

Extending C++ with Unicode Mathematical Operators

Steve Downey <sdowney@gmail.com>

Document #: D4345R1
Date: 2026-10-20
Project: Programming Language C++
Audience: SG16, EWG
Source: <https://github.com/steve-downey/backtick-papers/unicode-mathematical-operators.org>
heads/blank-slate-0-gfdeaeb0-dirty

1 Abstract

I propose adding the ability to declare user defined operators, such as operator $\boxplus$, allowing users to write `a  $\boxplus$  b`. The list of allowed operators is drawn from the mathematical operators described in Unicode Technical Report #25, *Unicode Support for Mathematics* (Beeton, Freytag, and Sargent III 2017), with confusables removed. The operators are otherwise normal functions, found by ordinary lookup rules and ordinary overload resolution.

```
constexpr int operator $\boxplus$ (int a, int b) { return a + b; }
static_assert(5  $\boxplus$  7 == 12);

Matrix operator $\otimes$ (Matrix const&, Matrix const&);
a  $\otimes$  b  $\otimes$  c // operator $\otimes$ (operator $\otimes$ (a, b), c)

Vec operator $\otimes$ (Vec const&); // one parameter: unary prefix
ev // operator $\otimes$ (v)
```

2 Revision history

2.1 R1

Rewrite from scratch without technical change.

2.2 R0

Initial Paper

3 Before / After

Before	After
<pre>Matrix mul(Matrix const&, Matrix const&); Matrix kron(Matrix const&, Matrix const&); auto r = mul(kron(a, b), mul(c, d));</pre>	<pre>Matrix operator$\otimes$(Matrix const&, Matrix const&); Matrix operator$\boxtimes$(Matrix const&, Matrix const&); auto r = (a $\boxtimes$ b) $\otimes$ (c $\otimes$ d);</pre>
<pre>Set union_(Set const&, Set const&); Set inter(Set const&, Set const&); auto s = union_(inter(x, y), z);</pre>	<pre>Set operator$\cup$(Set const&, Set const&); Set operator$\cap$(Set const&, Set const&); auto s = (x $\cap$ y) $\cup$ z;</pre>

4 The design

Recognize the mathematical symbols that are not identifier as possible C++ operators when they appear in operator position. Since they are disjoint from identifiers there is no ambiguity. Defer the meaning of the expression to name lookup and overload resolution. For the operators assign a fixed precedence and associativity so that parsing does not need to take into account declarations.

4.1 $x \boxplus y$ is the call

The form $x \boxplus y$ desugars to `operator $\boxplus$ (x,y)` and everything else follows from that. $x \boxplus y$ is `operator $\boxplus$ (x, y)` `ex` is `operator $\boxplus$ (x)` and the member forms are `x.operator $\boxplus$ (y)` and `x.operator $\boxplus$ ()`.

4.2 What this paper changes

An *operator-function-id* may be **operator** followed by a **user-operator token** — a single non-ASCII code point drawn from an enumerated set. Such a function is an ordinary free or member function.

One precedence level A user-operator token after a complete operand is a binary operator at the **user-infix level** (the highest binary level, tighter than `*`, looser than the unary operators), left-associative, with cast-expression operands. Every user-introduced infix operator binds at that one strength, and there is no table.

No fixity declarations Precedence and associativity are fixed.

Position decides fixity at the use Operand position is prefix, post-operand is infix. No lookahead or whitespace rule. No consultation of what declarations are visible during parsing.

Arity decides the form at the declaration, and does not filter uses Two parameters, or one as a member, is infix; one parameter, or none as a member, is prefix. The same convention as `operator-()`.

No built-in candidates There is no built-in meaning to protect, so nothing adds any. $p \boxplus n$ on `int*` is a no-viable-overload error.

Candidates are assembled as for an existing overloaded operator member and non-member together in one set, with ADL.

No postfix form, and no fold operator Fold is not precluded, but not proposed. Postfix is unusual in math notation, and is also not proposed.

4.3 Design Constraints

1. The operator set is disjoint from the identifier character set.
2. The list is derived from Unicode recommendations.

4.4 Relation to P4307R0

Infix named operators are a disjoint proposal. The only shared bit is the proposed *user-infix level* for precedence.

5 The token set

5.1 Deriving it

UAX #31 defines **R3c, "Operator Identifiers"** — introduced by §4.2.1, "User-Defined Operators" — as operator syntax over the characters with the `Pattern_Syntax` property. C++23 already references UAX #31 normatively for identifiers (Downey et al. 2021), so the citation is the same shape to the same document.

