P3658R0

Adjust *identifier* following new Unicode recommendations

From: Robin Leroy (eggrobin@unicode.org)
To: ISO/IEC JTC 1/SC 22/WG 21/SG 16
Date: 2025-03-13

1. Proposal

Allow identifiers to be composed from characters with the properties ID_Compat_Math_Start property and characters with the ID_Compat_Math_Continue property, in addition to characters with the XID_Start and XID_Continue properties.

In addition, adopt this proposal as a Defect Report against C++23 and earlier.

2. Rationale

This follows the recommendations in Section 3.1 of Unicode Technical Standard #55, *Unicode Source Code Handling*:

> General-purpose programming languages should extend the identifier definition using the mathematical compatibility notation profile defined in *Section 7.1, Mathematical Compatibility Notation Profile*, of *Unicode Standard Annex #31, Unicode Identifiers and Syntax* [UAX31]. This is because these languages are used in scientific computing, which can benefit from the greater legibility and disambiguation afforded by allowing these additional characters in identifiers.

3. Examples

Identifiers	**C++11–C++20 as originally published**	**After P1949**	**With this proposal**
`Hawaiʻi`, `!nu`, [illegible], `𓋴𓄿𓎛𓏏𓆇`, `ík!aːⁿd̥ă̤`, `f′`, `grad_𝑓`, `𝑥2`, `xⁿ`, `𝔰𝔡`	OK	OK	OK
`𝛁f`, `x²`, `x₂`, `𝜕Ω`	OK	Invalid	OK
`∇f`, `∂Ω`, `C∞`	Invalid	Invalid	OK
`∇Ꞙ`, `⚗`, `🙵✱“”`	OK	Invalid	Invalid

4. Q&A

P1949 was supposed to solve the identifier definition problem once and for all; what happened?

C++11 through C++20 originally had an identifier definition based on Unicode's UAX31-R2, immutable identifiers; this definition is appropriate for languages that require forward as well as backward compatibility, *e.g.*, data interchange formats.

C++23 changed its identifier definition based on P1949 in order to align with the Unicode recommendations for programming languages that require backward but not forward compatibility: UAX31-R1, default identifiers. This allowed it to require normalization, which is only stable on assigned characters, and thus can only be stably required of identifiers consisting of assigned characters. Adopting default identifiers also makes it possible for programming environment to emit additional diagnostics based on security considerations, such as the General Security Profile defined in UTS #39 *Unicode Security Mechanisms*. The change was applied as a defect report to C++20 and earlier.

The recommendations for identifier definitions in Unicode Standard had not foreseen the possibility of changes to requirements. It was expected that a programming language designer would pick an identifier definition which would be appropriate for all time, and that language evolution would not affect this choice. As a result, the Unicode Standard provided no guidance on compatibility when switching from one identifier definition to the other; implementers ran into user complaints resulting from the incompatibility.

Shortly after P1949 was adopted, Unicode started revising its guidance on programming languages, significantly updating UAX #31, *Unicode Identifiers and Syntax*, and published a new UTS #55, *Unicode Source Code Handling*. The question of language evolution was explicitly taken into account in this revision; see Section 3.3 of UTS #55.

C++, a major programming language, had been allowing the use of nearly the entire Unicode code space in identifiers for 10 years; other programming languages followed the same definition, notably Swift. This was an amazing experiment to check whether the original Unicode definition of default identifiers made in 1999 was appropriate. The Unicode Technical Committee was able to inspect large amounts of code written using these identifier definitions to ascertain whether some characters had been missed. It turned out that users had taken advantage of a small number of characters not allowed by default identifiers. The Unicode Technical Committee therefore updated its recommendations to programming languages to allow these additional characters.

How do we know that this proposal fixes the compatibility issues?

It has been deployed in Clang since 2022, and user complaints about this issue have ceased as a result; see llvm/llvm-project#54732#issuecomment-1354447207.

Is this going to keep happening?

Clearly, the incompatibility in C++ is a one-off. The change to Unicode recommendations was informed by a decade of experimentation by C++ and other languages influenced by it. It is in principle possible that usage patterns could change over time, and that some high-profile programming languages would allow users to name identifiers using the entire code space, allowing for a review of usage patterns, but none of that is likely to happen often or quickly.

Was P1949 a mistake?

It fixed a lot more bugs than it created. This proposal is removing the single inconvenience that has been noticed in real life.

Why didn't the Unicode Technical Committee just change XID_Start and XID_Continue? Then we would have picked up these additional characters without even noticing.

XID_Start and XID_Continue are used in identifier definitions far beyond the scope of general-purpose programming languages, such as usernames or identifiers in markup languages. Making the relatively common ID_Compat_Math_* characters identifier characters in general would be more disruptive than beneficial in these contexts. In a markup language, a command followed by a literal superscript 2 would become a different identifier. On social media, a reference to a username with a footnote would become a reference to a different user.

