Restore simd::vec broadcast from int

ABSTRACT

The broadcast constructor in the Parallelism 2 TS allowed construction from (`unsigned`) `int`, allowing e.g. `vec<float>() + 1`, which is ill-formed in the CD. This breaks existing code that gets ported from the TS to `std::simd`. The design intent behind `std::simd` was for this to work. However, the understanding in LEWG appeared to be that we can't get this right without constexpr function arguments getting added to the language. This paper shows that a `consteval` constructor overload together with `constexpr` exceptions can resolve the issue for C++26 and is a better solution than constexpr function arguments would be.

CONTENTS

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

3   MOTIVATION

It is very common in floating-point code to simply write e.g. `* 2` rather than `* 2.f` when multiplying a `float` with a constant:

```cpp
float f(float x) { return x * 2; } // converts 2 to float (at compile time)

float g(float x) { return x * 2.; } // converts x to double (at run time)

float h(float x) { return x * 2.f; } // no conversions
```

More importantly, using `* 2` works reliably in generic code, where the type of `x` could be any arithmetic type.

Since this is so common, `std::experimental::simd<T>` made an exception for `int` in the broadcast constructor to not require value-preserving conversions. Consequently, the TS allows:

```cpp
using floatv = std::experimental::native_simd<float>;

floatv f(floatv x) { return x * 2; } // converts 2 to float and broadcasts (at compile time)

floatv g(floatv x) { return x * 2.; } // ill-formed

floatv h(floatv x) { return x * 2.f; } // broadcasts 2.f to floatv
```

When porting existing code written against the TS to C++26, the first step is to adjust the types:

```cpp
using floatv = std::~~experimental::native_simd~~simd::vec<float>;
```

Except for uses of `std::experimental::where`, which need to be refactored to use `simd::select`, the remaining code should work. The one place where it doesn't work is code such as in function `f`, where 2 needs to be replaced:

```cpp
floatv f(floatv x) { return x * ~~2~~std::cw<2>; }
```

Since we don't have `constexpr` function arguments in the language, `std::simd` works around it by recognizing *integral-constant-like* / `constant-wrapper-like` types, that encode a *value* into a type. This, however, comes at a compile-time cost. Every different value leads to a template specialization of both `constant_wrapper` and a `basic_vec` broadcast constructor (with it's helper types/concepts to determine whether the specialization is allowed). Consequently, for `vec<float>`, I would recommend to always use an `f` suffix rather than `std::cw`.

But that solution is fairly limited, since we don't have literals for 8-bit and 16-bit integers in the language. A function template like

```cpp
template <simd_integral V>
V f(V x) {
  return x + 1; // ill-formed for V::value_type = (u)int8_t, (un)int16_t, and uint32_t
}
```

needs to use `x + V(1)`[1]. A clever user might write `x + '\1'` instead. But that fails for the `char` type with different signedness.

Consequently, users would need to get used to writing explicit conversions for the constants they use in `std::simd` code. That's not only verbose and ugly, it is also error-prone. Whenever we coerce our users into writing explicit conversions, then value-changing conversions cannot be diagnosed as erroneous anymore. An explicit `static_cast<uint64_t>(-1)` means `0xffff'ffff'ffff'ffff`, whereas `uint64_t x = -1` could have been intended to mean `0x0000'0000'ffff'ffff` or is a result of a logic flaw in the code. E.g., GCC's `-Wsign-conversion` diagnoses the latter, but not the former[2].

If, with C++26, our users are starting to explicitly convert their `int` constants to `basic_vec`, then the interface of `basic_vec` is at least in part guilty for introducing harder to find bugs.

