Slaying Some Earthly Demons - remove UB 30 - Approach 1

Document: n3603

Author: Glenn COATES

Date: 2025-07-19

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).

Analysis

§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.”

These are the minimum requirements that a conforming implementation must support. An implementation may support more significant characters, but it is not required to do so.

Undefined behavior arises in the following example:
- Two identifiers in different translation units differ only in characters beyond the number of significant initial characters defined by the implementation.

- The implementation defines only the first 31 characters of external identifiers as significant (e.g., due to linker constraints).

- The implementation treats both identifiers as the same name. As a result, the program contains multiple definitions of the same external identifier.

- Under §6.4.3.1, this currently results in undefined behavior. In practice, most implementations treat the identifiers as equivalent and report a multiple definition error at link time.

Coding standards help mitigate such issues. For example, MISRA C:2023 Rule 5.1 requires that: “External identifiers shall be distinct in the first 31 characters and internal identifiers shall be distinct in the first 63 characters.”

Multiple definition conflicts may occur both within a single translation unit and across translation units. Those involving external linkage are typically detected by the linker, while those involving internal linkage are generally diagnosed at compile time.

§5.2.1.2 defines the entire translation process, from preprocessing to linkage, as the responsibility of a conforming implementation. According to that section, linkage occurs in translation phase 8, where external

references are resolved and a complete program image is produced.

Although linkage is within the scope of the C standard, implementations frequently rely on external linkers and object file formats shared with other programming languages. These components may impose constraints on identifier handling that affect how the implementation fulfils its obligations under the standard.

A wide variety of object file formats are in use, including proprietary and platform-specific formats. A Wikipedia article (https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats) on executable file formats lists over 30 such formats. Compiler toolchains often depend on linkers and binary formats not controlled by the compiler implementor and not developed with input from WG14 participants.

Ideally, the distinction between significant and nonsignificant characters in identifiers would be removed, and implementations would instead define the maximum total length of identifiers they support. This would simplify the specification and reduce the potential for accidental name collisions. However, this is difficult to achieve in practice due to limitations in current linker implementations and object file formats. Some of these are legacy or proprietary systems that are not easily changed.

For these reasons, reclassifying this case as implementation-defined reflects the practical limitations of real-world toolchains while requiring conforming implementations to document their behavior. This improves clarity and predictability for programmers without imposing unrealistic requirements on existing or external linkers.

Recommendation

Change the following in the Implementation limits section of §6.4.3.1, to:
Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is implementation-defined.

As a result, the following entry should be removed from Annex J.2, since the behaviors they describe become diagnosable constraint violations rather than undefined behavior:

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

Rewording

6.4.3 Identifiers
6.4.3.1 General
…
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~~ implementation-defined.

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, Aaron Ballman, Philipp Klaus Krause, Dave Banham and the UBSG.