Slaying Some Earthly Demons - remove UB 30

Document: **n3911**

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

Date: 2026-6-27

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).

Abstract

This paper proposes removing the concept of significant initial characters for identifiers from ISO/IEC 9899 and eliminating Undefined Behaviour J.2(30), Two identifiers differ only in nonsignificant characters.

C's identifier and linkage rules are central to how translation units are combined into a program. The current standard retains a legacy model in which only an initial prefix of each identifier is required to be significant, and behaviour is undefined when identifiers differ only in nonsignificant characters.

This model originated from historical constraints of early linkers and object file formats that imposed short external symbol limits and, in some cases, ignored case. Sections §6.4.3.1 and §5.3.5.2 reflect this model by permitting implementations to compare only a prefix of identifiers and to treat remaining characters as nonsignificant. This behaviour no longer describes mainstream C implementations. Mainstream implementations treat all characters in an identifier as significant, up to implementation-defined and typically generous length limits.

Under the proposed change, all characters in an identifier are significant, the maximum identifier length is implementation-defined, and a diagnostic is required if that maximum is exceeded. The existing numerical limits in §5.3.5.2 (31 characters for external identifiers and 63 for internal identifiers and macro names) are retained, but reinterpreted as minimum total supported identifier lengths rather than significant-prefix counts.

This change removes undefined behaviour, aligns C's identifier semantics with modern implementation practice and with C++.

Current Wording

The following text is reproduced from ISO/IEC 9899:2024 and is the subject of this proposal.

§6.4.3.1 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."

§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:"

"63 significant initial characters in an internal identifier or a macro name,"

“31 significant initial characters in an external identifier.”

Translation Phase 8 and the Linker

§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.

This implies:

> The specification covers the behaviour of the linker as part of the translation process. When an implementation delegates linkage to an external system linker, that linker is still part of the conforming implementation.

Identifier handling during linkage, including name comparison, identifier length limits, and preventing aliasing due to truncation, is within the normative scope of ISO C.

Why the Current Model Is Obsolete

Mainstream toolchains do not implement the significant-prefix model. Mainstream compilers (GCC, Clang, MSVC, IBM XL C, etc.) treat all characters as significant up to their (large) internal limits. Contemporary object formats (ELF, COFF/PE, Mach-O, GOFF, etc.) store full symbol names and do not truncate them.

If a toolchain can distinguish two identifier strings, then in practice all of their characters are significant.

No mainstream implementation:
- truncates identifiers to a fixed significant prefix while ignoring the remainder
- relies on undefined behaviour when identifiers differ only after that prefix

Keeping the significant prefix model:
- gives users an inaccurate impression of actual implementation behaviour
- permits silent aliasing that no known mainstream compiler performs.

GCC[1], 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.”

Microsoft [2], 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, not increase it. 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.”

“If a program contains external names longer than *number*, the extra characters are ignored. If you compile a program without /H and if an identifier contains more than 2,047 characters, the compiler will generate Fatal Error C1064."

Appendix A summarises relevant modern and historical object file formats.

C and C++ consistency

Programs frequently consist of separately compiled C and C++ translation units that are linked together. Consistent identifier semantics between the two languages therefore makes sense.

C++ does not retain C’s model of significant initial characters. Instead, it states that it does not place a translation limit on significant characters for external identifiers. “Extended characters can produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers” [10]. Annex B then states the minimum recommendations for identifiers as: “Number of characters in an internal identifier (5.11) or macro name (15.7) [1 024]” and “Number of characters in an external identifier (5.11, 6.7) [1 024]” [10].

Removing the concept of significant characters in C better aligns the two standards and improves predictability for programs that combine C and C++ translation units.

Modernisation Precedent

There is clear precedent for modernising identifier rules. C90 required only six significant characters and allowed case insensitive external identifiers. “The implementation may further restrict the significance of an external name (an identifier ,that has external linkage) to six characters and may ignore distinctions of alphabetical case for such names” [8].

C99 [9] increased the minima to 31/63 characters and mandated case sensitivity. Eliminating significant prefix rules may be viewed as a natural continuation of this evolution.

Potential Risks

The potential risks if this proposal is adopted include the following:

• Code bases that rely on identifier truncation and undefined behaviour
A code base may rely, intentionally or otherwise, on the current undefined behaviour in which two identifiers differ only in characters beyond the implementation-defined significance limit, yet the program happens to work after truncation. Such code depends on undefined behaviour and is not portable.