Tony Table 1 presents an example of the solution[3]. Note that the code on the left will never warn about the value-changing conversion, even with all conversion related warnings enabled. This is due to the explicit conversion, which is telling the compiler "I know what I'm doing; no need to warn me about it".

before	with P3844R0
| ```
template <simd_floating_point V>
V f(V x) {
  return x + V(0x5EAF00D);
}

f(vec<double>()); // OK

// compiles but
// adds 99282960 instead of 99282957
f(vec<float>());

// compiles but
// adds infinity instead of 99282957
f(vec<std::float16_t>());
``` | ```
template <simd_floating_point V>
V f(V x) {
  return x + 0x5EAF00D;
}

f(vec<double>()); // OK

// ill-formed: uncaught exception on
// value-changing conversion
f(vec<float>());

// ill-formed: uncaught exception on
// value-changing conversion
f(vec<std::float16_t>());
``` |

~~Tony~~Before/After Table 1: Add an offset

A safer implementation of the code on the left side of Tony Table 1 (without this paper) would have been to write `x + std::cw<0x5EAF00D>` instead. Then the value-changing conversion would have resulted in a constraint failure on the broadcast constructor. However, `V(0x5EAF00D)` is shorter and needs fewer template instantiations. I expect most users (including myself) will/do not use `std::cw` all over the place.

---

1 explicit conversion to `basic_vec` allows conversions that are not value-preserving
2 And that's useful, because the former says "I'm intentionally doing this conversion, no need to warn."
3 I got bitten by this in my `std::simd` unit tests

4   DESIGN SPACE

In the design review of this issue of the broadcast constructor it was overlooked (and never discussed) that a `consteval` overload of the broadcast constructor could solve this problem. Before `constexpr` exceptions, we would have worded it to be ill-formed (by unspecified means) if the value changes on conversion to the `basic_vec`'s value-type. Now that we have `constexpr` exceptions, we can specify a `consteval` broadcast overload that throws on value-changing conversion. If the caller cares, the exception can even be handled at compile time. (I believe it should not throw in C++26, for a minimal change to the WD.)

Ordering the overloads for overload resolution is tricky, which is another reason why we should consider this issue before C++26 ships and potentially take action even if we don't add a `consteval` overload. Overload resolution does not take `consteval` into account. The process of finding candidate functions ([over.match.funcs.general]), however, does remove explicit constructors from the candidate set if the context does not allow the explicit constructor to be called.

4.1   STATUS QUO

The following code shows the properties of the current broadcast constructor.

See Appendix A for the definition of the `really_convertible_to` concept.

```cpp
using V = simd::vec<float>;

template <typename T> struct X { explicit operator T() const; };

static_assert(not   std::convertible_to<X<float>, V>);
static_assert(       std::convertible_to<float, V>);
static_assert(       std::convertible_to<short, V>);
static_assert(   really_convertible_to<short, V>);
static_assert(not   std::convertible_to<int, V>);
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // ill-formed
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // OK
  x = m;      // OK
```

```
  x = l;       // OK (because convertible_to<decltype(l), float> is true)
  x = f;       // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.2   MORE CONSTRAINED CONSTEXPR OVERLOAD

A possible solution selects the existing (`constexpr`) broadcast constructor for everything but the cases where the value of the argument needs to be checked. Thus, we need the existing constructor to always be *more constrained* ([temp.constr.order]) than the `consteval` constructor. The `consteval` constructor can then only be selected if the other constructor is not part of the candidate set at all (via `explicit`).

Sketch:

```
template <class From, class To>
  concept simd-consteval-broadcast-arg = constructible_from<To, From>;

template <class From, class To>
  concept simd-broadcast-arg = simd-consteval-broadcast-arg<From, To> and true;

template <class T>
class basic_vec
{
public:
  template <simd-broadcast-arg<T> U>
    constexpr explicit(see below) basic_vec(U&&); // #1
  template <simd-consteval-broadcast-arg<T> U>
    consteval basic_vec(U&&);                     // #2
    // Mandates: convertible_to<U, value_type> && is_arithmetic_v<remove_cvref_t<U>>
};
```

Now every explicit call to the broadcast constructor will always select #1. Implicit calls to the broadcast constructor will select #1 if the condition in the `explicit` specifier is `false`. Otherwise, #1 is not part of the candidate set and #2 is called. Thus, the condition on the `explicit` specifier determines whether the `consteval` overload is chosen or not.

