Document Number: P3275R0
Date: 2024-05-22
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG
Target: C++26

Replace simd operator[] with getter and setter functions — or not

Abstract

There was discussion in LEWG in Kona '23 whether `operator[]` is the right interface for reading and writing individual elements of a `basic_simd` or `basic_simd_mask`. This paper discusses the underlying issue and explores alternatives.

Contents

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

3 MOTIVATION

3.1 PROXY REFERENCES IN C++ MAKE ME :'(

[simd.general]

A data-parallel type consists of one or more elements of an underlying vectorizable type, called the element type. […] The elements in a data-parallel type are indexed from 0 to width − 1.

Since that's a given, we sure want to be able to access individual elements. `basic_simd` and `basic_simd_mask` implement `operator[]` for element access:

```
std::simd<int> x = 0;
x[0] = 1;       // OK
int y = x[0];   // OK
x[0] = y;       // OK
auto z = x[0]; // OK, but:   #1
z = 2;          // ill-formed #2
```

The `basic_simd` and `basic_simd_mask` types hold values of their `value_type`, but they typically don't hold *objects* of `value_type`. Consequently, both `operator[]` overloads cannot return an lvalue-reference. The `const` overload therefore returns a prvalue and the non-`const` overload returns a proxy reference. The proxy reference implements assignment and compound-assignment operators for assigning through to the selected element of the `basic_simd`/`basic_simd_mask`. Thus, line `#1` above deduces the type of `z` to be that proxy reference type. The declaration of `z` does not look like a reference at all. Therefore, assignment in line `#2` is ill-formed, in order to avoid the surprising behavior of modifying `x`.

In any case, the fact that a proxy reference is used instead of an lvalue-reference, makes subscripting into a `simd` error-prone. Whenever the subscript expression is used in a function argument with deduced type, bad things are likely to happen.[1] If we had a language feature to decay the proxy reference type to `value_type` on deduction, then a lot of problems could be avoided. But we don't have that feature and there's no reasonable chance to get it for C++26.

1 Just recall `vector<bool>::reference`.

3.2 why, though? lvalue ref is fine, no?

With GCC today you can write

```
using simd [[gnu::vector_size(16)]] = int;

simd x = {};
int& ref = x[0];
x += 1;
ref = 2;
```

If that's possible, then `std::simd<int>`'s subscript operator can simply return an lvalue reference, no? While that's true for this example, it's not true in general. For one, Clang is fairly strict about not handing out lvalue references. I.e. the above code does not compile. But more importantly, for some targets or implementations an intrinsic type might need to be used, which doesn't allow forming lvalue references to its elements either. Also `basic_simd` does not prohibit an implementation to use a different element type internally for its SIMD registers. E.g. an efficient implementation of 8-bit and 16-bit integers on the (outdated) Intel Knights architecture required the use of 32-bit integer SIMD registers and instructions. It is also conceivable that implementations will implement `simd<std::float16_t>` using 32-bit `float` SIMD registers for targets without hardware support.

The situation for `basic_simd_mask` is much clearer. There are three typical storage formats for masks in current hardware:

1. Full SIMD registers where either all bits are 0 or all bits are 1 for the complete number of value bits.
2. Bitmasks use one bit per mask element. This is analogue to `vector<bool>` and `bitset` not being able to return lvalue references to `bool`.
3. Mask registers that use one bit per value type byte. This is similar to the above, where we would need to return a reference to a single bit (just at a different position).

Therefore, even if Clang would implement GCC's behavior with regard to forming lvalue references to vector elements, that doesn't help for `basic_simd_mask`.

3.3 simd<UDT>

If we want to extend `basic_simd`'s vectorizable types to user-defined types, we need to consider a consistency issue: `simd<T>` applies every operator and operation element-wise (unless the name clearly hints at a horizontal operation).

While I don't think e.g. `simd<array<short, 4>, 2>` is a useful thing, it's also not completely crazy. However, its only interesting semantics in a `basic_simd` is data-parallel subscripting (apply `operator[]` element-wise):

```
using xyzw = std::array<short, 4>;
std::simd<xyzw, 2> a = {};
std::simd<short, 2> w = a[3]; // yes, simd not array: element-wise subscript!
a[3] = w + std::integral_constant<short, 1>();
```

(This is neither going to be great for performance, nor is it clear whether we should implement such a "data-parallel subscript", which requires a proxy reference again.)

