Slaying the Earthly Demons of Invalid Identifier Characters (UB28, UB29)

Document: **n3772**

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

Date: 2025-12-1

Undefined Behavior:
(28) A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges.
(29) The initial character of an identifier is a universal character name designating a digit.

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

Introduction

Section §6.4.3.1 of ISO/IEC 9899:2024 (C23) states:

> "Each character and universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 has the XID_Continue property. The initial character (which can be a universal character name) shall designate a character whose encoding in ISO/IEC 10646 has the XID_Start property. An identifier shall conform to Normalization Form C as specified in ISO/IEC 10646. Annex D provides an overview of the conforming identifiers."

This requirements appear in the Semantics section of the standard. Section §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."

In C17, this structure led directly to undefined behaviour when identifiers violated semantic shall requirements after being accepted by the lexical grammar and is noted in C17 Annex J.2

In C23, however, the same situations cannot be formed at all, because the revised lexical grammar prevents such identifiers from being constructed. As a result, these undefined behaviours are now ghost undefined behaviour: they remain listed, but are unreachable in a conforming C23 program.

Why UB 28 and 29 Existed in C17

In ISO/IEC 9899:2018 (C17) [1], the lexical grammar for identifiers allowed any universal character name without restriction and also permitted implementations to introduce additional implementation-defined characters into identifiers. As a result, the lexical rules accepted identifiers that were rejected by the semantic rules that followed. Section 6.4.2 Identifiers defined the grammar as follows:

6.4.2 Identifiers
6.4.2.1 General
Syntax
identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name

other implementation-defined characters

The production *other implementation-defined characters* allowed implementations to accept characters beyond those defined by the standard. In addition, the grammar permitted a *universal-character-name* to appear as an *identifier-nondigit*, without restriction on the Unicode value it represented.

As a result, the lexical grammar accepted identifiers containing characters that were not permitted by the semantic rules for identifiers. For example, the identifier:

```
\u0030foo   // UCN for digit zero
```

was lexed successfully as an identifier preprocessing-token in C17.

In C17, the identifier grammar was intentionally permissive, with restrictions imposed later by semantic rules. Section 6.4.2.1 Semantics then imposed constraints using shall requirements. In particular, §6.4.2.1p3 states:

> "Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in D.1."

It further states:

> "The initial character shall not be a universal character name designating a character whose encoding falls into one of the ranges specified in D.2."

If these shall requirements are not met, the program's behaviour is undefined under §4. As a result, identifiers containing universal character names that designate disallowed characters could be accepted by the lexical grammar, but would then violate semantic requirements.

This use of shall requirements in the semantic rules, applied after a permissive lexical grammar, is the direct origin of Undefined Behaviour J.2 items (28) and (29) in C17.

In C17, these restrictions are expressed in terms of Unicode code point ranges listed in Annex D (D.1 and D.2), rather than through the Unicode-derived XID_Start and XID_Continue properties introduced in later revisions of the standard. Annex D is reproduced at the end of this paper for reference.

Why UB 28 and 29 Are Ghost UB in C23

C23 introduces a revised identifier grammar in which every component of an identifier must denote a character with the appropriate XID_Start or XID_Continue property. Under this grammar, malformed or invalid universal character names cannot form identifiers. As a result, the situations described by Undefined Behaviour J.2 items (28) and (29) cannot occur in a conforming C23 program.

This can be seen by examining the identifier grammar together with the enclosing lexical grammar rules. Section 6.4.3.1 of ISO/IEC 9899:2024 (C23) defines identifiers as follows:

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

Under this grammar, the initial character of an identifier may only be a nondigit (a letter or underscore), a character with the XID_Start property, or a universal character name designating a character with the XID_Start property. After the first character, subsequent characters may be digits or nondigits, or characters with the XID_Continue property, including via a universal character name designating an XID_Continue character.

Unlike C17, C23 does not permit *other implementation-defined characters* to appear in identifiers. The only implementation-defined extension explicitly permitted is whether $ is treated as a nondigit character. No other characters may appear as the first character of an identifier unless they satisfy the XID_Start requirement.

The surrounding lexical grammar defined in ISO/IEC 9899:2024 §6.4 ensures that this identifier syntax is enforced during preprocessing-token classification and again during conversion to language tokens. Section 6.4.1 defines preprocessing tokens as follows:

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each universal character name that cannot be one of the above
each non-white-space character that cannot be one of the above

Preprocessing tokens are formed during translation phases 1–3, as specified in §5.2.1.2, which states in part:

> “The source file is decomposed into preprocessing tokens and sequences of white-space characters.”

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. These preprocessing tokens are then converted into tokens (including identifiers) in translation phase 7.

During preprocessing-token formation, any universal character name or character that cannot satisfy the productions of identifier-start or identifier-continue is nevertheless classified as a preprocessing token by falling into one of the two catch-all categories defined in §6.4.1:

- each universal character name that cannot be one of the above, or
- each non-white-space character that cannot be one of the above.

Such preprocessing tokens cannot be converted into identifier language tokens.

Section §6.4.1 then imposes the following constraint on the conversion step:

> “Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.”

