Document Number: P3430R3
Date: 2025-02-13
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LWG
Target: C++26

SIMD ISSUES: EXPLICIT, UNSEQUENCED, IDENTITY-ELEMENT POSITION, AND MEMBERS OF DISABLED SIMD

ABSTRACT

This paper collects all issues that came up in LWG review of P1928 (merge `std::simd`), which require LEWG approval.

CONTENTS

1 CHANGELOG

1.1 CHANGES FROM REVISION 0

Previous revision: P3430R0

- Add Wrocław LEWG on Mon poll results.
- Ask for making hidden friend compound assignment operators members instead.

1.2 CHANGES FROM REVISION 1

Previous revision: P3430R1

- Improve prose in Section **??**.
- Found no motivating example for Issue 4 and turned it into a Non-Issue instead.
- Add Wording section that combines all outstanding changes of the paper.
- Bump feature test macro?

2 STRAW POLLS

2.1 LEWG AT WROCŁAW 2024

Poll: Remove wording that unconditionally allows calls to `gen` from the generator constructors to be unsequenced with respect to each other. At the same time, remove `noexcept` from the constructors. (P3430R0 Section 4)

SF	F	N	A	SA
2	9	0	0	0

Poll: Reorder `binary_op` and `identity_element` as suggested by LWG and implemented in P1928R12.

SF	F	N	A	SA
3	11	0	0	0

3 ISSUE 1: EXPLICIT

`simd` has 7 constructors and one conversion operator:

default constructor	not `explicit`	
copy constructor	not `explicit`	
broadcast constructor	not `explicit`, ill-formed when not value-preserving	← reconsider!
conversion constructor	conditionally `explicit`: depends on participating value types	
generator construtor	`explicit`	
load constructors	`explicit`	
Recommended practice: conversion constructor from *implementation-defined* set of types (intrinsics / vector builtin)	`explicit`	← reconsider!
Recommended practice: conversion operator to *implementation-defined* set of types (intrinsics / vector builtin)	`explicit`	← reconsider!

3.1 BROADCAST CONSTRUCTOR

The authors do not recall that moving the constraint of the broadcast constructor to a conditional `explicit` was considered in LEWG. The behavior of broadcast and `basic_simd` conversion constructors is currently inconsistent. One allows conversions that are not value-preserving, via explicit constructor / `static_cast`. The other does not. We recommend that the broadcast constructor is changed to be conditionally `explicit`:

```
template<class U>
  constexpr explicit(see below) basic_simd(U&& x) noexcept;
```

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

2 *Constraints*: `value_type` satisfies `constructible_from<U>`. ~~`From` satisfies `convertible_to<value_type>`, and either~~

- ~~`From` is an arithmetic type and the conversion from `From` to `value_type` is value-preserving ([simd.general]), or~~

- ~~From is not an arithmetic type and does not satisfy *constexpr-wrapper-like*, or~~
- ~~From satisfies *constexpr-wrapper-like* ([simd.syn]) remove_const_t<decltype(From::value)> is an arithmetic type, and From::value is representable by value_type.~~

3 *Effects*: Initializes each element to ~~the value of the argument after conversion to value_type~~`value_type(forward<U>(x))`.

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

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

---

before

```cpp
using floatv = std::simd<float>;

void f(floatv x)
{
  x + 2; // ill-formed
  x + float(2); // OK
  x + floatv(2); // ill-formed

  x = 2 // ill-formed
  x = float(2) // OK
  x = floatv(2) // ill-formed
}
```

with P3430R3

```cpp
using floatv = std::simd<float>;

void f(floatv x)
{
  x + 2; // ill-formed
  x + float(2); // OK
  x + floatv(2); // OK

  x = 2 // ill-formed
  x = float(2) // OK
  x = floatv(2) // OK
}
```

~~Tony~~Before/After Table 1: Make explicit conversions more consistent

3.2 CONVERSION FROM/TO INTRINSIC

The policy draft on `explicit` says "Implicit conversions should exist only between types that are fundamentally the same". The intrinsic types and vector builtin types implemented as extensions in basically every compiler are "fundamentally the same" as the `simd` types of equal value type and width. Consequently, we should consider implicit conversions. The reason for the current wording to say `explicit` still stems from the TS design which deliberately wanted to err on the "too strict" side[1]. This choice was never reconsidered while merging the TS wording to the IS.

