Defect Report #142

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 026: Reservation of macro names

Is it permitted to **#undef** a reserved macro name? Consider the translation unit:

```
#include <errno.h>
#undef EASTER
#undef EDOM
#undef __ERRNO_BASE
int error (void) { return errno == ERANGE; }
```

Considering each of the three **#undef** directives independently, is each directive permitted in a strictly conforming program? Is the translation unit strictly conforming?

Subclause 7.1.3 describes various classes of reserved identifiers, and then states:

> If the program declares or defines an identifier with the same name as an identifier reserved in that context (other than as allowed by 7.1.7), the behavior is undefined.

However, this does mention the use of **#undef**. Subclause 7.1.7 does so, for certain identifiers, but in rather ambiguous words:

> The use of **#undef** to remove any macro definition will also ensure ...

It has been suggested that this wording merely describes a strictly conforming coding technique, rather than establishing a special case (rather like the wording about placing the name in parentheses does).

Therefore, can a strictly conforming program **#undef** a name which is reserved for any use at that point?

There is a good reason to allow such an **#undef**. A program can make use of a identifier which is convenient but would otherwise be reserved (for example, the identifier **EASTER**). There is also a good reason to forbid it: the macro **ERANGE** might actually be defined as (**__ERRNO_BASE + 42**). This leads to the conclusion that it might be best to permit it for some names but not others.

A further example [inserted at the request of BSI] is the translation unit:

```
#include <stdlib.h>
#undef __INCLUDED_STDLIB_H
#include <stdlib.h>
```

Suggested Technical Corrigendum:

Add to the end of subclause 7.1.3:

If the program removes (with **#undef**) the macro definition of an identifier in the first group listed above, the behavior is undefined.

Technical Corrigendum

Replace the third bullet in subclause 7.1.3 with the following:

Each macro name in any of the following subclauses (including **Future library directions**) is reserved for use as specified if any of its associated headers is included, unless explicitly stated otherwise.

Forward reference: 7.17.

Defect Report #143

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 027: fopen modes

[BSI characterize this issue as minor.]

The definition of file opening modes is self-contradictory.

Subclause 7.9.5.3 reads in part:

> The argument mode points to a string beginning with one of the following sequences:

and then lists all of `r`, `r+`, `rb`, and `rb+` or `r+b`, with different meanings. Obviously, it is possible for a string to begin with up to three of these simultaneously, and thus the quoted text is contradictory.

Also, the wording is confusing since it can easily be misread as "beginning with exactly one of the following sequences," which would prohibit those of the specified modes that are longer than one character.

Suggested Technical Corrigendum:

Change the quoted text to:

> The mode is determined by the longest match of the following sequences to the initial characters of the string pointed to by the argument `mode`; at least the initial character shall match.

Response

The current C Standard has correct meaning, but the wording could be clearer. We suggest the following change for the revised Standard:

> The argument `mode` points to a string. The mode is determined by the string's longest initial match to the following sequences; at least the initial character shall match:

Defect Report #144

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 028: Preprocessing of preprocessing directives

Can the white space preceeding the initial # of a preprocessing directive be derived from macro expansion? Consider the following code extract:

```
#define EMPTY
# EMPTY include <file.h>  /* Line A */
EMPTY # include <file.h>  /* Line B */
```

Line A is clearly forbidden by subclause 6.8:

> The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

However, this text does not appear to forbid line B. Nor does subclause 6.8.3.4:

> The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one. If that subclause applies only to the expansion of `EMPTY`, it is not relevant. If it applies to both the expansion and the following preprocessing token sequence, then no subsequent preprocessing directive could ever be processed.

Is line B strictly conforming, or does it violate a constraint (and if so, which), or does it cause undefined behavior?

Suggested Technical Corrigendum:

In subclause 6.8 Description, change:

> A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that is either ...

to:

> A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that (at the start of translation phase 4, before any preprocessing takes place) is either ...

Response

The current C Standard has correct meaning, but the wording could be clearer. We suggest the following change for the revised Standard:

> A preprocessing directive consists of a directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that (at the start of translation phase 4) is either ...

Defect Report #145

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 029: Constant expressions

There is a confusion of contextual levels in subclause 6.4. Subclause 6.4 lists four possible forms for a constant expression in an initializer:

> Such a constant expression shall evaluate to one of the following:
>
> an arithmetic constant expression,
>
> a null pointer constant,
>
> an address constant, or
>
> an address constant for an object type plus or minus an integral constant expression.

