Document Number:	P4012R0
Date:	2026-02-23
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LEWG, LWG
Target:	C++26

Value-preserving consteval broadcast to simd::vec

Abstract

The broadcast constructor in the Parallelism 2 TS allowed construction from (`unsigned`) `int`, allowing e.g. `vec<float>() + 1`, which is ill-formed in the CD. This breaks existing code that gets ported from the TS to `std::simd`. The design intent behind `std::simd` was for this to work. However, the understanding in LEWG appeared to be that we can't get this right without constexpr function arguments getting added to the language. This paper shows that a `consteval` constructor overload together with `constexpr` exceptions can resolve the issue for C++26 and is a better solution than constexpr function arguments would be.

Contents

1 CHANGELOG

This paper was originally published as [P3844R2], which additionally contained a re-specification of [simd.math]. [P3844R4], which was voted to plenary by LWG, only contains the [simd.math] changes that make conversion behavior more compatible with the `<cmath>` function overloads. Since there was strong concern over the `consteval` broadcast constructor change in LWG, this part was split into its own paper, separating the controversial and uncontroversial parts.

Changes since LEWG saw this paper as [P3844R1]:

- Remove [simd.math] discussion and wording changes.
- Add discussion of concerns raised in LWG.
- Shorten the discussion of the design space, discussing only what was accepted by LEWG in Kona.
- Move content around to arrive quickly at the change and its consequences.

2 STRAW POLLS

2.1 LEWG @ Kona 2025

Poll: We would like to pursue fixing the issue brought up in "DE-286 29.10.7.2p1–4 [simd.ctor] Add consteval broadcast constructor from constant integer (P3844)" for C++26.

SF	F	N	A	SA
13	5	1	0	0

Poll: Resolve "DE-286 29.10.7.2p1–4 [simd.ctor] Add consteval broadcast constructor from constant integer (P3844)" by adding a consteval broadcast overload for value-preserving conversions, and re-specify in [simd.math] and send to LWG.

SF	F	N	A	SA
3	15	1	0	0

2.2 Suggested polls

Poll: We confirm that the issues raised by LWG are understood and re-confirm P4012R0 for C++26 to resolve DE-286.

SF	F	N	A	SA

Poll: We share LWG's concern and want to take more time to get this right. Remove explicit conversions on broadcast for C++26[1].

SF	F	N	A	SA

1 reverting a part of [P3430R3]

3 THE PROPOSAL

1. Add a `consteval` broadcast constructor overload that is constrained to a minimal set of not-value-preserving types.
2. Reword the constraint of the existing broadcast constructor for correct partial ordering with the new `consteval` overload.
3. The presence of the new overload effectively reverts a small part of "Issue 1" (Section 3) of [P3430R3].

The consequence of this change is summarized in Tony Table 1. To understand why the concepts

before	with P4012R0

`int n = 1;`
`// ------ no change: --------`
`simd::vec<float> x = 1.f;`
`// x = 0x5EAF00D; // ill-formed`
`// x = n; // ill-formed`
`x = static_cast<float>(n); // OK`
`static_assert(constructible_from<V, int>);`
 	
`// ------ different: --------`
`// x = 1; // ill-formed`
`x = static_cast<simd::vec<float>>(n); // OK`
`// pow(x, 3); // ill-formed`
`static_assert(not convertible_to<int, V>);`
`// common_type_t<V, int> y = x; // ill-formed`
	`int n = 1;`
	`// ------ no change: --------`
	`simd::vec<float> x = 1.f;`
	`// x = 0x5EAF00D; // ill-formed`
	`// x = n; // ill-formed`
	`x = static_cast<float>(n); // OK`
	`static_assert(constructible_from<V, int>);`
	
	`// ------ different: --------`
	`x = 1; // OK`
	`// x = static_cast<simd::vec<float>>(n); // ill-formed`
	`pow(x, 3); // OK`
	`static_assert(convertible_to<int, V>); // opposite`
	`common_type_t<V, int> y = x; // V`

~~Tony~~Before/After Table 1: Effect of this paper

work out that way, consider that concepts (and traits) basically only work at the type level (and overload resolution rules). Whether an immediate function is called with an argument that is not a constant expression is simply not part of the consideration. The types match, overload resolution does not discriminate on `consteval` and so the following example comes out seemingly wrong:

```
struct X { consteval X(int) {} };
void f(X);
int g();

static_assert(requires { f(g()); }); // OK
void test()            { f(g()); }   // error: call to consteval 'X::X'
```

The requires expression says `f(g())` is fine, but when we actually use it, it is ill-formed. This is the same "feature" which leads to mischaracterization of `constructible_from` and `convertible_to` with the `basic_vec` `consteval` broadcast constructor.