1 that wasn't my preference, but guidance from WG21 at the time

---

3 *Recommended practice*: Implementations should enable ~~explicit~~implicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class `basic_simd`:

```
constexpr ~~explicit ~~operator implementation-defined() const;
constexpr ~~explicit ~~basic_simd(const implementation-defined& init);
```

[*Example:* Consider an implementation that supports the type `__vec4f` and the function `__vec4f _vec4f_addsub(__vec4f, __vec4f)` for the architecture of the execution environment. A user may require the use of `_vec4f_addsub` for maximum performance and thus writes:

```
using V = basic_simd<float, simd_abi::__simd128>;
V addsub(V a, V b) {
  return ~~static_cast<V>(~~_vec4f_addsub(~~static_cast<__vec4f>(~~a~~)~~, ~~static_cast<__vec4f>(~~b~~)~~));
}
```

— *end example*]

---

before	with P3430R3

`void f(std::simd<int, 4> x)`
`{`
`  x = static_cast<std::simd<int, 4>>(`
`    _mm_add_epi32(static_cast<__m128i>(x),`
`                  static_cast<__m128i>(x)));`
`}`
	`void f(std::simd<int, 4> x)`
	`{`
	`  x = _mm_add_epi32(x, x);`
	`}`

~~Tony~~Before/After Table 2: Calling an SSE intrinsic

3.3 SUGGESTED POLLS

Poll: Make the broadcast constructor conditionally `explicit` (P3430R3 Section 3.1)

SF	F	N	A	SA

Poll: Make conversions to/from implementation-defined vector types implicit (strike `explicit`) (P3430R3 Section 3.2)

SF	F	N	A	SA

4 ISSUE 2: DROP "UNSEQUENCED" FROM GENERATOR CTOR

(This has already been resolved in Wrocław.)

The current wording for the generator constructors (`basic_simd` and `basic_simd_mask`) says:

The calls to `gen` are unsequenced with respect to each other. Vectorization-unsafe ([algorithms.parallel.defns]) standard library functions may not be invoked by `gen`.

To the authors knowledge this has never been explicitly implemented. Yes, compilers can relatively easily vectorize generator constructor calls, but that doesn't require this wording. In other words, there is no need to restrict user code for the cases where we expect vectorization.

On the other hand, this requirement on user code is likely to be violated in practice. However, as long as implementations implement the broadcast constructor as an unrolled loop over all calls, the UB will never materialize. Unless, at some point in the future an implementation can annotate its unrolled loop with the necessary "unsequenced" property. Suddenly latent bugs would materialize.

Furthermore, the current restriction disallows legitimate use cases, such as calling a random number generator/distribution, performing potentially blocking/synchronizing calls, throwing an exception, or `std::print` debugging.

Therefore, we propose to remove the requirement on the user code and at the same time drop `noexcept` (because throwing from the callable is a valid strategy for error handling).

If we ever find the need for a function that generates `simd` objects from unsequenced calls to scalar functions we can add a named function to do so. The name of such a function could help to indicate unsequenced execution, which helps in code reviews to catch potential issues.

[simd.ctor]

`template<class G> constexpr explicit basic_simd(G&& gen)` ~~`noexcept`~~`;`

7 Let $\texttt{From}_i$ denote the type `decltype(gen(integral_constant<`*`simd-size-type`*`, `*`i`*`>()))`.

8 *Constraints*: $\texttt{From}_i$ satisfies `convertible_to<value_type>` for all i in the range of `[0, size())`. In addition, for all i in the range of `[0, size())`, if $\texttt{From}_i$ is an arithmetic type, conversion from $\texttt{From}_i$ to `value_type` is value-preserving.