The first two of these are syntactic forms, not something that a syntactic form would evaluate to. The third is the result of an evaluation, and the fourth is a compound of the two types of entity.

This confusion makes it unclear whether expressions like:

```
(int *)0
```

which is not a null pointer constant, or

```
&x[5] - &x[2]
```

which is clearly a constant, are permitted in initializers.

Suggested Technical Corrigendum:

Replace the quoted text with:

> Such a constant expression shall be either an arithmetic constant expression, a null pointer constant, or an address constant expression.

In the second subsequent paragraph, change:

> An address constant is a pointer to an lvalue designating an object of static storage duration, or to a function designator; it shall be created explicitly, using the unary & operator, or implicitly ...

to:

> An address constant expression shall have pointer type, and shall evaluate to:
>
> a null pointer,
>
> the address of a function, or
>
> the address of an object of static storage duration plus or minus some integer.
>
> The address may be created explicitly, using the unary & operator, or implicitly ...

Defect Report #146

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 030: Nugatory constraint

[BSI characterize this issue as minor.]

The constraint of 6.1.2 serves no purpose. Subclause 6.1.2 states in part:

> **Constraints**
>
> In translation phases 7 and 8, an identifier shall not consist of the same sequence of characters as a keyword.
>
> **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.

Given the latter text [added in Technical Corrigendum 1, reference DR 017 Q39], the constraint can never be violated.

Suggested Technical Corrigendum:

Delete the constraint of subclause 6.1.2.

Defect Report #147

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 031: Sequence points in library functions

There is no requirement for a sequence point to occur within a library function, since it might not be written in C. Consider the following code:

```
#include <string.h>
char s[10];
/* ... */
(strcpy)(s, "Testing") [0] = 'X';
```

Any function written in C must have a sequence point after the last full expression evaluated (which will be the returned value if there is one), so if `strcpy` were a C function, the assigning of `'T'` to `s[0]` would be completed before the call returned.

However, since library functions might not be written in C, they might not have such a sequence point. If not, then the above statement is in breach of the requirements of the second paragraph of subclause 6.3.

Suggested Technical Corrigendum:

Add to the end of subclause 7.1.7:

> There is a sequence point immediately before a library function returns.

Add to the end of annex C:

Immediately before a library function returns (7.1.7).

Add a reference to 7.1.7 in the **Forward References** of 5.1.2.3, and in the relevant **Index** entry.

Response

We agree that the current wording does not make this clear. The next revision of the C Standard will clarify that every function return is a sequence point.

Defect Report #148

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 032: Defining library functions

Subclause 7.1.7 is unclear about when it is permitted to declare a library function. Consider the following translation unit:

```
#include <math.h>
double (sin)(double);
```

Subclause 7.1.7 states in part:

> Any function declared in a header may be additionally implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included.

Since the wording uses the term "should", this does not appear to actually be a requirement on programs, and the code appears to be strictly conforming; in other words, the Standard here simply uses overly restrictive wording while trying to assist readers, and does not actually forbid the above code.

Is this interpretation correct?

Note that code such as the above is useful if the `#include` is conditionally compiled or is within a header not under the control of the code's author.

Suggested Technical Corrigendum:

If the intent was to forbid such a declaration, then change the quoted text to:

> A library function shall not be declared explicitly if its header is included.

If the intent was to allow the macros described in subclause 7.1.7 to be object-like macros (though other wording in 7.1.7 appears to forbid this), then change the quoted text to:

> A library function must not be declared explicitly if its header is included, unless any macro definition of the name has been removed with `#undef`.

If the intent was to allow the example declaration, then change the quoted text to:

> Any function declared in a header may be additionally implemented as a macro defined in the header, so one of the techniques below should be used to ensure that any explicit declaration of a library function is not affected by any such macro.

Response

The wording of the C Standard is as intended. The term "should" is intended as guidance to the reader. See also Defect Report #142.

Defect Report #149

Submission Date: 23 Feb 95
Submittor: BSI
Source: Clive D.W. Feather

Question

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, except where stated, opinions expressed or implied should not be assumed to be those of any person other than myself.

Defect Report UK 033: The term "variable"

[BSI characterize this issue as minor.]

The term "variable" is used in subclause 7.7.1.1, but is never defined in the Standard.