```
static_assert(      std::convertible_to<X<float>, V>); // different to status quo
static_assert(      std::convertible_to<float, V>);
static_assert(      std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(      std::convertible_to<int, V>);      // different to status quo
static_assert(not really_convertible_to<int, V>);
```

```
static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // OK (different to status quo)
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // OK
  x = m;      // OK
  x = V(l);   // OK
  x = f;      // OK
  x = X<float>(); // ill-formed: static_assert failed (different reason to status quo)
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.3   more constrained consteval overload

A viable alternative involves the removal of explicit conversions from arithmetic types to `basic_-vec`. The `consteval` constructor is declared with additional constraints over the existing constructor (satisfies `convertible_to`, `is_arithmetic_v`, and not value-preserving conversion). This way the `consteval` constructor is always chosen if the conversion of the given (arithmetic) type to `value_-type` is not value-preserving. Otherwise, the `constexpr` overload is used.

Sketch:

```
template <class From, class To>
  concept simd-broadcast-arg = constructible_from<To, From>;

template <class From, class To>
  concept simd-consteval-broadcast-arg
    = simd-broadcast-arg<From, To> && convertible_to<From, To>
        && !value-preserving-convertible-to<From, To>;

template <class T>
class basic_vec
{
public:
  template <simd-broadcast-arg<T> U>
    constexpr explicit(see below) basic_vec(U&&); // #1
  template <simd-consteval-broadcast-arg<T> U>
    consteval basic_vec(U&&); // #2
```

```
};
```

Here, every explicit call to the broadcast constructor with an arithmetic type is equivalent to
an implicit conversion, since the `consteval` overload is viable and more constrained. Every type
with a value-preserving conversion to `T` will select #1 (because of the constraint on #2). Every non-
arithmetic type (notably, user-defined types with conversion operator to some arithmetic type) will
continue to work as today, since #2 is not viable.

```cpp
static_assert(not    std::convertible_to<X<float>, V>); // equal to status quo / different to above
static_assert(       std::convertible_to<float, V>);
static_assert(       std::convertible_to<short, V>);
static_assert(    really_convertible_to<short, V>);
static_assert(       std::convertible_to<int, V>);        // different to status quo / equal to above
static_assert(not really_convertible_to<int, V>);

static_assert(    std::constructible_from<V, X<float>>);
static_assert(not std::constructible_from<V, X<short>>);
static_assert(    std::constructible_from<V, float>);
static_assert(    std::constructible_from<V, short>);
static_assert(    std::constructible_from<V, int>);

V f(int n, short m, std::reference_wrapper<int> l, std::reference_wrapper<float> f)
{
  V x = '\1'; // OK
  x = 1;      // OK (different to status quo / equal to above)
  x = 0x5EAF00D; // ill-formed
  x = V(n);   // ill-formed (different to both)
  x = m;      // OK
  x = V(l);   // OK
  x = f;      // OK
  x = X<float>(); // ill-formed: no match for operator= (no known conversion …[])
  x = float(X<float>()); // OK (obvious)
  x = V(X<float>()); // OK
}
```

4.4   how to handle bad value-preserving casts

The `consteval` broadcast overload needs to be ill-formed if the argument value cannot be con-
verted to the value type without changing the value. This can be achieved via the mechanism used
in ([simd.bit]) for `bit_ceil`. The constructor would spell out a precondition followed by *Remarks*: An
expression that violates the precondition in the *Preconditions*: element is not a core constant ex-
pression ([expr.const]).