This example is not meant to motivate an element-wise `operator[]` for `simd`. It's meant to show that the current `simd::operator[]` is inconsistent with the "apply operators element-wise" rule. Applying `operator[]` element-wise on a `simd<int>` is obviously ill-formed since `int` doesn't have a subscript operator. Consequently, maybe the current P1928 `basic_simd` and `basic_simd_mask` shouldn't overload `operator[]`?

4 REPLACEMENT EXPLORATION

If we only want to get rid of the proxy reference but are not concerned about the consistency argument in Section 3.3, then we could consider a read-only subscript operator. We still have two choices:

read-only forever	keep design space open

`class simd {`
`  value_type operator[](simd-size-type i) const;`
 	
`};`
	`class simd {`
	`  value_type operator[](simd-size-type i) const;`
	`  void operator[](simd-size-type) = delete;`
	`};`

`std::simd<int> v;` `int x = v[0];`	`std::simd<int> v;` `int x = std::as_const(v)[0];`

4.1 MAKING THE CASE FOR: A READ-ONLY SUBSCRIPT IS SUFFICIENT

A common use case for the subscript operator arises through the generator constructors of `basic_simd` and `basic_simd_mask`. With P1928 you would write a permutation like this:

```
simd<int> v;
simd<int> reversed([&](int i) { return v[v.size - i - 1]; });
```

The generator constructor often reads from another `basic_simd` and since it needs to compute a scalar, it typically only reads one element at a time. And it never updates a value via subscripting; the update happens by constructing a whole new `basic_simd` (with a good chance of the compiler producing vector instructions). Making such code any harder to write is not necessarily helping users. The above example is intuitively understandable (well, the subscripting part, the generator constructor maybe less so).

Therefore, it seems like the simplest and still fairly usable "fix" is to remove the non-const subscript overload.

There is some curious existing practice in GCC supporting this approach:

```
using simd [[gnu::vector_size(16)]] = int;

constexpr simd f(simd x) {
  x[0] = 1;
  return x;
}

constexpr simd test0 = f(simd{}); // ill-formed: x[0] = 1 is not a constant expression

constexpr simd g(simd x) {
  x = simd{1, x[1], x[2], x[3]};
  return x;
}

constexpr simd test1 = f(simd{}); // OK
```

I.e. assignment through vector subscripts cannot be used in constant expressions. Instead a complete new vector must be constructed. If the non-const subscript operator is removed from `basic_simd` and `basic_simd_mask`, then GCC's restriction for constant expressions becomes `std::basic_simd`'s behavior.

4.2 Is simd as a read-only range a sufficient replacement?

If `basic_simd` and `basic_simd_mask` have a `begin()` and `end()` iterator, making them read-only random-access ranges, then accessing an element is equivalent to accessing a scalar from an `initilizer_list`:

```
std::simd<int, 4> v;
auto v0 = v.begin()[0];
auto v3 = v.begin()[3];
```

In a very similar approach, making `basic_simd` convertible to `array` allows subscripting through the array:

```
std::simd<int, 4> v;
std::array a = v;
a[1] += 1;
v = a;
```

4.3 Allowing for writable subscript after C++26

If we want to keep the design space open while still overloading `basic_simd ::operator[]`, then subscripting would become even more awkward to use. Consequently, `basic_simd` and `basic_-`

`simd_mask` should rather have no subscript operator at all for C++26. In the following exploratory examples, I will use the function names `get` and `set` as placeholder names. I also added a line to every example, considering the same syntax for the degenerate case of an `int` instead of a `simd<int>`.