4 THE CONCERNS

It quickly became apparent that LWG would not feel comfortable voting this paper to plenary. While I had considered all the concerns that were brought up, I did not feel like LEWG had taken the time in Kona to understand these concerns and think it all through. Under these circumstances I don't want to force a contested vote in plenary but rather make sure this is well understood.

The concerns raised in LWG:

1. Novel (no other type in the standard library does this[2]).
2. Constrained code can be ill-formed, because the traits/concepts lie.
3. Immediate-escalation could force more code to be immediate than was intended by the user.
4. Errors after immediate-escalation are potentially harder to understand.

5 MOTIVATION

It is very common in floating-point code to simply write e.g. `* 2` rather than `* 2.f` when multiplying a `float` with a constant:

```
float f(float x) { return x * 2; } // converts 2 to float (at compile time)

float g(float x) { return x * 2.; } // converts x to double (at run time)

float h(float x) { return x * 2.f; } // no conversions
```

More importantly, using `* 2` works reliably in generic code, where the type of `x` could be any arithmetic type.

Since this is so common, `std::experimental::simd<T>` made an exception for `int` in the broadcast constructor to not require value-preserving conversions. Consequently, the TS behavior is:

```
using floatv = std::experimental::native_simd<float>;

floatv f(floatv x) { return x * 2; } // converts 2 to float and broadcasts (at
                                     //                                compile time)
floatv g(floatv x) { return x * 2.; } // ill-formed

floatv h(floatv x) { return x * 2.f; } // broadcasts 2.f to floatv
```

When porting existing code written against the TS to C++26, the first step is to adjust the types:

```
using floatv = std::experimental::native_simdsimd::vec<float>;
```

Except for uses of `std::experimental::where`, which need to be refactored to use `simd::select`, the remaining code should work. The one place where it doesn't work is code such as in function `f`, where 2 needs to be replaced:

```
floatv f(floatv x) { return x * 2std::cw<2>; }
```

Since we don't have `constexpr` function arguments in the language, `std::simd` works around it by recognizing *`integral-constant-like`* / `constant-wrapper-like` types, that encode a *value* into a type. This, however, comes at a compile-time cost. Every different value leads to a template specialization of both `constant_wrapper` and a `basic_vec` broadcast constructor (with it's helper types/concepts to determine whether the specialization is allowed). Consequently, for `vec<float>`, I would recommend to always use an `f` suffix rather than `std::cw`.

But that solution is fairly limited, since we don't have literals for 8-bit and 16-bit integers in the language. A function template like

2 `format` strings are very close, though: `https://compiler-explorer.com/z/7z8Y5ooj3`

```
template <simd_integral V>
V f(V x) {
  return x + 1; // ill-formed for V::value_type = (u)int8_t, (un)int16_t, and uint32_t
}
```

needs to use `x + V(1)`[3]. A clever user might write `x + '\1'` instead. But that fails for the `char` type with different signedness.

Consequently, users would need to get used to writing explicit conversions for the constants they use in `std::simd` code. That's not only verbose and ugly, it is also error-prone. Whenever we coerce our users into writing explicit conversions, then value-changing conversions cannot be diagnosed as erroneous anymore. An explicit `static_cast<uint64_t>(-1)` means `0xffff'ffff'ffff'ffff`, whereas `uint64_t x = -1` could have been intended to mean `0x0000'0000'ffff'ffff` or is a result of a logic flaw in the code. E.g., GCC's `-Wsign-conversion` diagnoses the latter, but not the former[4].

If, with C++26, our users are starting to explicitly convert their `int` constants to `basic_vec`, then the interface of `basic_vec` is at least in part guilty for introducing harder to find bugs.

Tony Table 2 presents an example of the solution[5]. Note that the code on the left will never warn about the value-changing conversion, even with all conversion related warnings enabled. This is due to the explicit conversion, which is telling the compiler "I know what I'm doing; no need to warn me about it".

before

```
template <simd_floating_point V>
V f(V x) {
  return x + V(0x5EAF00D);
}

f(vec<double>()); // OK

// compiles but adds 99282960 instead of 99282957
f(vec<float>());

// compiles but adds infinity instead of 99282957
f(vec<std::float16_t>());
```

with P4012R0

```
template <simd_floating_point V>
V f(V x) {
  return x + 0x5EAF00D;
}

f(vec<double>()); // OK

// ill-formed: value-changing conversion
f(vec<float>());

// ill-formed: value-changing conversion
f(vec<std::float16_t>());
```

~~Tony~~Before/After Table 2: Add an offset

A safer implementation of the code on the left side of Tony Table 2 (without this paper) would have been to write `x + std::cw<0x5EAF00D>` instead. Then the value-changing conversion would have resulted in a constraint failure on the broadcast constructor. However, `V(0x5EAF00D)` is shorter and needs fewer template instantations. I expect most users (including myself) will/do not use `std::cw` all over the place.