• Code bases whose identifiers exceed the newly proposed implementation-defined length limits
A code base could be strictly conforming under the current rules because all identifiers are distinct within the implementation-defined significant length (for example, 31 characters). However, a programs identifier length exceed the maximum supported by an implementation under the new proposed model. In such cases, the proposed new rules would require a diagnostic.

• Implementations whose back ends do not follow the standard’s abstract character counting rules for identifiers
Implementations may invoke different external assemblers and linkers whose identifier handling rules are outside the compiler front end control. These tools may impose different identifier limits, use different encodings for external names, or compare symbols by byte sequence rather than by the character accounting rules used by the standard. As a result, the front end may not be able to determine which identifier limits or comparison rules will ultimately apply at linkage. For example, under §5.3.5.2, a character such as ü is counted according to the length of its corresponding universal character name \u00FC, whereas a backend may compare a shorter encoded form such as a two-byte UTF-8 sequence. In such implementations, replacing significance based rules with a simple maximum length model may require a diagnostic even when the backend can safely distinguish the identifiers. Appendix B gives a simple example of this for illustration purposes.

If an implementation cannot satisfy a particular requirement, that may be noted as a non-conformance, just as with other existing departures from the standard. Where necessary, backend tools can be updated to meet the standard’s requirements.

WG14 Direction

At the 73rd meeting of ISO/IEC JTC 1/SC 22/WG14, the committee considered options for addressing significant-character semantics. The recorded preferences were:

- 7 in favour of taking no action,
- 4 in favour of making the matter implementation-defined,
- 17 in favour of removing significant characters altogether.

This represents a clear majority in favour of removal, which this paper implements.

Why UB #30 should be removed

- This UB is unnecessary for modern toolchains. Mainstream compilers and object file formats treat all characters in an identifier as significant. The UB gives users a misleading picture of real behaviour.

- Silent aliasing can occur under this UB. Two identifiers differing only beyond the significance limit could silently resolve to the same symbol at link time, with no diagnostic or runtime indication. This may lead a a memory safety issue since silent aliasing of identifiers can cause the wrong function to be called or the wrong object to be accessed, potentially corrupting program state in ways that may be difficult to diagnose.

- A program must artificially constrain identifier length to remain portable and safe. Under the current rules, the only standard conforming defence against silent aliasing is keeping all external identifiers within 31 characters and internal identifiers within 63 characters. This may be an issue for automatically generated C code.

- Alignment with C++. C++ places no limit on significant characters for external identifiers. Mixed C/C++ codebases are common, and divergent identifier semantics between the two languages is an unnecessary source of inconsistency.

- Removing UB is in the C Standard charter. “Undefined behaviors, unspecified behaviors, implementation-defined behaviors, and other portability issues enumerated in Annex J of the Standard should be eliminated or reduced. **These issues might lead to application vulnerabilities.**” [11]

Conclusion

The existing model of significant initial characters reflects historical implementation constraints that are no longer representative of mainstream C toolchains. Contemporary compilers and linkers treat all characters in an identifier as significant, up to documented limits, and do not rely on Undefined Behaviour J.2(30) in practice.

This paper proposes to:

- replace significant-prefix semantics with a model in which all identifier characters are significant,
- retain the existing 31/63 character limits as minimum supported identifier lengths,
- require a diagnostic when implementation-defined identifier length limits are exceeded, and
- remove J.2(30), eliminating undefined behaviour.

These changes align the C standard with current implementation practice and with C++ identifier semantics, improve predictability and safety.

The following entry should be removed from Annex J.2, “(30) Two identifiers differ only in nonsignificant characters (6.4.3.1).”

The following entry should also be removed from Annex J.1, “(8) How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.3.5.2).”

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 maximum 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.~~

If an identifier exceeds the implementation's documented maximum, a diagnostic is required.

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).

…

6.11.3 External names

Restriction of the ~~significance~~ maximum length of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

…

J.3.4 Identifiers

(1) Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.3).

~~(2) The number of significant initial characters in an identifier (5.3.5.2, 6.4.3).~~

…

~~**J.5.4 Lengths and cases of identifiers**~~

~~All characters in identifiers (with or without external linkage) are significant (6.4.3).~~

…

Acknowledgments

Many thanks to David Svoboda, Martin Uecker, Chris Bazley, Dave Banham, Philip Krause, Joseph Myers and the UBSG. Thanks to Peter Smith, Arm.

