| | |
|---|---|
| Document Number: | P3430R0 |
| Date: | 2024-10-15 |
| Reply-to: | Matthias Kretz <m.kretz@gsi.de> |
| Audience: | LEWG |
| 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

(placeholder)

# 2 STRAW POLLS

(placeholder)

# 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 | with P3430R0 |
|---|---|
| `using floatv = std::simd<float>;`<br><br>`void f(floatv x)`<br>`{`<br>`  x + 2; // ill-formed`<br>`  x + float(2); // OK`<br>`  x + floatv(2); // ill-formed`<br><br>`  x = 2 // ill-formed`<br>`  x = float(2) // OK`<br>`  x = floatv(2) // ill-formed`<br>`}` | `using floatv = std::simd<float>;`<br><br>`void f(floatv x)`<br>`{`<br>`  x + 2; // ill-formed`<br>`  x + float(2); // OK`<br>`  x + floatv(2); // OK`<br><br>`  x = 2 // ill-formed`<br>`  x = float(2) // OK`<br>`  x = floatv(2) // OK`<br>`}` |

~~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.

---

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 P3430R0 |
|---|---|
| `void f(std::simd<int, 4> x)`<br>`{`<br>`  x = static_cast<std::simd<int, 4>>(`<br>`  _mm_add_epi32(static_cast<__m128i>(x),`<br>`  static_cast<__m128i>(x)));`<br>`}` | `void f(std::simd<int, 4> x)`<br>`{`<br>`  x = _mm_add_epi32(x, x);`<br><br>`}` |

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

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

### 3.3 suggested polls

Poll: Make the broadcast constructor conditionally `explicit` (P3430R0 Section 3.1)

| SF | F | N | A | SA |
|---|---|---|---|---|
| | | | | |

Poll: Make conversions to/from implementation-defined vector types implicit (strike `explicit`) (P3430R0 Section 3.2)

| SF | F | N | A | SA |
|---|---|---|---|---|
| | | | | |

## 4 Issue 2: drop "unsequenced" from generator ctor

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 $i^{\text{th}}$ 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 $i^{\text{th}}$ 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 $i$.

---

## 4.1 SUGGESTED POLL

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 |
|---|---|---|---|---|
| | | | | |

# 5 Issue 3: Reorder identity_element and binary_op on reduce

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`.

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`.

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`.

- 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 $k_0, \dots, k_n$ 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 ISSUE 4: UNDO REMOVAL OF MEMBERS OF DISABLED basic_simd

LEWG directed me to remove the `size` member from `basic_simd` and `basic_simd_mask` for disabled specializations (it was called "not supported" back then):

Poll: Make `simd_size` exposition only and cause `simd` to have the size static data member if and only if `T` is a vectorizable type and `Abi` is an ABI tag.

| SF | F | N | A | SA |
|---|---|---|---|---|
| 1 | 7 | 4 | 1 | 0 |

But that change breaks valid uses cases on constexpr if:

```
if constexpr (std::default_initializable<V>) {
  constexpr int width = V::size();
}
```

Because `size` is not a member of disabled `basic_simd` specializations anymore, the code above is still ill-formed even though the user tried to guard against it with the constexpr if branch. What we need to consider is that name lookup still isn't allowed to fail inside constexpr if branches. It is okay if no viable overload exists, as long as there is at least one.[3] The same argument can be made for all the other members of `basic_simd` and `basic_simd_mask`.

Possible resolutions:

- Bite the bullet and go all the way: Make disabled `basic_simd` and `basic_simd_mask` incomplete types.
- Walk back the change requested by LEWG in Varna and keep all the members in disabled `basic_simd` and `basic_simd_mask` specializations. (TS behavior and my preference / suggestion)

Suggested wording change:

---

[simd.overview]

2 Every specialization of **`basic_simd`** is a complete type. The types **`basic_simd<T, deduce-t<T, N>>`** for all vectorizable `T` and with `N` in the range of $[1, 64]$ are enabled. It is implementation-defined whether any other **`basic_simd<T, Abi>`** specialization with vectorizable `T` is enabled. Any other specialization of **`basic_simd`** is disabled.

If **`basic_simd<T, Abi>`** is disabled, the specialization has a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. ~~In addition only the `value_type`, `abi_type`, and `mask_type` members are present.~~

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

---

## 6.1 Suggested poll

Poll: Name lookup of members of disabled `basic_simd` specializations works if it works for enabled `basic_simd` specializations. Likewise for `basic_simd_mask`.

| SF | F | N | A | SA |
|---|---|---|---|---|
| | | | | |

3 Well... depends: `https://compiler-explorer.com/z/xP3PPjezx`