9 *Effects*: Initializes the i^{th} element with `static_cast<value_type>(gen(integral_constant<`*`simd-size-type`*`, i>()))` for all i in the range of `[0, size())`.

10 ~~The calls to gen are unsequenced with respect to each other. Vectorization-unsafe ([algorithms.parallel.defns]) standard library functions may not be invoked by gen.~~ `gen` is invoked exactly once for each i.

[simd.mask.ctor]

`template<class G> constexpr explicit basic_simd_mask(G&& gen)` ~~`noexcept`~~`;`

4 *Constraints*: `static_cast<bool>(gen(integral_constant<`*`simd-size-type`*`, i>()))` is well-formed for all i in the range of `[0, size())`.

5 *Effects*: Initializes the i^{th} element with `gen(integral_constant<`*`simd-size-type`*`, i>())` for all i in the range of `[0, size())`.

6 ~~The calls to gen are unsequenced with respect to each other. Vectorization-unsafe ([algorithms.parallel.defns]) standard library functions may not be invoked by gen.~~ `gen` is invoked exactly once for each i.

5 ISSUE 3: REORDER IDENTITY_ELEMENT AND BINARY_OP ON REDUCE

(This has already been resolved in Wrocław and is already part of the working draft.)

The masked `std::reduce` overloads for `simd` require an identity element (for efficient implementation[2]). The value of the identity element is know for all vectorizable types and if the `BinaryOperation` is one of `std::plus<>`, `std::multiplies<>`, `std::bit_and<>`, `std::bit_or<>`, or `std::bit_xor<>`. For every other user-defined binary operation, the caller must provide a value for the identity element:

P1928R11

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(
    const basic_simd<T, Abi>& x, const typename basic_simd<T, Abi>::mask_type& mask,
    type_identity_t<T> identity_element, BinaryOperation binary_op)
```

The original `reduce` overload for the TS was modeled after the overloads that provide an *initial value*: `reduce(InputIt first, InputIt last, T init, BinaryOp op)`. For these functions the `init` parameter precedes the `BinaryOp` parameter.

However, the initial value is a very different parameter: It provides an additional value that is included in the reduction together with the given range. This is not the case for the `simd` overload, where the identity element is included 0-`simd::size()` times in the reduction. More importantly, the value must be such that it doesn't influence the result, otherwise it violates a precondition of `reduce`.

2 The basic idea is to fill all masked elements of the given `simd` object with the identity element and then perform a tree reduction over all elements of the `simd`.

Because of this different nature of the parameter, and because we can provide a default for known binary operations, the `identity_element` parameter can and should be after the `BinaryOp`. Then the 6 overloads for masked reductions are reduced to a single overload of the form:

P1928R12

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(
    const basic_simd<T, Abi>& x, const typename basic_simd<T, Abi>::mask_type& mask,
    BinaryOperation binary_op = {}, type_identity_t<T> identity_element = see below);
```

6 *Constraints*:

- `BinaryOperation` models *`reduction-binary-operation`*`<T>`.
- An argument for `identity_element` is provided for the invocation, unless `BinaryOperation` is one of `plus<>`, `multiplies<>`, `bit_and<>`, `bit_or<>`, or `bit_xor<>`.

7 *Preconditions*:

- `binary_op` does not modify `x`.
- For all finite values `y` representable by `T`, the results of `y == binary_op(simd<T, 1>(identity_element), simd<T, 1>(y))[0]` and `y == binary_op(simd<T, 1>(y), simd<T, 1>(identity_element))[0]` are `true`.

8 *Returns:* If `none_of(mask)` is `true`, returns `identity_element`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, simd<T, 1>(x[`k_0`]), ..., simd<T, 1>(x[`k_n`]))[0]` where $k_0, \ldots, k_n$ are the selected indices of `mask`.