6 DESIGN SPACE

In the design review of P1928 of this issue of the broadcast constructor it was overlooked (and never discussed) that a `consteval` overload of the broadcast constructor could solve this problem. Before

3 explicit conversion to `basic_vec` allows conversions that are not value-preserving

4 And that's useful, because the former says "I'm intentionally doing this conversion, no need to warn."

5 I got bitten by this in my `std::simd` unit tests

`constexpr` exceptions, we would have worded it to be ill-formed (by unspecified means) if the value changes on conversion to the `basic_vec`'s value-type. Now that we have `constexpr` exceptions, we can specify a `consteval` broadcast overload that throws on value-changing conversion. If the caller cares, the exception can even be handled at compile time. (I believe it should not throw in C++26, for a minimal change to the WD this late in the C++26 cycle.)

Ordering the overloads for overload resolution is tricky, which is another reason why we should consider this issue before C++26 ships and potentially take action even if we don't add a `consteval` overload. Overload resolution does not take `consteval` into account. The process of finding candidate functions ([over.match.funcs.general]), however, does remove explicit constructors from the candidate set if the context does not allow the explicit constructor to be called.

6.1 potentially-convertible-to

R0 proposed to allow any conversion from arithmetic type `U`, that satisfies `convertible_to<value_type>` and does not satisfy *`value-preserving-convertible-to`*`<value_type>` via the `consteval` broadcast constructor. This was too broad, since it would lead to `vec<float>() + 1.5` being valid (of type `vec<float>`). While technically not wrong (no loss on conversion from `1.5`), it is too surprising that an operation involving a `double` operand is evaluated in single precision.

Therefore, R1 of this paper tightens the constraints for the `consteval` broadcast to producing a less surprising common type. If the given constant is of arithmetic type `T`, then we now require `common_type_t<T, value_type>` to be `value_type`. Since `common_type_t<double, float>` is `double`, the expression `vec<float>() + 1.5` becomes ill-formed. However, this rule alone still breaks the case of `vec<short>() + 1`, which the user cannot changed to use a `short` literal (because we don't have one). The TS made an explicit exception for `int` and `unsigned int`[6], which is what we still need for integer types (with lower rank than `int`).

So the final *`potentially-convertible-to`* constraint looks like this:

```
template <typename From, typename To>
  concept potentially-convertible-to = is_arithmetic_v<From>
    && convertible_to<From, To> && !value-preserving-convertible-to<From, To>
    && (is_same_v<common_type_t<From, To>, To>
      || (is_same_v<From, int> && is_integral_v<To>)
      || (is_same_v<From, unsigned> && unsigned_integral<To>));
```

6.2 status quo

The following code shows the properties of the current broadcast constructor. See Appendix A for the definition of the `really_convertible_to` concept.

6 The TS broadcast constructor has a constraint "[...], or `From` is `int`, or `From` is `unsigned int` and `value_type` is an unsigned integral type."

```
using V = simd::vec<float>;

template <typename T> struct X { explicit operator T() const; };

template <typename... Ts>
concept has_common_type = requires { typename std::common_type_t<Ts...>; };

static_assert(not   std::convertible_to<X<float>, V>);
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(not   std::convertible_to<int, V>);
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, double>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

static_assert(not has_common_type<V, double>);
static_assert(not has_common_type<V, int>);

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // ill-formed
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // OK
  x = m;      // OK
  x = l;      // OK (because convertible_to<decltype(l), float> is true)
  x = f;      // OK
  x = 1.1;    // ill-formed
  x = V(1.1); // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

6.3 The proposed solution

The new `consteval` constructor is more constrained than the existing constructor (using subsumption). Thus, the `consteval` constructor is always chosen if the conversion of the given (arithmetic) type to `value_type` is *`potentially-convertible-to`*. Otherwise, the `constexpr` overload is used. As a consequence, some explicit conversions from arithmetic types to `basic_vec` do not work anymore.

Sketch:

