Document Number:	P3319R1
Date:	2024-06-28
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	SG9, LEWG
Target:	C++26

Add an iota object for simd (and more)

ABSTRACT

There is one important constant in SIMD programming: `0, 1, 2, 3, ...`. In the standard library we have an algorithm called `iota` that can initialize a range with such values. For `simd` we want to have simple to spell constants that scale with the SIMD width. This paper proposes a simple facility that can be generalized.

CONTENTS

1 CHANGELOG

(placeholder)

1.1 CHANGES FROM REVISION 0

Previous revision: P3319R0

- Add a simple example to the motivation section.
- Expand the "Generalization" section to clearly define the feature rather than just sketching it. Also add a discussion of initial value and step.
- Discuss why reusing the existing `iota` algorithm/view does not work/suffice for the `simd` use case.
- Discuss why `iota_v` is the right name.

2 STRAW POLLS

(placeholder)

3 MOTIVATION

The 90%[1] use case for simd generator constructors is a simd with values 0, 1, 2, 3, ... potentially with scaling and offset applied. However, often it would be easier and more readable to use an "iota" `simd` object instead.

generator ctor

```
std::simd<int> a([](int i) { return i; };

std::simd<int> b([](int i) { return 2 + 3 * i; };
```

iota

```
auto a = std::iota_v<std::simd<int>>;

auto b = 2 + 3 * std::iota_v<std::simd<int>>;
```

1 Sorry, that number is completely made up.

An example where an `iota_v<simd>` comes up is the calculation of the Mandelbrot set. The program needs to iterate over all visible pixels and calculate the corresponding value in the complex plane. Thus a loop like

```
for (int x = 0; x < 1024; ++x) {
  float real = float(x) * scale + offset;
```

turns into

```
using floatv = simd<float>;
using intv = rebind_simd_t<int, floatv>;
for (intv x = iota_v<intv>; any_of(x < 1024); x += intv::size()) {
  floatv real = floatv(x) * scale + offset;
```

The minimal definition I propose for `basic_simd` can look like this:

```
namespace std {
  template <class T>
    requires std::is_arithmetic_v<T> or detail::simd_type<T>
    inline constexpr T
    simd_iota_v = T();

  template <detail::simd_type T>
    inline constexpr T
    simd_iota_v<T> = T([](auto i) { return static_cast<typename T::value_type>(i); });
}
```

If [P3287R0] is adopted to introduce a `std::simd` namespace, it would be called `std::simd::iota_v` (and `std::simd_generic::iota_v`) instead.

4 GENERALIZATION

By defining a variable template `std::simd_iota_v<T>` where `T` must be a `basic_simd` type, we're simply initializing a sequence of values at compile time. We can create such an object for more types. This is especially interesting for the degenerate case in SIMD-generic programming, where `T` could e. g. be an `int`. An `std::iota_v<int>` is nothing other than an object `int` with value `0`.

We can easily generalize to `iota_v<std::array<T, N>>` and `iota_v<T[N]>`. And the next step then is to allow any type that

- has a static extent,
- has a `value_type` member type,
- can be list-initialized with `N` numbers of type `value_type`, where `N` equals the static extent of the type, and
- where `value_type() + 1` is an constant expression and convertible to `value_type`.

But there are more types (in the standard library and beyond) where we can create such an object. All we need is a type

1. with valid `ranges::range_value_t<T>` type (this could be weakened to also allow `std::tuple<int, int>`),
2. with static extent (`T::size()`, `T::extent`, `std::extent_v<T>`, or `std::tuple_size_v<T>`),
3. and that can be list-initialized from a sequence of `N` integers (cast to `range_value_t<T>`), where `N` equals the static extent of the type.

For the scalar case, a very general constraint requires `T` to be

- a regular type
- that can be list-initialized from a single value
- and that compares equal to that value after construction.

Consequently you could write

```
auto x = std::iota_v<float[5]>;
auto y = std::iota_v<std::array<my_fixed_point, 8>>;
// ...
```

A second generalization could allow different sequences other than only `0, 1, 2, 3, 4, ...`. `std::iota` and `std::ranges::iota` take a `value` argument to define the first value in the sequence. They do not allow any different step other than applying the pre-increment operator.