9 *Throws:* Any exception thrown from `binary_op`.

10 *Remarks:* The default argument for `identity_element` is equal to

- `T()` if `BinaryOperation` is `plus<>`,
- `T(1)` if `BinaryOperation` is `multiplies<>`,
- `T(~T())` if `BinaryOperation` is `bit_and<>`,
- `T()` if `BinaryOperation` is `bit_or<>`, or
- `T()` if `BinaryOperation` is `bit_xor<>`.

Note that the latest revision of P1928, already contains this new signature / wording, as this was preferred by LWG. LEWG still needs to re-confirm that change, otherwise I will have to roll it back.

5.1 Suggested poll

Poll: Reorder `binary_op` and `identity_element` as suggested by LWG and implemented in P1928R12.

SF	F	N	A	SA

6 NON-ISSUE 4: UNDO REMOVAL OF MEMBERS OF DISABLED `basic_simd`

Revision 0 of this paper raised the issue that constexpr-if branches require name lookup and that fails with disabled `basic_simd` and `basic_simd_mask` specializations. I Wrocław LEWG review, a request for a more real-world use case was made.

The issue broke legitimate `simd` unit test code. However, I cannot come up with any other case where this matters. Therefore, **I am not pursuing this any further.**

7 NON-ISSUE 5: HIDDEN FRIEND COMPOUND ASSIGNMENT OPERATORS

Consider:

```
std::simd<int, 4> s1, s2;
auto r = std::ref(s1); // r is a std::reference_wrapper

r += s2; // modifies s1: apply += to element wise to s1,sw3
r  = s2; // rebinds r to point to s2
r += s2; // modifies s2
```

This is due to `r` being convertible to `basic_simd&` and thus binding to:

```
template<class T, class Abi> class basic_simd {
  // …
  friend constexpr basic_simd& operator+=(basic_simd&, const basic_simd&) noexcept;
};
```

However, if compound assignment were specified as a member then name lookup would not find the operator (no member function lookup via ADL) and the example above becomes ill-formed:

```
template<class T, class Abi> class basic_simd {
  // …
  constexpr basic_simd& operator+=(const basic_simd&) noexcept;
};
```

LWG initially asked me whether this was a conscious design choice by LEWG and to consider using members for compound assignments. On the other hand, the following is already well-formed for scalars with the exact same behavior as for `simd` with hidden friend compound assignment:

```
int s1, s2;
auto r = std::ref(s1); // r is a std::reference_wrapper

r += s2; // modifies s1: apply += to element wise to s1,sw3
r  = s2; // rebinds r to point to s2
r += s2; // modifies s2
```

Consequently, changing compound assignment for `simd` to member operators creates an inconsistency between `simd<T>` and `T`.

Also, we need to consider that not every proxy reference type implements `operator=` as rebind. Other types with a conversion operator to lvalue-reference might implement it as assign-through. (e.g., proxy reference types similar to what we had for `simd::operator[]` or `vector<bool>::reference`)

After discussing the above in LWG, *LWG does not feel a need for changing this. But LWG would still like LEWG to sign off on the status quo.* ("BTW, did you know …?")

No poll requested.

8 WORDING

8.1 FEATURE TEST MACRO

No feature test macro is added or bumped.

8.2 MODIFY [SIMD.OVERVIEW] AND [SIMD.CTOR]

In [simd.overview], add:

[simd.overview]

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

At the end of [simd.overview], change:

[simd.overview]

If `basic_simd<T, Abi>` is enabled, `basic_simd<T, Abi>` is trivially copyable.

