Document Number: P3319R5
Date: 2025-02-14
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LWG
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

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.

1.2 CHANGES FROM REVISION 1

Previous revision: P3319R1

- Add SG9 poll results.
- Add wording for `std::simd_iota`.

1.3 CHANGES FROM REVISION 2

Previous revision: P3319R2

- Clean up naming in the discussion.
- Discuss overflow in a new section (Section 8).
- Mandate "no overflow" in the wording.

1.4 CHANGES FROM REVISION 3

Previous revision: P3319R3

- Adjust names after simd subnamespace was accepted.
- Add feature test macro bump to the wording.

1.5 CHANGES FROM REVISION 4

Previous revision: P3319R4

- Add constraint on vectorizable as directed by LEWG.

2 STRAW POLLS

2.1 SG9 at Wrocław 2024

Poll: We want the variable template that creates an iota sequence described in the paper for `basic_simd` and arithmetic scalars.

SF	F	N	A	SA
6	1	1	0	0

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

SF	F	N	A	SA
0	0	4	3	1

Poll: Assuming the author provides wording and a wording expert verifies that it matches design intent, forward P3319R1 to LEWG for inclusion in C++26.

SF	F	N	A	SA
6	1	1	0	0

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.

<table>
<tr><th>generator ctor</th><th>iota</th></tr>
<tr><td><pre>namespace dp = std::datapar;

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

dp::simd<int> b([](int i) { return 2 + 3 * i; });</pre></td><td><pre>namespace dp = std::datapar;

auto a = dp::iota<dp::simd<int>>;

auto b = 2 + 3 * dp::iota<dp::simd<int>>;</pre></td></tr>
</table>

1 Sorry, that number is completely made up.

An example where an `datapar::iota<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 = dp::simd<float>;
using intv = dp::rebind_t<int, floatv>;
for (intv x = dp::iota<intv>; any_of(x < 1024); x += intv::size()) {
  floatv real = floatv(x) * scale + offset;
```

The minimal definition proposed can be implemented like this:

```
namespace std::datapar {
  template <class T>
    requires is_arithmetic_v<T>
      || (simd-type<T> && is_arithmetic_v<typename T::value_type>)
    constexpr T iota = T();

  template <class T, class Abi>
    constexpr basic_simd<T, Abi>
    iota<basic_simd<T, Abi>>([](T i) {
      static_assert(basic_simd<T, Abi>::size() - 1 <= numeric_limits<T>::max());
      return i;
    });
}
```

4 GENERALIZATION

By defining a variable template `std::datapar::iota<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`. A `std::datapar::iota<int>` is nothing other than an object `int` with value 0.

We can easily generalize to `std::iota_v<std::array<T, N>>` and `std::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<simd<int>>` instead of `simd<int>::iota` is that `iota<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 = dp::iota<dp::simd<int>>; all_of(i < 1024); i += dp::simd<int>::size) {
  ...
}

// simd-generic loop:
for (auto i = dp::iota<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 + dp::iota<T>;
  ...
}
```

In addition, with [P3299R3] *Proposal to extend std::simd with range constructors* we continue to only enable construction and load from contiguous ranges. 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?

The outcome of [P3299R3] *Proposal to extend std::simd with range constructors* is that `simd(range)` requires a statically sized contiguous range with exactly matching size. Thus, even the call `std::simd_-unchecked_load<simd<int>>(std::views::iota(0))` does not work. It's also not a solution to the problem posed, since it is now even more verbose than the generator constructor 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 sufficiently different that we don't want to overload that exact name. In addition, with `std::iota` being a function and this proposal adding a variable template it is technically impossible to overload the same name.

If we decide not to generalize the facility then `std::datapar::iota` is the preferred name. If we do want to generalize, we propose the name `std::iota_v`, since we're defining an "iota value". If LEWG considers the `_v` suffix to be reserved for traits then we should consider `std::iota_value` instead.

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 BEHAVIOR ON OVERFLOW

Consider `iota<simd<char, 512>>` where `is_signed_v<char>` is `true`. While the standard only requires support of `basic_simd` width up to 64, implemenations are still free to enable larger widths. Should this be ill-formed (Mandates vs. Constraint) or should it match `std::iota` and `std::ranges::iota` behavior and produce a sawtooth wave?

I was using `std::datapar::iota` in test code and encountered both cases. In one case I had an error in my test code and making it ill-formed helped fixing the problem. In another case I was comparing against memory intitialized by `std::iota` and making `std::datapar::iota` ill-formed unnecessarily made my test cases harder to write.

Granted, most people won't use `std::datapar::iota` in order to compare it against `std::iota`. Instead, the most likely use will be as a sequence of increasing offsets. In that case wraparound introduces a bug, and potentially even out-of-bounds indexes leading to memory-safety issues. Therefore, I prefer making `std::datapar::iota` ill-formed if the `basic_simd` width is larger than

2 in terms of SIMD width

the largest representable number. In terms of helpful diagnostics, a "Mandates" clause is the better solution. The wording below implements it that way.

9 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

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

SF	F	N	A	SA

10 WORDING

10.1 FEATURE TEST MACRO

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

10.2 CHANGES TO [SIMD]

Add the following to ([simd.syn]), after the declaration of `cat`:

[simd.syn]

```
template<size_t Bs, class... Abis>
  constexpr basic_simd_mask<Bs, deduce-t<integer-from<Bs>,
                             (basic_simd_mask<Bs, Abis>::size() + ...)>>
    cat(const basic_simd_mask<Bs, Abis>&...) noexcept;
```

```
template<class T> inline constexpr T iota = see below;

// [simd.mask.reductions], basic_simd_mask reductions
```

Add the following at the end of ([simd.creation]):

[simd.creation]

5 *Returns:* A data-parallel object initialized with the concatenated values in the `xs` pack of data-parallel objects: The i^{th} `basic_simd`/`basic_simd_mask` element of the j^{th} parameter in the `xs` pack is copied to the return value's element with index i + the sum of the width of the first j parameters in the `xs` pack.

```
template<class T> inline constexpr T iota = see below;
```

6 *Constraints*: Either `T` is vectorizable and `is_arithmetic_v<T>` is `true`, or `T` is an enabled specialization of `basic_simd`.

7 *Mandates*: `is_arithmetic_v<T>` is `true` or `T::size() - 1` $\leq$ `numeric_limits<typename T::value_type>::max()`.

8 *Effects*: If `is_arithmetic_v<T>` is `true` the value of `iota<T>` is equal to `T()`. Otherwise, the value of `iota<T>` is equal to `T([](typename T::value_type i) { return i; })`.

(10.2.0.1) 29.10.7.7 Algorithms [simd.alg]

A BIBLIOGRAPHY

[P3299R3] Daniel Towner, Matthias Kretz, and Ruslan Arutyunyan. *Proposal to extend std::simd with range constructors*. ISO/IEC C++ Standards Committee Paper. 2024. URL: https://wg21.link/p3299r3.