As a result:
- UCNs that do not designate characters with the required XID properties (UB 28),
- UCNs that designate digits when used as the first character of an identifier (UB 29)

can never form identifier language tokens and therefore cannot trigger undefined behaviour.

For this reason, UB 28 and UB 29 are ghost undefined behaviour in C23. The revised lexical grammar makes it impossible for a conforming program to reach the semantic situations that formerly invoked these undefined behaviours.

WG14 Direction

At the 73rd meeting of ISO/IEC JTC 1/SC 22/WG14, the committee considered how the situations described by Undefined Behaviour J.2 items (28) and (29) should be treated in future revisions of the C standard.

The question explicitly asked whether these cases should become implementation-defined rather than a constraint violation. The recorded straw poll was:
- 14 - Yes - implementation defined
- 1 - No - Constraint
- 12 - Abstain

In C17, the identifier grammar explicitly permitted other *implementation-defined characters*, allowing implementations to extend the set of characters accepted in identifiers, including through universal character names. This extension point no longer exists in C23.

As shown in the preceding section, the C23 grammar strictly defines the characters that may appear in identifiers and prevents malformed universal character names from ever forming identifiers. Such characters are classified by the surrounding lexical grammar and must produce a diagnostic during token conversion in translation phase 7. Consequently, the semantic situations that previously gave rise to UB 28 and UB 29 are unreachable in C23.

The WG14 vote therefore cannot be interpreted as authorising implementations to extend *identifier-start* or *identifier-continue* beyond what is permitted by §6.4.3.1. Rather, the vote reflects a preference not to introduce an additional explicit constraint, while relying on the revised lexical grammar to ensure that malformed identifiers are diagnosed.

C23 provides a complete and closed definition of identifier syntax. Removing UB 28 and UB 29 simply reflects this updated model and does not disadvantage any conforming C23 implementation.

Notes on Identifier Extensions

Section §4 of ISO/IEC 9899:2024 defines the relationship between conforming implementations and strictly conforming programs. The standard does not define a concept of a strictly conforming implementation; strict conformance applies only to programs. An implementation is conforming if it accepts all strictly conforming programs and documents any extensions it provides.

Section §4 explicitly permits extensions and requires that they be documented:

> “An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.”

Section §4 further defines the requirements on strictly conforming programs:

> “A strictly conforming program shall use only those features of the language and library specified in this document.”

And

> “…shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit”.

Accordingly, an implementation may accept additional constructs outside the standard, including additional characters in identifiers such as emoji, as an extension while remaining a conforming implementation. This is provided that all strictly conforming programs continue to be accepted and their behavior is unchanged. Programs that rely on such extensions are conforming programs, but not strictly conforming programs.

Rewrding

…

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

An identifier shall conform to Normalization Form C as specified in ISO/IEC 10646.

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.

The character classes XID_Start and XID_Continue are Derived Core Properties as described by UAX #44.64) ~~Each character and universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 has the XID_Continue property. The initial character (which can be a universal character name) shall designate a character whose encoding in ISO/IEC 10646 has the XID_Start property. An identifier shall conform to Normalization Form C as specified in ISO/IEC 10646.~~ Annex D provides an overview of the conforming identifiers.

…

NOTE 1 Characters or universal character names that do not have the XID_Start or XID_Continue properties cannot match the productions identifier-start or identifier-continue. Such characters therefore form the preprocessing tokens “each universal character name that cannot be one of the above” or “each non-white-space character that cannot be one of the above” (6.4.1), and cannot be converted into identifier tokens.

…

The following entries should be removed from from Annex J.2:

~~(28) A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges.~~

~~(29) The initial character of an identifier is a universal character name designating a digit.~~

References

[1] ISO/IEC 9899:2018, Information technology — Programming languages — C,
4th edition, International Organization for Standardization / International Electrotechnical Commission,
Geneva, 2018.
Available at: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf

Acknowledgments

Many thanks to David Svoboda, Martin Uecker, Chris Bazley, Dave Banham, Joseph Myers and the UBSG.

C17 Annex D

(normative)

Universal character names for identifiers

1 This clause lists the hexadecimal code values that are valid in universal character names in identifiers.

D.1 Ranges of characters allowed

1 00A8, 00AA, 00AD, 00AF, 00B2–00B5, 00B7–00BA, 00BC–00BE, 00C0–00D6, 00D8–00F6, 00F8–00FF
2 0100–167F, 1681–180D, 180F–1FFF
3 200B–200D, 202A–202E, 203F–2040, 2054, 2060–206F
4 2070–218F, 2460–24FF, 2776–2793, 2C00–2DFF, 2E80–2FFF
5 3004–3007, 3021–302F, 3031–303F
6 3040–D7FF
7 F900–FD3D, FD40–FDCF, FDF0–FE44, FE47–FFFD
8 10000–1FFFD, 20000–2FFFD, 30000–3FFFD, 40000–4FFFD, 50000–5FFFD, 60000–6FFFD, 70000–7FFFD, 80000–8FFFD, 90000–9FFFD, A0000–AFFFD, B0000–BFFFD, C0000–CFFFD, D0000–DFFFD, E0000–EFFFD

D.2 Ranges of characters disallowed initially

1 0300–036F, 1DC0–1DFF, 20D0–20FF, FE20–FE2F