92 X3J11/91-062 (Derek M. Jones)

Q1: What is the composite type of `f(int)` and `f(const int)`?

Q2: Is an implementation that fails to equal the value of a library environmental limit conforming?

Q3: Does violation of an "environmental constraint" require a diagnostic?

Q4: Should the response to Defect Report #017 Q39 be reconsidered?

Q5: Can a conforming implementation accept `long long`?

Q6: Can one use `offsetof(struct t1, mbr)` before `struct t1` is completely defined?

Q7: Can `sizeof` be applied to earlier parameter names in a prototype, or to earlier fields in a struct?

Q8: What arithmetic can be performed on a `char` holding a defined character literal value?

Q9: Should the response to Defect Report #017 Q27 be reconsidered?

#041 10 Dec 92 X3J11/91-076 (Andrew Josey)

Q1: Are `'A'` through `'Z'` always `isupper` in all locales?

#042 10 Dec 92 X3J11/92-001 (Tom MacDonald)

Q1: Does `memcpy` define a (sub)object?

Q2: If so, how big is the object defined by `memcpy`?

Q3: How big is a string object defined by the `str*` functions?

#043 10 Dec 92 X3J11/92-004 (Robert Paul Corbett)

Q1: Can `NULL` be defined as `4-4`?

Q2: Can a macro that starts with an underscore be defined if a standard header is included?

#044 10 Dec 92 X3J11/92-010 (Steve M. Hoxey)

Q1: What does it mean to say that the type of `offsetof` is `size_t`?

Q2: Must the expansion of a standard header be a strictly conforming program?

Q3: Can expanding `offsetof` result in a non-strictly conforming program?

Q4: Can one use `offsetof` in a strictly conforming program?

Q5: How can`offsetof` be reconciled with the requirements for strictly conforming programs?

#045 10 Dec 92 X3J11/92-036 (David J. Hendricksen)

Q1: Can one `freopen` an already closed file?

#046 10 Dec 92 X3J11/92-041 (Neal Weidenhofer)

Q1: May a typedef be redeclared as a parameter in a new-style function parameter type list?

#047 10 Dec 92 X3J11/92-040 (Randall Meyers)

Q1: Can an array parameter have elements of incomplete type?

#048 10 Dec 92 X3J11/92-043 (David F. Prosser)

Q1: Is `abort` compatible with POSIX?

#049 10 Jan 93 David Metsky

Q1: Can `strxfrm` produce a longer translation string?

#050 24 Feb 93 C. Breeus

Q1: Does a proper definition of `wchar_t` need to be in scope to write a wide-character literal?

#051 08 Mar 93 Andrew R. Koenig

Q1: Can one index beyond the declared end of an array if space is allocated for the extra elements?

#052 21 Mar 93 Paul Edwards

Q1: Should the `mktime` example use `(time_t)-1` instead of `-1`?

Q2: Is the index entry for `static` correct?

172

Q3: Does the ISO C Standard come with a Rationale, as indicated in Footnote 1?

#053 25 Mar 93 Larry Jones

Q1: Do the aliasing rules cover accesses to compatible types properly?

#054 01 Apr 93 Larry Jones

Q1: What is the behavior of various string functions with a specified length of zero?

#055 14 Apr 93 Loren Schall

Q1: Must the `SIG*` macros have distinct values?

#056 15 Apr 93 Thomas Plum

Q1: How accurate must floating-point arithmetic be?

#057 07 Jun 93 Fred Tydeman

Q1: Must there exist a user-accessible integral type for every pointer?

#058 07 Jun 93 Fred Tydeman

Q1: What is the number of digits that can be processed by the `scanf` and `strtod` families?

#059 15 Jun 93 Martin Ruckert

Q1: Must an incomplete type be completed by the end of a translation unit?

#060 19 Jul 93 Larry Jones

Q1: Does a short string literal initialize an entire array?

#061 19 Aug 93 Ed Bendickson

Q1: Can a white-space directive in `fscanf` match zero input bytes?

#062 19 Aug 93 David J. Hendrickson

Q1: Can `rename` always fail if it must copy the file?