Slaying the Earthly Demon of Significant Characters (UB30)

Document: **n3771**

Author: Glenn Coates - glenn.coates.uk@gmail.com

Date: 2025-12-31

C Standard: ISO/IEC 9899:2024, Information technology — Programming languages — C, 5th edition, International Organization for Standardization / International Electrotechnical Commission, Geneva, 2024.

Undefined Behavior: (30) Two identifiers differ only in nonsignificant characters (6.4.3.1).

Introduction

§6.4.3.1 Implementation limits states:

> "As discussed in 5.3.5.2, an implementation may limit the number of significant initial characters in an identifier; the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.
>
> Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined."

§5.3.5.2 Translation limits requires that:
"The implementation shall be able to translate and execute a program that uses but does not exceed the following limitations for these constructs and entities:

> …
>
> — 63 significant initial characters in an internal identifier or a macro name,
>
> — 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 00FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)14
> …"

Ideally, the distinction between significant and nonsignificant characters in identifiers could be removed, and implementations would instead define the maximum total length of identifiers they support. This approach is explored in paper N3604. In practice, most modern implementations already treat all characters in an identifier as significant and support very long identifier names. However, fully standardising this model is difficult to achieve due to limitations in some existing assembler and linker implementations.

WG14 Direction

At the 73rd meeting of ISO/IEC JTC 1/SC 22/WG14, the author presented two papers addressing Undefined Behaviour J.2(30), Two identifiers differ only in nonsignificant characters:

- Slaying Some Earthly Demons – remove UB 30 – Approach 1 (N3603), which proposed resolving the undefined behaviour by making the case where two identifiers differ only in nonsignificant characters implementation-defined rather than undefined-behavior.

- Slaying Some Earthly Demons – remove UB 30 – Approach 2 (N3604), which proposed removing the concept of significant characters altogether from the specification, so that all characters in an identifier are treated as significant, subject only to an implementation-defined maximum identifier length.

Following discussion of these papers, the committee conducted a straw poll on how to address significant character semantics. The recorded preferences were:
- 7 in favour of taking no action,
- 4 in favour of treating identifiers that differ only in nonsignificant characters as implementation-defined (Approach 1),
- 17 in favour of removing the concept of significant characters altogether (Approach 2).

While a clear majority favoured complete removal of significant character semantics, the discussion also highlighted practical concerns for implementations that delegate linkage to external tools and cannot reliably enforce full character based identifier significance. In particular, adopting full removal would present practical difficulties for implementations such as SDCC, given their reliance on externally supplied assemblers and linkers.

This proposal therefore adopts an intermediate position (referred to here as approach 1.5), intended to maximise consensus across the committee. The compromise retains the existing concept of significant characters, but removes undefined behaviour by requiring a collision based diagnostic rule. Under this rule, if two identifiers differ only in nonsignificant characters and the implementation would otherwise treat them as the same identifier, a diagnostic message is produced.

Background: removing the concept of significant characters

Modern C implementations do not adopt the historical significant character model described by the C standard. In such cases, all characters in an identifier are treated as significant, subject only to large implementation limits.

GCC documents this behaviour explicitly in its identifier implementation notes (https://gcc.gnu.org/onlinedocs/gcc/Identifiers-implementation.html):

> “For internal names, all characters are significant. For external names, the number of significant characters are defined by the linker; for almost all targets, all characters are significant.”

Similarly, Microsoft’s compiler documents a large supported limit for external identifiers (https://learn.microsoft.com/en-us/cpp/build/reference/h-restrict-length-of-external-names?view=msvc-170):

> “By default, the length of external (public) names is 2,047 characters. This is true for C and C++ programs.”

This illustrates that modern implementations already treat identifiers as fully significant with limits far beyond the historical 31 and 63 character minima required by the C standard.

Section §5.2.1.2 defines the entire translation process, from preprocessing through compilation and linkage, as the responsibility of a conforming implementation. In particular, linkage occurs in translation phase 8, where external references are resolved and a complete program image is produced. This places identifier handling during linkage within the scope of the C standard. In practice, however, some implementations delegate phase 8 responsibilities to external assemblers and linkers, often as a consequence of supporting multiple backend targets; SDCC is an example of such an implementation.

It was considered that approach 2 could be accommodated by SDCC by retaining the existing minima in §5.3.5.2 of 63 characters for internal identifiers and macro names and 31 characters for external identifiers but interpreting those values as minimum identifier lengths that must be supported (with implementations free to support longer identifiers) rather than as limits on significant initial characters. Under this model, all characters within the supported range would be treated as significant and any remaining characters would be ignored.

However, this approach breaks down in practice for SDCC, which may invoke different external assemblers and linkers with differing identifier handling rules that are outside of the compiler front end's control. The front end may not reliably know which identifier limits will apply.

Rationale for a conditional diagnostic rule

As discussed in the preceding section, implementations such as SDCC may delegate linkage to different external assemblers and linkers with differing significant-character limits. In addition, the C standard's abstract character accounting rules do not necessarily correspond to the way identifiers are represented or compared by those back ends. This creates a mismatch between the standard's accounting model and backend identifier comparison. As a result, the standard's accounting rules cannot reliably predict how identifiers will be compared at linkage. This motivates a diagnostic rule based on actual identifier collisions rather than on abstract character counts alone.