Further, some of the characters in the ID_Compat_Math_* sets have the Pattern_Syntax property, which means that the Unicode stability policy forbids ever adding them to XID_Start and XID_Continue (this prohibition exists to ensure compatibility across versions of Unicode in languages that allow literal text interspersed with identifiers, which is not the case of C++).

Will this pose problems if we want to add user-defined operators at some point in the future?

This has been taken into account when defining the sets of characters with the properties ID_Compat_Math_Start and ID_Compat_Math_Continue. In particular, since Swift, a programming language that allows for user-defined operators, was one of the programming languages that used the old C++ identifier definition, we were able to see what characters were actually used in operators. See Appendix B of document L2/22-102.

Will this result in source code spoofing problems?

It cannot be the goal of the lexical identifier definition to deal with spoofing issues, as solutions to spoofing issues are necessarily unstable. Instead, programming environments can emit warnings based on the mechanisms defined in UTS #39 and UTS #55, including warnings about characters in uncommon use based on Identifier_Status, for use in code bases where spoofing is a concern. Such warnings would flag the ID_Compat_Math_* characters, as they are not included in the General Security Profile. Except for `Hawaiʻi`, all of the identifiers in the table of examples above are disallowed by the General Security profile.

UAX #31 also mentions the possibility of allowing emoji in identifiers. Should we do that?

Unicode Technical Standard #55, which recommends the use of the mathematical compatibility notation profile for general-purpose programming languages, does not recommend the use of the emoji profile.

In its survey of source code written using the old C++ identifier definition, the source code working group found some usage of emoji in source code, but contrary to the characters used in mathematical compatibility notation, there was no evidence that this usage was contrastive or otherwise improved readability; instead it was primarily limited to placeholder identifiers in test code. In addition, parsing emoji is technically much more complicated than the simple Start Continue* of default identifiers, so this would place an unnecessary burden on implementers.

Why does it say Compat in the property name and "mathematical compatibility notation" in the title? Are we just doing that for compatibility with old versions of C++?

Mind the word order: this should be read as the "(mathematical (compatibility notation)) profile", not the "(mathematical notation) (compatibility profile)". The word "compatibility" is here to avoid suggesting that mathematical notation is plain text; instead what is representable is a limited "compatibility notation". Compare the use of "compatibility" in "compatibility decomposable characters": modifier letters (superscripts used in some orthographies and in phonetics, which decompose to their non-superscripted counterparts under NFKC) are being encoded regularly, and these modern additions are not for compatibility with older encodings.

Unicode Technical Standard #55 recommends the use of the mathematical compatibility notation profile in general-purpose programming languages even when they do not have a prior history of allowing these characters. Of course, for those languages like C++ that have such a history, compatibility with older versions is an additional reason to adopt this profile.

5. Wording

In [lex.name], change *identifier-start* and *identifier-continue* as follows:

> *identifier-start:*
>
> > *nondigit*
> >
> > an element of the translation character set with the Unicode property XID_Start or the Unicode property ID_Compat_Math_Start
>
> *identifier-continue:*
>
> > *digit*
> >
> > *nondigit*
> >
> > an element of the translation character set with the Unicode property XID_Continue or the Unicode property ID_Compat_Math_Continue

In [uaxid.def.general], modify paragraph 1 as follows:

> UAX #31 specifies a default syntax for identifiers based on properties from the Unicode Character Database, UAX #44. The general syntax is
>
> > <Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
>
> where <Start> has the XID_Start property, <Continue> has the XID_Continue property, and <Medial> is a list of characters permitted between continue characters. UAX #31 also specifies some standard profiles, including the Mathematical Compatibility Notation profile. For C++ we apply the Mathematical Compatibility Notation profile, and we add the character U+005F LOW LINE, or _, to the set of permitted <Start> characters~~,~~. T~~t~~he <Medial> set is empty~~, and the <Continue> characters are unmodified~~. In the grammar used in UAX #31, this is
>
> > <Identifier> := <Start> <Continue>*
> >
> > <Start> := XID_Start + ID_Compat_Math_Start + U+005F

<Continue> := <Start> + XID_Continue + ID_Compat_Math_Continue

Delete [uaxid.def.rfmt]. This subclause refers to UAX31-R1a, which was removed in Unicode Version 15.1 (2023). U+200D and U+200C are now part of default identifiers, so C++ allows them. This addresses some of the March 2024 notes on CWG issue 2843.

~~**E.2.2 R1a Restricted format characters [uaxid.def.rfmt]**~~

~~If an implementation of UAX #31 wishes to allow format characters such as U+200D ZERO WIDTH JOINER or U+200C ZERO WIDTH NON JOINER it must define a profile allowing them, or describe precisely which combinations are permitted.~~

~~C++ does not allow format characters in identifiers, so this does not apply.~~

6. Acknowledgements

Tom Honermann provided valuable feedback on the organization of this proposal. Corentin Jabot implemented the proposal in Clang in 2022. Pascal Leroy suggested improvements to the structure of the historical background. Jens Maurer suggested many of the questions in the Q&A section.