This paper pins UAX #31 at **revision 43** (Davis and Leroy 2025), the revision that goes with Unicode 17.0, and the derivation below runs against UCD 17.0 (The Unicode Consortium 2025). Revision 45 (Davis and Leroy 2026), for Unicode 18.0, was published on 2026-09-01, while I was working on this paper. There is no particular reason not to adopt Unicode 18, but because code points can be added to the set we must set a baseline for portable code.

R3c does not hand us a usable set directly. `Pattern_Syntax` includes the ASCII operator characters, every one of which is already a token, a token prefix, or blocked by adjacency. The usable pool is `Pattern_Syntax` minus ASCII.

A code point is a user-operator token if all of:

1. it has `Pattern_Syntax`;
2. it is outside ASCII;
3. it lies in the mathematical and arrow blocks (U+2190–21FF, U+2200–22FF, U+2300–23FF, U+27C0–27EF, U+27F0–27FF, U+2900–297F, U+2980–29FF, U+2A00–2AFF, U+2B00–2BFF);

4. its `General_Category` is `Sm` or `So`, which excludes the paired brackets (`Ps`/`Pe`) as delimiters worth leaving unspent;
5. it is not on the exclusion list below.

Applied once against UCD 17.0, that yields **1,381 code points in 32 contiguous ranges**. The lexer's table is 256 bytes and one binary search, on the non-ASCII path both compilers already walk for extended identifiers. The cost is very small.

This paper proposes to list the allowed code point table explicitly with the algorithm as non-normative.

5.2 Immutable is not closed

An immutable property in Unicode means that an assigned code point will never gain or lose that property. However, that does not mean that newly assigned code points could not be given the property. In particular this means that confusability analysis can not be carried out in advance and must be performed after code point assignment. This is the reason not to just apply the algorithm blindly.

5.3 The exclusion sets

5.3.1 Identifier Characters

TR31 §7.1's mathematical profile earmarks ∂ , ∇ and ∞ as *identifier* characters, and its companion syntax profile removes them from syntactic use.

5.3.2 Confusable with an existing token

A character that looks like `-` and is not `-` should fail to lex.

<code>-</code>	U+2212	(-)
<code>/</code>	U+2215	(/)
<code>/</code>	U+2044	(/)
<code>*</code>	U+2217	(*)
<code> </code>	U+2223	()
<code>:</code>	U+2236	(:)
<code>•</code>	U+2219	(.)
<code>•</code>	U+22C5	(.)
<code>≤</code>	U+2264	(<=)
<code>⇐</code>	U+21D0	(<=)
<code>≥</code>	U+2265	(>=)
<code>⇒</code>	U+21D2	(>=)
<code>↔</code>	U+21D4	(<=>)

5.3.3 Emoji Presentation

The code points inside the blocks that TR31 §7.2's emoji profile carves out: U+231A, U+231B, U+23E9 through U+23EC, U+23F0, U+23F3, U+2B1B, U+2B1C, U+2B50 and U+2B55 — watches, hourglasses, media-control triangles, large squares, a star and a circle. They are named by code point here because I could not get them to render consistently.

5.4 Spelling: the glyph and the UCN

A universal-character-name designating a user-operator code point (including the C++23 named form) forms that operator token. `operator` $\boxtimes$, `operator``\u229E` and `operator``\N{SQUARED PLUS}` are the same declaration. Declare with one, define with another, use with a third, they are emitted identically. `a` `\N{CIRCLED TIMES}` `b` is `a`  $\otimes$  `b`.

6 Placement in Grammar

6.1 One level for all user-introduced infix

```
user-infix-expression:
    cast-expression
    user-infix-expression user-operator cast-expression

unary-expression:
    ...
    user-operator cast-expression
```

Binary user operators occupy one precedence level: the highest binary level, tighter than `*` and looser than the unary operators, left-associative, with cast-expression operands. That is the *user-infix level*, and it is meant to serve all user-introduced infix syntax: it is the same level P4307R0 introduces for the backtick operator.

The precedence rules mean that `-a ▯ -b` is `operator▯(-a, -b)`.

```
-a ▯ -b           // operator▯(-a, -b)
a * b ▯ c         // a * operator▯(b, c)
a ▯ b ▯ c         // operator▯(operator▯(a, b), c)
a ▯ eb           // operator▯(a, operator▯(b))
ea ▯ 2 * eb      // (operator▯(a) ▯ 2) * operator▯(b)
```

6.2 Prefix, and how position decides

Arity determines at the declaration. In use, it is purely positional. Post-operand is infix, in operand position it is prefix. There is no lookahead or whitespace sensitivity.