For `simd`, I would typically just write e. g.

```
constexpr auto vec = std::iota_v<std::simd<int>> * 3 + 5; // 5, 8, 11, ...
```

To construct the same sequence for an array, `iota_v` would require a "first" and a "step" argument:

```
constexpr auto arr = std::iota_v<std::array<int, 4>, 5, 3>; // 5, 8, 11, 14
```

Providing a (defaulted) "step" argument is simple and more general. The only reason, that I can think of, for not adding it is that `std::iota` / `std::ranges::iota` don't have it.

5 Alternative: reuse existing iota

We already have `std::iota` and `std::ranges::iota`. Why isn't that sufficient to create a solution that composes?

One motivation for `iota_v<simd<int>>` instead of `simd<int>::iota` is that `iota_v<int>` works while `int::iota` cannot work. The same is true for `simd<int>(views::iota(0))` vs. `int(views::iota(0))`. Supporting the degenerate case is very helpful for SIMD-generic programming.

```
// scalar loop:
for (int i = 0; i < 1024; ++i) {
  ...
}

// simd loop:
for (auto i = iota_v<simd<int>>; all_of(i < 1024); i += simd<int>::size) {
  ...
}

// simd-generic loop:
for (auto i = iota_v<T>; all_of(i < 1024); i += simd_size_v<T>) {
  ...
}

// alternative:
for (int ii = 0; ii < 1024; ii += simd_size_v<T>) {
  T i = ii + iota_v<T>;
  ...
}
```

In addition, what `simd(range)` does depends on the outcome of P3299 "range ctors". In any case we are requiring contiguous_range. So `simd(random_access_range)` needs another paper altogether (while convenient, this is rarely what the user wanted; making non-contiguous loads ill-formed helps against "performance errors"). So we could overload for specific non-contiguous ranges, where we know that we can restore good performance. But that's going to be a closed set, rather than a general concept. Why then would `simd(std::views:iota(0))` work but `simd(boost::views::iota(0))` is ill-formed?

Our desired outcome of P3299 is that `simd(range)` requires a statically sized contiguous range with exactly matching size. Thus we would need to call `std::simd::load<simd<int>>(std::views::iota(0))` instead. This is now even more verbose than the current solution `simd<int>([](int i) return i; )`. It completely fails at the goal to make the code more readable.

Then what about `std::views::iota(0) | std::ranges::to<basic_simd>()`? It's still too long for a rather basic constant. And why should this work if both

- `std::views::iota(0) | std::ranges::to<std::array>();` and
- `std::views::iota(0) | std::ranges::to<std::array<int, 4>();`

don't work?

6 NAMING: IS REUSE OF THE TERM "IOTA" CONFUSING OR HELPFUL?

In the Vc library, the library behind the initial proposal back in 2013, there's a `Vc::Vector<T>::IndexesFromZero()` constant. Back then SG1/WG21 wanted to reduce the scope for the TS to a minimum and the constant was never considered any further. In any case, `IndexesFromZero` is a fairly descriptive/elaborate name. But in the standard library we already have a term for a sequence like this. And it's "iota". Using a different term for something that isn't different (concept) is confusing and incoherent.

`std::iota` has an existing meaning, as an algorithm that initializes a given existing range. What this paper proposes is still sufficiently different that we don't want to overload that exact name. Instead, since we're defining an "iota value", we propose the name `iota_v`.

If we don't move `simd` into a `std::simd` subnamespace and if we don't want to generalize the "iota value" beyond `simd`, then we should be considering `std::simd_iota_v` over `std::iota_v`.

7 RELATION TO LIST-INITIALIZATION OF `simd`

If we add a constructor to `basic_simd` that enables list-initialization, then many users might use that in place of a generator constructor. This leads to code that doesn't scale with the vector width anymore. Therefore we should provide a simple facility that is concise and portable[2].

8 PROPOSED POLLS

Poll: We want an iota facility for `basic_simd`

SF	F	N	A	SA

Poll: The iota facility should be generalized to scalars (for SIMD-generic programming)

SF	F	N	A	SA

Poll: The iota facility should be generalized to any sequence of static extent

SF	F	N	A	SA

Poll: The iota facility should be generalized to allow a different first value

SF	F	N	A	SA

2 in terms of SIMD width

Poll: The iota facility should be generalized to allow a different step value

SF	F	N	A	SA

9 WORDING

TBD after deciding on the preferred solution.

A BIBLIOGRAPHY

[P3287R0] Matthias Kretz. *P3287R0: Exploration of namespaces for std::simd*. ISO/IEC C++ Standards Committee Paper. 2024. URL: https://wg21.link/p3287r0.