1. P1928 status quo:

```
std::simd<int> v;
v[0] += 1;

int x;
x[0] += 1; // nope
```

2. `set(index, value)` member function:

```
std::simd<int> v;
v.set(0, 1 + v.get(0));

int x;
x.set(0, 1 + x.get(0)); // nope
```

3. `set(object, index, value)` non-member function:

```
std::simd<int> v;
set(v, 0, 1 + get(v, 0));

int x;
set(x, 0, 1 + get(x, 0)); // not impossible
```

4. explicit proxy reference without assignment and conversion operators:

```
std::simd<int> v;
element_reference(v, 0).set(1 + element_reference(v, 0).get());

int x;
element_reference(x, 0).set(1 + element_reference(x, 0).get()); // not impossible
```

5. explicit proxy reference with operators:

```
std::simd<int> v;
element_reference(v, 0) += 1;

int x;
element_reference(x, 0) += 1; // not impossible
```

6. make degenerate size 1 `basic_simd<T>` convertible to/from `T` (and `basic_simd_mask` to/from `bool`)[2]

```
std::simd<int> v;
int v0 = permute<1>(v, [](int) { return 0; });
v0 += 1;
v = permute<v.size>(simd_cat(v, v0), [](int i) { return i == 0 ? v.size : i; });

// not impossible:
int x;
int x0 = permute<1>(x, [](int) { return 0; });
x0 += 1;
x = permute<1>(simd_cat(x, x0), [](int i) { return i == 0 ? 1 : i; });
```

4.4 Discussion

In the example above I chose the problem of updating the value of a single element of a `basic_simd`, to showcase how much compound assignment can aid in readability. In my opinion, the missing compound read-modify-write syntax in examples 2, 3, and 4 is a huge downside.

Further observations on the above examples:

- Making `simd<T, 1>` convertible to `T` seems interesting, but not like a solution to this problem.
- `set(x, y, z)` is not intuitive whereas `x[y] = z` cleary states the intended operation.
- `x.set(y, z)` is better than `set(x, y, z)` in terms of "what is set where?", but ideally a "set" function would only take a single argument: the new value.
- This is achieved by example 4, which creates an object that identifies a single element, thus allowing `set` to only take the new value as function argument.
- We can pass lvalue-references around, (e.g. `int& x = data[0];`). Examples 2 and 3 don't allow an equivalent for `basic_simd` elements. 4 and 5 however would act as a drop-in for lvalue references and thus would allow modifying a single `basic_simd` element "from a distance".[3]

The ability to write simd-generic element access is not super important, but certainly aids against code duplication in some situations.

4.5 Recommendation

I still believe the use of the subscript operator for `basic_simd` and `basic_simd_mask` is fairly intuitive and natural. From experience I would guess that read-only subscript is 90% if not 99% of the typical

2 `permute<1>` returns a `simd<int, 1>`, which could be implicitly convertible to `int`.
3 Typically not a good idea, though.

current use of subscripting. I may be biased from writing many unit tests, and nobody actually uses assignment through subscripts (or if they do, a generator constructor would have been the better solution anyway). Therefore, I recommend to simply remove the non-const subscript operator from `basic_simd` and `basic_simd_mask`.

If that's not an acceptable outcome, my next recommendation would be the addition of an `element_reference` type that implements all (compound) assignment operators (but without restricting them to rvalue, like the current implicit proxy reference type does). Basically make example 5 work.

5 PROPOSED POLLS

All of these polls are phrased against the status-quo (P1928). Thus no concensus on all polls implies we keep the `basic_simd` and `basic_simd_mask` subscript operators with proxy-reference on non-const subscripts.

Poll: Remove non-const `operator[]` from `basic_simd` and `basic_simd_mask`. (⇒ Subscripting will stay read-only forever.)

SF	F	N	A	SA

Poll: Remove all subscript operators if we make `basic_simd` and `basic_simd_mask` random-access ranges (TBD). (⇒ status-quo until paper making `basic_simd` and `basic_simd_mask` a range lands.)

SF	F	N	A	SA

Poll: Replace subscript operators by member `get` and `set` functions (names TBD).

SF	F	N	A	SA

Poll: Replace subscript operators by non-member `get` and `set` functions (names TBD).

SF	F	N	A	SA

Poll: Replace subscript operators by `element_reference` and `set` functions (names TBD).

SF	F	N	A	SA

6 WORDING

TBD after deciding on the preferred solution.

A ACKNOWLEDGMENTS

Daniel Towner and Ruslan Arutyunyan contributed to this paper via discussions / reviews. Thanks also to Jeff Garland for reviewing.