```
template <class From, class To>
  concept simd-consteval-broadcast-arg =
    explicitly-convertible-to<From, To> && potentially-convertible-to<remove_cvref_t<From>, To>;

template <class T>
```

```
class basic_vec
{
public:
  template <explicitly-convertible-to<T> U>
    constexpr explicit(see below) basic_vec(U&&); // #1
  template <simd-consteval-broadcast-arg<T> U>
    consteval basic_vec(U&&); // #2
};
```

Now, every explicit call to the broadcast constructor with a type `U` that satisfies *`potentially-convertible-to`*<T> is equivalent to an implicit conversion, since the `consteval` overload is viable and more constrained. Every type with a value-preserving conversion to `T` will select #1 (because of the constraint on #2). Every non-arithmetic type (notably, user-defined types with conversion operator to some arithmetic type) will continue to work as today, since #2 is not viable.

```
static_assert(not   std::convertible_to<X<float>, V>); // equal to status quo / different to above
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(      std::convertible_to<int, V>);      // different to status quo / equal to above
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, double>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

static_assert(not has_common_type<V, double>);
static_assert(    has_common_type<V, int>); // it's vec<float>

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // OK (different to status quo / equal to above)
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // ill-formed (different to both)
  x = m;      // OK
  x = V(l);   // OK
  x = f;      // OK
  x = 1.1;    // ill-formed
  x = V(1.1); // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

6.4 How to handle bad value-preserving casts

The `consteval` broadcast overload needs to be ill-formed if the argument value cannot be converted to the value type without changing the value. This can be achieved via the mechanism used in ([simd.bit]) for `bit_ceil`. The constructor would spell out a precondition followed by *Remarks:* An

expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

The alternative that was mentioned before is to throw an exception (at compile time). Since in virtually all cases, such an exception would not be caught at compile time, the program becomes ill-formed. The ability to catch the exception allowed me to hack up a `really_convertible_to` concept. But otherwise, the utility of using an exception here seems fairly limited. The main reason for using an exception is better diagnostics on ill-formed programs. If we decide to add the `consteval` constructor for C++26, then we might want to delay the new exception type for C++29, though.

7 DIFFERENCES

Differences between the status quo and the two alternatives above:

	status quo	Section 6.3
`convertible_to<int, V>`	`false`	`true`
`common_type_t<V, int>`	✘	`V`
`x = 1;`	✘	✔
`x = V(n);`	✔	✘

Note that while some values of constant expressions of type `int` are convertible to `vec<float>`, it is not true in general that `int` is convertible to `vec<float>`.

8 IMPLEMENTATION EXPERIENCE

Both solutions (and a lot more variants that were discarded) have been implemented and tested in my implementation. Several days (if not weeks) of exploration and testing went into this paper. I implemented the `consteval` overloads for a complete set of vectorizable types with an ability to select between the different behaviors discussed in [P3844R1]. A representative set of [simd.math] is implemented.

9 RECOMMENDATION

My recommendation continues to be to add the `consteval` broadcast overload for C++26. However, we do have the option of opening up the design space for doing something in C++29. I acknowledge that the concerns are troubling issues. Maybe time is all we need to get comfortable. Maybe something else can be done that we don't think of yet. Therefore, to keep the design space open for C++29, we can remove explicit conversions on broadcast. Explicit conversion on broadcast is not an important feature to have, in my opinion, and removing it gives us the necessary breathing room for C++29.

10 WORDING FOR CONSTEVAL BROADCAST

10.1 FEATURE TEST MACRO

In [version.syn] bump the `__cpp_lib_simd` version.

10.2 Modify [simd.expos]

In [simd.expos], insert:

[simd.expos]

```
template<class From, class To>
  concept simd-consteval-broadcast-arg = see below;                    // exposition only

template<class V, class T> using make-compatible-simd-t = see below; // exposition only

template<class V>
  concept simd-vec-type =                                              // exposition only
    same_as<V, basic_vec<typename V::value_type, typename V::abi_type>> &&
    is_default_constructible_v<V>;
```

10.3 Modify [simd.expos.defn]

In [simd.expos.defn], insert:

[simd.expos.defn]

```
template<class From, class To> concept simd-consteval-broadcast-arg = see below;
```

-?- *simd-consteval-broadcast-arg* subsumes *explicitly-convertible-to*.