The alternative that was mentioned before is to throw an exception (at compile time). Since in
basically all cases such an exception would not be caught at compile time, the program becomes

ill-formed. The ability to catch the exception allowed me to hack up a `really_convertible_to` concept. But otherwise, the utility of using an exception here seems fairly limited. The main reason for using an exception is better diagnostics on ill-formed programs. If we decide to add the `consteval` constructor for C++26, then we might want to delay the new exception type for C++29, though.

5   DIFFERENCES

Differences between the status quo and the two alternatives above:

	status quo	Section 4.2	Section 4.3
`convertible_to<X<float>, V>`	false	true	false
`convertible_to<int, V>`	false	true	true
`x = 1;`	✘	✔	✔
`x = V(n);`	✔	✔	✘

Note that `X<float>` is never implicitly convertible to `vec<float>`, so the solution in Section 4.2 lies about that. Also while some values of constant expressions of type `int` are convertible to `vec<float>`, it is not true in general that `int` is convertible to `vec<float>`.

6   RECOMMENDATION

My recommendation is to go with the solution presented in Section 4.3 without a new exception type (Section 4.4) for C++26. This would roll back a small part of a recent change done by [P3430R3].

Rationale for my preference:

1. The explicit conversion from arithmetic types via broadcast constructor is significantly less important after implicit conversion from constant expressions becomes possible.

2. The new `consteval` overload cannot be fully constrained in the solution presented in Section 4.2, leading to incorrect answers on traits or in requires expressions.

3. This should be part of C++26 because it helps avoiding bugs in user code.

4. A new exception type is not important enough to add it to C++26 and it can easily be added later.

7   IMPLEMENTATION EXPERIENCE

Both solutions (and a lot more variants that were discarded) have been implemented and tested in my implementation.

8   PROPOSED POLLS

Poll: Something needs to be done for C++26. (If we do nothing, the design space is constrained and Section 4.3 would be a breaking change.)

SF	F	N	A	SA

Poll: Add a `consteval` broadcast overload for value-preserving conversions to C++26.

SF	F	N	A	SA

If the vote for C++26 failed:

Poll: Add a `consteval` broadcast overload for value-preserving conversions to C++29.

SF	F	N	A	SA

Otherwise *maybe* poll:

Poll: It is better to add a `consteval` broadcast overload for C++29 rather than C++26.

SF	F	N	A	SA

Note: The next poll makes sense for C++26, even if we only intend to add the new overload for C++29.

Poll: The new constructor overload should be fully constrained, which requires the removal of `explicit` (not value-preserving conversions) from the existing broadcast constructor.

SF	F	N	A	SA

Poll: Add a new exception type to C++26 that is thrown from the new `consteval` broadcast constructor.

SF	F	N	A	SA

Poll: Encourage a paper targeting C++29 on a new exception type that is thrown from the new `consteval` broadcast constructor.

SF	F	N	A	SA

9   WORDING FOR SECTION 4.2

9.1   feature test macro

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

9.2   modify [simd.expos]

In [simd.expos], insert:

_____ [simd.expos]

```
template<class V, class T> using make-compatible-simd-t = see below;  // exposition only

template<class From, class To>
  concept simd-consteval-broadcast-arg = constructible_from<To, From>;  // exposition only

template<class From, class To>
  concept simd-broadcast-arg =                              // exposition only
    simd-consteval-broadcast-arg<From, To> && true;

template<class V>
  concept simd-vec-type =                                   // exposition only
```

9.3   modify [simd.overview]

In [simd.overview], insert:

_____ [simd.overview]

```
// ([simd.ctor]), basic_vec constructors
template<classsimd-broadcast-arg U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
template<simd-consteval-broadcast-arg U>
  consteval basic_vec(U&& x)
template<class U, class UAbi>
  constexpr explicit(see below) basic_vec(const basic_vec<U, UAbi>&) noexcept;
```

9.4   modify [simd.ctor]

<div align="right">[simd.ctor]</div>

```
template<class simd-broadcast-arg U> constexpr explicit(see below) basic_vec(U&& value) noexcept;
```

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

2   *Constraints*: `value_type` satisfies `constructible_from<U>`.

