Slaying Some Earthly Demons - remove UB 30 - Approach 2

Document: n3604

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 states: “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:” and includes:

- “63 significant initial characters in an internal identifier or a macro name,” and
- “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.

The idea of dividing identifier characters into significant and non-significant becomes questionable when, in practice, compilers and linkers treat all characters as significant. If the compiler and linker can tell two identifiers apart, then those characters are clearly being treated as significant.

For GCC, https://gcc.gnu.org/onlinedocs/gcc/Identifiers-implementation.html states: “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.”

For the Microsoft compiler, https://learn.microsoft.com/en-us/cpp/build/reference/h-restrict-length-of-external-names?view=msvc-170 states: “By default, the length of external (public) names is 2,047 characters. This is true for C and C++ programs. Using
/H can only decrease the maximum allowable length of identifiers… You may find /H useful:
When you create mixed-language or portable programs.
When you use tools that impose limits on the length of external identifiers.
When you want to restrict the amount of space that symbols use in a debug build.”

A fundamental approach to address UB #30 is to remove the concept of significant and nonsignificant characters altogether:

- All characters in an identifier (up to an implementation-defined limit) are treated as significant.
- A conforming implementation must compare the entire identifier within the allowed length.
- Length limits remain in §5.3.5.2, which define the minimum number of characters an implementation must support for identifiers. This may be different for internal and external identifiers.

However, it should be noted that such an approach may inadvertently break some programs, for example:

- Programs that rely (perhaps unintentionally) on identifier truncation and UB. Where two identifiers collide after the linker drops characters beyond the significance limit, but the program happens to work anyway.
- Programs that are currently valid because all identifiers are unique within the implementation-defined significant length, e.g. 31 characters. However, identifiers lengths are greater than those supported by the implementation under the new approach.

While linkage is within the scope of the C standard, some implementations depend on proprietary or legacy linkers and object file formats. These linkers may be hard to update and may not follow changes to the C standard. Their developers may not be involved in WG14 and may not be aware of updates to the specification. As a result, requiring changes to such systems could be unrealistic or introduce compatibility risks for existing toolchains and applications.

Recommendation

Terminology referring to “significant initial characters” should be removed from §5.3.5.2 Translation limits and §6.4.3.1 Implementation Limits. Instead, the text should simply refer to the number of characters in an identifier.

The existing limits of 31 characters for external identifiers and 63 for internal identifiers were originally defined as the minimum number of significant characters an implementation had to support. The same values should be reused as the minimum total identifier lengths, which helps preserve compatibility with existing code and older toolchains.

If this proposal is accepted, the following entry should be removed from Annex J.2, because the concept of significant and non-significant characters would no longer exist in the specification:

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

Rewording

5.3.5.2 Translation limits

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:13)

— 127 nesting levels of blocks

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or void type in a declaration

— 63 nesting levels of parenthesized declarators within a full declarator

— 63 nesting levels of parenthesized expressions within a full expression

— 63 ~~significant initial~~ characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)

— 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)

…

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.~~

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.