This mismatch is illustrated by identifiers containing non-ASCII characters such as ü. Under the C standard's accounting rules for external identifiers, such a character is counted according to the length of its corresponding universal character name. §5.3.5.2 Translation limits, states:

> "— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 00FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)14)"

Consequently, an identifier consisting solely of the character ü is considered to have six characters.

In contrast, implementations such as SDCC pass external identifiers to assemblers and linkers that operate on encoded representations. For the character ü, the C standard counts the portable source spelling \u00FC, while a backend may compare a much shorter encoded form, for example two UTF-8 bytes. As shown in Appendix A, an identifier that exceeds the standard's abstract character accounting may therefore remain unambiguous and distinguishable at linkage.

As noted in the previous section, a first difficulty is that multiple backend assemblers and linkers may be used, each with different significant-character limits. This difficulty is compounded by the fact that different back ends may also use differing identifier encodings and comparison rules.

For this reason, the diagnostic requirement must be based on actual implementation behaviour. A diagnostic should be required only when two identifiers that differ only in nonsignificant characters would otherwise be treated as the same identifier by the implementation. This ensures that diagnostics are produced precisely when a real collision would occur, and not merely when an abstract accounting rule is exceeded.

Recommendation

§4 of the standard states: “If a ‘shall’ or ‘shall not’ requirement that appears outside of a constraint or runtime constraint is violated, the behavior is undefined.”

Add a Constraints subsection to §6.4.3.1 requiring a diagnostic when two identifiers differ only in nonsignificant characters and the implementation would otherwise treat them as the same identifier.

As a result, the following entry should be removed from Annex J.2:

> (30) Two identifiers differ only in nonsignificant characters (6.4.3.1).

Rewording

6.4.3 Identifiers
6.4.3.1 General
Syntax
identifier:

```
        identifier-start
        identifier identifier-continue
```

identifier-start:

```
        nondigit
        XID_Start character
        universal character name of class XID_Start
```

identifier-continue:

```
        digit
        nondigit
        XID_Continue character
        universal character name of class XID_Continue
```

nondigit: one of

```
        _ a b c d e f g h i j k l m
        n o p q r s t u v w x y z
        A B C D E F G H I J K L M
        N O P Q R S T U V W X Y Z
```

digit: one of

```
        0 1 2 3 4 5 6 7 8 9
```

Constraints
Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters and the implementation would otherwise treat them as the same identifier, the implementation shall produce a diagnostic message.

Semantics
An XID_Start character is an implementation-defined character whose corresponding code point in ISO/IEC 10646 has the XID_Start property. An XID_Continue character is an implementation-defined character whose corresponding code point in ISO/IEC 10646 has the XID_Continue property. An identifier is a sequence of one identifier start character followed by 0 or more identifier continue characters, which designates one or more entities as described in 6.2.1. It is implementation-defined if a $ (U+0024, DOLLAR SIGN) may be used as a nondigit character. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

Implementation limits
As discussed in 5.3.5.2, an implementation may limit the number of significant initial characters in an identifier; the limit for an external name (an identifier that has external linkage) may be more

restrictive than that for an internal name (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.

~~Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.~~

Forward references: universal character names (6.4.4), macro replacement (6.10.5), reserved library identifiers (7.1.3), use of library functions (7.1.4), attributes (6.7.13.2).
…

Acknowledgments

Many thanks to David Svoboda, Martin Uecker, Chris Bazley, Dave Banham, Philip Krause, and members of the UBSG for helpful discussion and feedback. Thanks also to Peter Smith (Arm) for his insights.

Appendix A — Example False Diagnostic

This appendix demonstrates how the C standard's character accounting rules for external identifiers (§5.3.5.2) can require a diagnostic even when the backend assembler and linker are able to distinguish the identifier without ambiguity. The same identifier is shown written once using an extended source character and once using a universal character name (UCN).

Assumptions (toy example)

Standard significant character limit for external identifiers: 3.

Backend assembler/linker significant prefix: 4 bytes.

External identifiers are mangled by prepending an underscore.

The backend compares identifiers by byte sequence.

Example A1 — Extended Source Character

Source code example:
int ü;

C language model:
The identifier consists of one abstract character. That character is significant.

Standard character accounting (§5.3.5.2):
The character ü corresponds to the universal character name \u00FC.
The universal character name \u00FC specifies a short identifier of 00FFFF or less.
Therefore, it is considered 6 characters.
Since 6 is greater than 3, the identifier exceeds the standard's toy limit.

Backend symbol representation (UTF-8 bytes):
The mangled name is _ü.
The underscore occupies 1 byte.
The character ü occupies 2 bytes.
The total is therefore 3 bytes.
Since 3 is less than 4, the backend can represent and distinguish the symbol.

Result:
A diagnostic based solely on the standard's character accounting would reject this identifier, even though the backend can handle it safely. This is a false positive.

Example A2 — Same Identifier Written Using a UCN

Source code example:
int \u00FC;

C language model:
The identifier denotes the same abstract character as in Example A1.

Standard character accounting (§5.3.5.2):
The universal character name \u00FC is spelled using six source characters.
Therefore, the identifier is again considered 6 characters.
Since 6 is greater than 3, the identifier exceeds the standard's toy limit.

Backend symbol representation:
The backend symbol is identical to Example A1.
The mangled name is _ü.

The UTF-8 byte sequence is the same.
The total length remains 3 bytes.

Result:
The same diagnostic outcome is required, despite identical backend behaviour and identical identifier semantics.