2 *Recommended practice*: Implementations should support ~~explicit~~implicit conversions between specializations of `basic_simd` and appropriate implementation-defined types. [*Note:* Appropriate types are non-standard vector types which are available in the implementation. — *end note*]

(8.2.0.1) 29.10.6.2 `basic_simd` constructors [simd.ctor]

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

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

2 *Constraints*: `value_type` satisfies `constructible_from<U>`. ~~`From` satisfies `convertible_to<value_type>`, and either~~

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

```
template<class U, class UAbi>
  constexpr explicit(see below) basic_simd(const basic_simd<U, UAbi>& x) noexcept;
```

5 *Constraints*: `simd-size-v<U, UAbi> == size()` is `true`.

6 *Effects*: Initializes the i^{th} element with `static_cast<T>(x[i])` for all i in the range of $[0, \texttt{size()})$.

7 *Remarks:* The expression inside `explicit` evaluates to `true` if either

- the conversion from `U` to `value_type` is not value-preserving, or
- both `U` and `value_type` are integral types and the integer conversion rank ([conv.rank]) of `U` is greater than the integer conversion rank of `value_type`, or
- both `U` and `value_type` are floating-point types and the floating-point conversion rank ([conv.rank]) of `U` is greater than the floating-point conversion rank of `value_type`.

`template<class G> constexpr explicit basic_simd(G&& gen)` ~~`noexcept`~~`;`

8 Let $\texttt{From}_i$ denote the type `decltype(gen(integral_constant<simd-size-type, i>()))`.

9 *Constraints*: $\texttt{From}_i$ satisfies `convertible_to<value_type>` for all i in the range of $[0, \texttt{size()})$. In addition, for all i in the range of $[0, \texttt{size()})$, if $\texttt{From}_i$ is an arithmetic type, conversion from $\texttt{From}_i$ to `value_type` is value-preserving.

10 *Effects*: Initializes the i^{th} element with `static_cast<value_type>(gen(integral_constant<simd-size-type, i>()))` for all i in the range of $[0, \texttt{size()})$.

11 *Remarks:* ~~The calls to gen are unsequenced with respect to each other. Vectorization-unsafe ([algorithms.parallel.defns]) standard library functions may not be invoked by gen.~~ `gen` is invoked exactly once for each i, in increasing order of i.

8.3 Modify [simd.mask.overview] and [simd.mask.ctor]

At the end of [simd.mask.overview], change:

[simd.mask.overview]

If `basic_simd_mask<Bytes, Abi>` is enabled, `basic_simd_mask<Bytes, Abi>` is trivially copyable.

2 *Recommended practice*: Implementations should support ~~explicit~~implicit conversions between specializations of `basic_simd_mask` and appropriate implementation-defined types. [*Note:* Appropriate types are non-standard vector types which are available in the implementation. — *end note*]

(8.3.0.1) 29.10.8.2 `basic_simd_mask` constructors [simd.mask.ctor]

```
constexpr explicit basic_simd_mask(value_type x) noexcept;
```

1 *Effects*: Initializes each element with `x`.

```
template<size_t UBytes, class UAbi>
  constexpr explicit basic_simd_mask(const basic_simd_mask<UBytes, UAbi>& x) noexcept;
```

2 *Constraints*: `basic_simd_mask<UBytes, UAbi>::size() == size()` is `true`.

3 *Effects*: Initializes the i^{th} element with `x[`*i*`]` for all i in the range of $[0, \texttt{size()})$.

```
template<class G> constexpr explicit basic_simd_mask(G&& gen) ~~noexcept~~;
```

4 *Constraints*: The expression `gen(integral_constant<`*`simd-size-type`*`, i>())` is well-formed and its type is `bool` for all i in the range of $[0, \texttt{size()})$.

5 *Effects*: Initializes the i^{th} element with `gen(integral_constant<`*`simd-size-type`*`, i>())` for all i in the range of $[0, \texttt{size()})$.

6 *Remarks:* ~~The calls to `gen` are unsequenced with respect to each other. Vectorization-unsafe ([algorithms.parallel.defns]) standard library functions may not be invoked by `gen`.~~ `gen` is invoked exactly once for each i, in increasing order of i.