6.3 Postfix, and why not

`operator++` in prefix form is parsed as a cast-expression, in postfix in postfix-suffix, and `++` has no infix form. This restriction is not available for user defined operators without somehow pre-classifying them, which would mean assigning meaning to them. That is not possible as different fields have already ambiguated them.

6.4 Fold expressions

A user operator is not a fold operator: `(... ▯ N)` is ill-formed.

This is primarily because the existing implementations do not provide a mechanism to extend the fold-expression node beyond the current fixed and closed set. Adding the capability is certainly possible if there is real demand, but should be a separate paper, possibly considering other monoidal operations.

7 Declaring, looking up, desugaring

7.1 Restrictions on Operators

[\[over.oper.general\]](#) imposes five restrictions on an operator function. A user operator inherits only one of them.

class-or-enum parameter does not apply. The rule protects a built-in meaning, and there is none. `constexpr int operator▯(int, int)` is legal, and `5 ▯ 7` finding it is the motivating case.

no default arguments does not apply.

not variadic does not apply, likewise.

the fixed arity table has no entry to consult, so the prefix/infix rule is written out.

not a static member kept, and kept as a *choice*, because the arity rule does not exclude it.

That rule counts operands, so `static S operator▯(S, S)` has two and would be accepted; an implementation must reject it deliberately. The reason is the desugaring: `x ▯ y` is defined to mean exactly one of two spellings, `operator▯(x, y)` or `x.operator▯(y)`, and a static member names neither. On a static member, `x.operator▯(y)` is legal C++ but discards the object expression and passes one argument to a two-parameter function.

Everything else is ordinary: templates, `constexpr`, `= delete`, member and non-member, explicit object parameters.

Dropping [\[over.oper.general\]](#) paragraph 10 has a visible consequence. Given only `constexpr int operator⊞(int a, int b = 1)`, the prefix use `⊞5` is accepted and calls it through the default argument, and adding a genuine prefix overload makes `⊞5` ambiguous in the ordinary way. Filtering candidates by declared arity would fix it. However, it would break the desugaring equivalence below, on which the whole design rests: `⊞5` *is* `operator⊞(5)`. So arity selects the form for *declarations*, and does not filter *uses*.

7.2 One candidate set, and no built-in candidates

`x ⊞ y` assembles member candidates from the left operand’s class and non-member candidates from unqualified lookup and ADL, ranked together in one set.

The normative requirement is that the operator form finds what the explicit call finds:

```
static_assert(__is_same(decltype(x ⊞ y), decltype(operator⊞(x, y))));
```

8 One level, and no fixity declarations

Precedence and associativity are fixed by this paper, at one level, and neither can be declared. Of the rules fixed above this is the one most likely to be argued, so the two alternatives are answered here: a declared fixity is an ODR factory, and a derived one has nothing in Unicode to derive from.

A declared precedence is a semantic property that must travel with the name across headers, modules and translation units. Two translation units disagreeing about `a ⊙ b ⊙ c` is an ODR factory, and the parse of an expression comes to depend on which imports are visible: Haskell’s fixity-import problem, and Swift’s `precedencegroup` conflicts ([“The Swift Programming Language: Advanced Operators,”](#) n.d.). With fixity settled by the standard, the *parse* of an expression depends on nothing but the expression. Only the *meaning* of `operator⊞` travels, and that is ordinary lookup.

9 Implementation experience

This proposal has been implemented in Clang against 23 and the changes ported forward to trunk. There were no particular surprises or difficulty once a new expression node was added.

9.1 Volume

	files	lines
Compiler proper	86	+2035 / −18
Tests	34	+5416
Total	120	+7451 / −18

9.2 Mangling and ABI

The prototype mangles through the Itanium ABI’s existing vendor-extended operator production, which works today, demangles with existing tools, and needs no ABI change. However, it would be better to have the Itanium ABI group provide first-class production keyed by code point and carrying a fixity marker.

The Microsoft ABI has no production for such operators at all, and I declined to invent one at let it escape.