Appendix A — Historical and Identifier Length Limits in Object File Formats

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.

This appendix summarises historical and contemporary object file formats in relation to identifier length limitations.

Historical Object File Formats

In the context of this appendix, “symbol” refers to the representation of an external identifier within an object file or linker symbol table. Limits on symbol names thus correspond directly to limits on external identifiers as processed by linkers and object file formats.

Object File Format	Era	Symbol Length Limit	Notes/Usage
OS/360 [4]	Circa 1965	8 bytes	IBM System/360 and successors Format conventions use uppercase only
IEEE-695 [3]	Circa 1985	Symbol names are stored as length-prefixed strings with the length encoded in a single byte, permitting symbol names of up to 255 characters	
COFF [5]	1980’s - 1990’s	Short names (8 characters or fewer) are stored directly in the symbol table. Longer names are stored as an offset into the string table at the end of the COFF object, making the effective identifier length implementation-defined.	Unix System V, replaced the previously used a.out format. Superseded by ELF on most modern UNIX systems.
Acorn Object Format (AOF) [6]	1980s–1990s	Truncates identifiers for external linkage when exceeding implementation-defined limits.	Used on RISC OS and early ARM toolchains. Modern ARM ELF-based toolchains no longer use AOF.
OMF [7]	Circa 1994	Symbol names are stored as counted (length-prefixed) strings with the length encoded in a single byte, which limits identifier names to a maximum of 255 characters.	IBM OS/2 operating system

Object File Format	Era	Symbol Length Limit	Notes/Usage
a.out	1970s–1980s	In the original a.out object file format, the symbol name field was declared as char n_name[8], limiting external identifier names to 8 characters. Modern BSD variants of a.out replaced the fixed n_name[8] with a string table, so names are stored as offsets and can be of arbitrary length.	Predecessor to COFF and ELF. Obsolete since early UNIX System V and BSD releases.
ELF	1990s–present	Stores symbol names as null terminated strings in a string table and does not impose a fixed limit on identifier length, making the effective maximum implementation-defined	Used by GCC and Clang.
SDCC	1990's - current	79 significant bytes.	

Summary of Practical Limits

1. Modern formats (ELF, COFF) allow unlimited identifiers.
2. Legacy formats with limits (e.g., OS/360's 8-character rule) correspond to platforms and toolchains that predate C90. They may already not support C99 31/63 minima.
3. Linkers are the determining factor for external name significance; modern linkers preserve full symbol names.
4. Compiler vendors such as GCC, MSVC, document that all characters are significant up to generous large bounds.

Appendix B — Example 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 results in 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.

References

[1] GCC Project, Identifiers Implementation, https://gcc.gnu.org/onlinedocs/gcc/Identifiers-implementation.html

[2] Microsoft Corporation, Restrict Length of External Names (/H), https://learn.microsoft.com/en-us/cpp/build/reference/h-restrict-length-of-external-names

[3] IEEE-695 Object File Format, §2.2.3, Orangec Documentation.

[4] OS/360 Object File Format, Wikipedia.

[5] AT&T UNIX System V Documentation, COFF Object Format Internals.

[6] ARM Ltd., Acorn Object Format (AOF), historical specification.

[7] IBM OS/2 16/32-bit Object Module Format (OMF)
and Linear executable Module Format (LX)
Revision 8, https://bitsavers.org/pdf/ibm/pc/os2/OS2_OMF_and_LX_Object_Formats_Revision_8_199406.pdf?utm_source=chatgpt.com

[8] American National Standards Institute, American National Standard for Programming Languages—C, ANSI/ISO 9899:1990 (revision and redesignation of ANSI X3.159-1989), New York, 1990

[9] ISO/IEC JTC 1/SC 22/WG14, Programming Languages — C, Committee Draft — September 7, 2007, ISO/IEC 9899:TC3, WG14/N1256, 2007.

[10] ISO/IEC JTC 1/SC 22/WG21, Working Draft, Standard for Programming Language C++, N4950, 12 May 2026. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/n5046.pdf

[11] ISO/IEC JTC 1/SC 22/WG14, The C Standard charter, N5046, 12 June 2024. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3280.htm