Suggested Technical Corrigendum:

In subclause 7.7.1.1, change:

> ... or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`.

to:

> ... or refers to any object with static storage duration other than by assigning a value to an object declared as `volatile sig_atomic_t`.

Response

The next revision of the C Standard will use the term "object" instead of "variable" uniformly.

Defect Report #150

Submission Date: 11 Jun 95
Submittor: DIN
Source: Jutta Degener

Question

This Defect Report was prepared with considerable help from Mark Brader, Clive Feather, Ronald Guilmette, and a person whose employment conditions require anonymity.

DIN-001:

According to the current standard, programs containing

```
char array[] = "Hello, World";
```

are not strictly conforming.

A **Constraint** in subclause 6.5.7 **Initialization**, demands that:

> All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

Subclause 6.4 **Constant expressions**, defines various kinds of constant expression. In its **Semantics** it states that a constant expression in an initializer evaluates to one of the following:

- an arithmetic constant expression
- a null pointer constant,
- an address constant, or
- an address constant for an object type plus or minus an integral constant expression.

String literals are neither. A string literal used to initialize a character array does not decay to a pointer to its first element, according to Subclause 6.2.2.1:

> Except when it is the operand of the `sizeof` operator or the unary `&` operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with `wchar_t`, an lvalue that has type "array of *type*" is converted to an expression that has type "pointer to *type*" that points to the initial element of the array object and is not an lvalue.

and hence is not an address constant.

Suggested Technical Corrigendum

In subclause 6.5.7, change:

> All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

to:

> All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions or string literals.

Technical Corrigendum

In subclause 6.5.7, change:

> All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

to:

> All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions or string literals.

Defect Report #151

Submission Date: 11 Jun 95
Submittor: DIN
Source: Jutta Degener

Question

This Defect Report was prompted by articles posted to comp.std.c by Bakul Shah, Rex Jaeschke, and Soenke Behrens.

DIN-002:

The C Standard's specification of what

```
printf("%#.0o", 0);
```

outputs is ambiguous, and compiler vendors have indeed interpreted it differently.

For a zero integer value, the descriptions of the # flag and the 0 precision in subclause 7.9.6.1 contradict each other.

The # demands that the first digit be zero;

> # The result is to be converted to an "alternate form." For o conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero.

But a precision of 0 demands that nothing at all be printed.

> o,u,x,X [...] The result of converting a zero value with a precision of zero is no characters.

In the hexadecimal case, the description of the # flag's effects has been worded such that the conflict is avoided:

> # [...] For x (or X) conversion, a nonzero result will have 0x (or 0X) prefixed to it.

If it was intended that the octal case, too, should print nothing, this crucial "nonzero" should be introduced into its description as well.

Suggested Technical Corrigendum

In subclause 7.9.6.1, replace:

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

by:

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

Response

The C Standard is clear enough as is. The call

```
printf("%#.0o", 0)
```

should print 0.

Defect Report #152

Submission Date: 11 Jun 95
Submittor: DIN
Source: Jutta Degener

Question

This Defect Report was prepared with considerable help from Mark Brader, Clive Feather, Ronald Guilmette, and a person whose employment conditions require anonymity.

DIN-003:

Can `longjmp` be used to return from a signal handler invoked other than through `abort` or `raise`? The descriptions of `signal` and `longjmp` contradict each other.

According to subclause 7.7.1.1 **The `signal` function:**

> If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler calls any function in the standard library other than the `signal` function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler) or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`.

Since `longjmp` is a function, it cannot be called.

But subclause 7.6.2.1 **The `longjmp` function**, broadly guarantees the opposite:

> As it bypasses the usual function call and return mechanisms, the `longjmp` function shall execute correctly in contexts of interrupts, signals and any of their associated functions.

Suggested Technical Corrigendum:

If the intent is to allow calls to `exit` and `longjmp` from signal handlers not invoked through calls to `raise` or `abort`, replace in subclause 7.6.2.1:

> ... other than the `signal` function itself ...

by:

> ... other than `longjmp`, `exit`, or the `signal` function itself ...

Alternatively, if the intent is to disallow calls to `longjmp` from signal handlers not invoked through calls to `raise` or `abort`, replace in subclause 7.7.1.1:

> As it bypasses the usual function call and return mechanisms, the `longjmp` function shall execute correctly in contexts of interrupts, signals and any of their associated functions. However, if the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

by:

> If the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.