9.3 Formatting

`clang-format` required about forty lines across three files: no new token type, no annotator state, no custom spacing rule, no custom break rule and no style option. A user operator is annotated as a binary or unary operator by position, with no delimiter pairing and no break-suppression rules, and the existing overloaded-operator handling rewrites the annotation after `operator` with no new rule, so `operator ⊞` canonicalizes to `operator⊞` under the same option that governs `operator +`. It also honours `BreakBeforeBinaryOperators`, which a delimiter-pair syntax structurally cannot.

10 Wording

The following wording is pro forma. It is intended to make the grammar change concrete; it is not offered as final CWG wording. Wording is relative to the current working draft.

10.1 [lex.operators]

Add *user-operator* as an alternative of *operator-or-punctuator* in [lex.operators]:

user-operator:
one of the characters in Table X, User-operator characters

[Note 1: Table X will contain the frozen enumeration of 1,381 code points described in 5. A *universal-character-name* designating one of those characters forms the same token by the existing translation rules. — end note]

10.2 [expr.unary.general]

Add an alternative to *unary-expression* in [expr.unary.general]:

unary-expression:
...
user-operator cast-expression

10.3 [expr.mptr.oper]

Modify the grammar of [expr.mptr.oper] paragraph 1:

pm-expression:
cast-expression *user-infix-expression*
pm-expression .* *cast-expression* *user-infix-expression*
pm-expression ->* *cast-expression* *user-infix-expression*

10.4 [expr.user] (new subclause)

Insert a new subclause between [expr.cast] and [expr.mptr.oper]:

User-defined operators [expr.user]

user-infix-expression:
cast-expression
user-infix-expression *user-operator* *cast-expression*

10.5 [over.oper.general]

Modify the *operator-function-id* grammar in [over.oper.general]:

operator-function-id:
operator *operator*
operator *user-operator*

Add after the grammar:

[x] An operator function whose *operator-function-id* contains a *user-operator* is a *user-operator function*.

Modify the first sentence of [over.oper.general] paragraph 7:

An operator function other than a user-operator function shall have at least one function parameter or implicit object parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration.

Modify [over.oper.general] paragraph 10:

An operator function other than a user-operator function cannot have default arguments, except where explicitly stated below. Operator functions other than user-operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of [over.oper].

10.6 [over.match.oper]

Modify the first sentence of [over.match.oper] paragraph 1:

Unless the operator is a *user-operator*, if ~~H~~ no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to [expr.compound].

Modify the first and last sentences of [over.match.oper] paragraph 2:

If the operator is a *user-operator* or either operand has a type that is a class or an enumeration, a user-defined operator function can be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. [...] For a *user-operator*, the evaluations are sequenced as specified for the selected function call. Otherwise, ~~However,~~ the operands are sequenced in the order prescribed for the built-in operator.

Modify the built-in-candidate bullet of [over.match.oper] paragraph 3:

For the operator `,`, the unary operator `&`, ~~or~~ the operator `->`, or a *user-operator*, the built-in candidates set is empty.

10.7 [over.unary] and [over.binary]

In [over.unary], extend the definition of a prefix unary operator function to include a function named by a *user-operator* in the added *unary-expression* production. In [over.binary], extend the definition of a binary operator function to include a function named by a *user-operator* in a *user-infix-expression*. The existing transformations to member and non-member function-call notation then apply unchanged.

Annex A ([gram]) is updated mechanically to match.

11 References

- Beeton, Barbara, Asmus Freytag, and Murray Sargent III. 2017. “Unicode Technical Report #25: Unicode Support for Mathematics (Revision 15).” The Unicode Consortium. <https://www.unicode.org/reports/tr25/tr25-15.pdf>.
- Davis, Mark, and Robin Leroy. 2025. “Unicode Standard Annex #31: Unicode Identifiers and Syntax (Revision 43).” The Unicode Consortium. <https://www.unicode.org/reports/tr31/tr31-43.html>.
- . 2026. “Unicode Standard Annex #31: Unicode Identifiers and Syntax (Revision 45).” The Unicode Consortium. <https://www.unicode.org/reports/tr31/tr31-45.html>.
- Downey, Steve, Zach Laine, Tom Honermann, Peter Bindels, and Jens Maurer. 2021. “C++ Identifier Syntax Using Unicode Standard Annex 31.” WG21. <https://wg21.link/p1949r7>.
- “The Swift Programming Language: Advanced Operators.” n.d. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/advancedoperators/>.
- The Unicode Consortium. 2025. “Unicode Character Database, Version 17.0.0.” <https://www.unicode.org/Public/17.0.0/ucd/>.