-?- From satisfies *simd-consteval-broadcast-arg*<To> only if

- remove_cvref_t<From> is an arithmetic type,
- From satisfies convertible_to<To>,
- the conversion from remove_cvref_t<From> to To is not value-preserving, and
- either
  - common_type_t<From, To> is To,
  - To is integral and remove_cvref_t<From> is int, or
  - To satisfies unsigned_integral and remove_cvref_t<From> is unsigned int.

10.4 Modify [simd.overview]

In [simd.overview], change:

[simd.overview]

```
// ([simd.ctor]), basic_vec constructors
template<~~class~~explicitly-convertible-to<value_type> U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
template<simd-consteval-broadcast-arg<value_type> U>
  consteval basic_vec(U&& x)
template<class U, class UAbi>
  constexpr explicit(see below) basic_vec(const basic_vec<U, UAbi>&) noexcept;
```

10.5 modify [simd.ctor]

In [simd.ctore], change:

[simd.ctor]

```
template<class explicitly-convertible-to<value_type> U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
```

1 Let `From` denote the type `remove_cvref_t<U>`.

2 ~~*Constraints*: U satisfies *explicitly-convertible-to*<value_type>.~~

3 *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

4 *Remarks:* The expression inside `explicit` evaluates to `false` if and only if `U` satisfies `convertible_to<value_type>`, and either

- `From` is not an arithmetic type and does not satisfy *constexpr-wrapper-like*,
- `From` is an arithmetic type and the conversion from `From` to `value_type` is value-preserving ([simd.general]), or
- `From` satisfies *constexpr-wrapper-like*, `remove_cvref_t<decltype(From::value)>` is an arithmetic type, and `From::value` is representable by `value_type`.

```
template<simd-consteval-broadcast-arg<value_type> U> consteval basic_vec(U&& x)
```

-?- *Preconditions*: The value of `x` is equal to the value of `x` after conversion to `value_type`.

-?- *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

-?- *Remarks:* An expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

11 Wording for removing explicit conversion on broadcast

11.1 feature test macro

In [version.syn] bump the `__cpp_lib_simd` version.

11.2 modify [simd.overview]

In [simd.overview], change:

[simd.overview]

```
// ([simd.ctor]), basic_vec constructors
template<class U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
template<class U, class UAbi>
  constexpr explicit(see below) basic_vec(const basic_vec<U, UAbi>&) noexcept;
```

11.3 MODIFY [SIMD.CTOR]

In [simd.ctore], change:

[simd.ctor]

```
template<class U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
```

5 Let `From` denote the type `remove_cvref_t<U>`.

6 *Constraints*: `U` satisfies ~~*explicitly-convertible-to*<value_type>.~~ `convertible_to<value_type>`, and either

- `From` is not an arithmetic type and does not satisfy *constexpr-wrapper-like*,
- `From` is an arithmetic type and the conversion from `From` to `value_type` is value-preserving ([simd.general]), or
- `From` satisfies *constexpr-wrapper-like*, `remove_cvref_t<decltype(From::value)>` is an arithmetic type, and `From::value` is representable by `value_type`.

7 *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

8 ~~*Remarks:* The expression inside `explicit` evaluates to `false` if and only if `U` satisfies `convertible_to<value_type>`, and either~~

- ~~`From` is not an arithmetic type and does not satisfy *constexpr-wrapper-like*,~~
- ~~`From` is an arithmetic type and the conversion from `From` to `value_type` is value-preserving ([simd.general]), or~~
- ~~`From` satisfies *constexpr-wrapper-like*, `remove_cvref_t<decltype(From::value)>` is an arithmetic type, and `From::value` is representable by `value_type`.~~

A REALLY_CONVERTIBLE_TO DEFINITION

```
template <typename To, typename From>
  consteval bool converting_limits_throws()
  {
    try {
      using L = std::numeric_limits<From>;
      [[maybe_unused]] To x = L::max();
      x = L::min();
      x = L::lowest();
    } catch(...) {
      return true;
    }
    return false;
  }

template <typename From, typename To>
  concept really_convertible_to = std::convertible_to<From, To>
                                  and not converting_limits_throws<To, From>();
```

B BIBLIOGRAPHY

[P3844R2] Matthias Kretz. *Restore `simd::vec` broadcast from `int`*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3844r2.

[P3844R1] Matthias Kretz. *Restore `simd::vec` broadcast from `int`*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3844r1.

[P3844R4] Matthias Kretz. *Reword [simd.math] for consteval conversions*. ISO/IEC C++ Standards Committee Paper. 2026. URL: https://wg21.link/p3844r4.

[P3430R3] Matthias Kretz. *simd issues: explicit, unsequenced, identity-element position, and members of disabled simd*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3430r3.