2   *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

3   *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<simd-consteval-broadcast-arg U> consteval basic_vec(U&& x)
```

4   *Mandates*:

- `U` satisfies `convertible_to<value_type>`, and

- `is_arithmetic_v<U>` is `true`,

5   *Preconditions*: The value of `x` is equal to the value of `x` after conversion to `value_type`.

6   *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

7   *Remarks:* An expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

10   WORDING FOR SECTION 4.3

10.1   feature test macro

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

10.2   modify [simd.expos]

In [simd.expos], insert:

─── [simd.expos]

```
template<class V, class T> using make-compatible-simd-t = see below;  // exposition only

template<class From, class To>
  concept simd-broadcast-arg = constructible_from<To, From>;       // exposition only

template<class From, class To>
  concept simd-consteval-broadcast-arg = see below;                // exposition only

template<class V>
  concept simd-vec-type =                                          // exposition only
```

10.3   modify [simd.expos.defn]

In [simd.expos.defn], insert:

─── [simd.expos.defn]

```
template<class V, class T> using make-compatible-simd-t = see below;
```

8   Let `x` denote an lvalue of type `const T`.

9   *make-compatible-simd-t*`<V, T>` is an alias for

   - *deduced-vec-t*`<T>`, if that type is not `void`, otherwise
   - `vec<decltype(x + x), V::size()>`.

```
template<class From, class To> concept simd-consteval-broadcast-arg = see below;
```

10   *simd-consteval-broadcast-arg* subsumes *simd-broadcast-arg.*

11   Types `From` and `To` satisfy *simd-consteval-broadcast-arg* only if `convertible_to<From, To>` is satisfied, `remove_cvref_t<From>` is an arithmetic type, and the conversion from `remove_cvref_t<From>` to `To` is not value-preserving.

10.4   modify [simd.overview]

In [simd.overview], insert:

—— [simd.overview]

```
// ([simd.ctor]), basic_vec constructors
template<class simd-broadcast-arg U>
  constexpr explicit(see below) basic_vec(U&& value) noexcept;
template<simd-consteval-broadcast-arg U>
  consteval basic_vec(U&& x)
template<class U, class UAbi>
  constexpr explicit(see below) basic_vec(const basic_vec<U, UAbi>&) noexcept;
```

10.5   modify [simd.ctor]

—— [simd.ctor]

```
template<class simd-broadcast-arg U> constexpr explicit(see below) basic_vec(U&& value) noexcept;
```

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

13   *Constraints*: `value_type` satisfies `constructible_from<U>`.

13   *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

14   *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<simd-consteval-broadcast-arg U> consteval basic_vec(U&& x)
```

15   *Preconditions*: The value of `x` is equal to the value of `x` after conversion to `value_type`.

16   *Effects*: Initializes each element to the value of the argument after conversion to `value_type`.

17   *Remarks:* An expression that violates the precondition in the *Preconditions*: element is not a core constant expression ([expr.const]).

A   REALLY_CONVERTIBLE_TO DEFINITION

```cpp
template <typename To, typename From>
  consteval bool converting_limits_throws()
  {
    try {
      using L = std::numeric_limits<From>;
      [[maybe_unused]] To x = L::max();
      x = L::min();
      x = L::lowest();
    } catch(...) {
      return true;
    }
    return false;
  }


template <typename From, typename To>
  concept really_convertible_to = std::convertible_to<From, To>
                                    and not converting_limits_throws<To, From>();
```

B   BIBLIOGRAPHY

[P3430R3]   Matthias Kretz. *simd issues: explicit, unsequenced, identity-element position, and members of disabled simd*. ISO/IEC C++ Standards Committee Paper. 2025. URL: https://wg21.link/p3430r3.