Document Number: P1813R0
Date: 2019-08-02
Audience: LEWG, SG6, SG8, SG9,
Reply to: Christopher Di Bella
cjdb.ns@gmail.com

A Concept Design for the Numeric Algorithms

Contents

1 Introduction [intro]

"Every time someone asks why we didn't cover `<numeric>` and `<memory>` algorithms: We thought 187 pages of TS was enough."

—*Casey Carter*

1.1 Motivation [intro.motivation]

N3351[17] served as the basis for the Ranges TS[15], which was merged into the C++20 Working Paper[12][13]. N3351 focused on defining concepts for the standard library, which is achieved by looking at the use-cases that concepts are designed for: generic algorithms. Specifically, N3351 looked at pinning down the concepts relevant to the algorithms found in `<algorithm>` after C++11. All known bodies of work from N4128[14] through to P0896 and P0898 — with the exception of P1033[11] — have continued to focus on studying and refining the contents of `<algorithm>`. P1033 takes the extremely low-hanging fruit and adds the uninitialised-memory algorithms from `<memory>` to the mix. To the author's best knowledge, all that's left to be added are possibly a few algorithms introduced in C++20, and all of the algorithms in `<numeric>`.

The numeric algorithms weren't abandoned or forgotten: given the limited resources, there simply wasn't enough time to study all of the algorithms in `<algorithm>` and `<numeric>`, and also introduce the basis for range adaptors in C++20. Now that we're moving into the C++23 design space, we should start reviewing the numeric algorithms in the same light as N3351 considered the `<algorithm>` algorithms.

A complete design is not as simple as taking the concepts introduced in P0896, slapping them on the numeric algorithms, and calling it a day. These algorithms have different requirements to those in `<algorithm>`, and P1813 takes aim at what those might look like. The current revision chooses to focus on only those algorithms introduced in C++98 and `reduce`; the remaining C++17 numeric algorithms are left to a subsequent revision.

1.2 Design ideals [intro.design.ideals]

The following section has been lifted almost completely verbatim from N3351. This serves as a reminder that the design ideals have not really changed since N3351's publication in 2012. Non-editorial changes are represented by showing ~~what is present in N3351~~and what is present in P1813.

1. The concepts for the STL must be mathematically and logically sound. By this, we mean to emphasise the fact that we should be able to reason about properties of programs (e.g. correctness) with respect to the semantics of the language and the types used in those programs.
2. The concepts used should express general ideas in the application domain (hence the name 'concepts') rather than mere programming language artifacts. Thinking about concepts as a yet another 'contract' language can lead to partially formed ideas. Contracts force programmers to think about requirements on individual functions or interfaces, whereas concepts should represent fully formed abstractions.
3. The concepts should specify both syntactic and semantic requirements ("concepts are all about semantics" — Alex Stepanov). A concept without semantics only partially specifies an interface and cannot be reasoned about; the absence of semantics is the opposite of soundness ("it is insanity" — Alex Stepanov).
4. Symbols and identifiers should be associated with their conventional meanings. Overloads should have well defined semantics and not change the usual meaning of the symbol or name.
5. The concepts as used to specify algorithms should be terse and readable. An algorithm's requirements must not restate the syntax of its implementation.
6. The number of concepts used should be low, in order to make them easier to understand and remember.
7. An algorithm's requirements must not inhibit the use of very common code patterns in its implementation.
8. An algorithm should not contain requirements for syntax that it does not use, thereby unnecessarily limiting its generality.
9. The STL with concepts should be compatible with ~~C++11~~C++20, except where that compatibility would imply a serious violation of one of the first two aims.

The following quote has also been extracted from N3351.

> "Every generic library design must choose the style in which it describes template requirements. The ways in which requirements are specified has a direct impact on the design of the concepts used to express them, and (as always) there are direct consequences of that choice. For example, we could choose to state template requirements in terms of the exact syntax requirements of the template. This leads to concept designs that have large numbers of small syntactic predicates (e.g. `HasPlus`, `HasComma`, etc.). The benefit of this style of constraint is that templates are more broadly adaptable: there are potentially more conforming types with which the template will interoperate. On the downside, exact requirements tend to be more verbose, decreasing the likelihood that the intended abstraction will be adequately communicated to the libraryâĂŹs users. The C++0x design is, in many aspects, a product of this style.
>
> On the other end of the spectrum, we could choose to express requirements in terms of the required abstraction instead of the required syntax. This approach can lead to (far) fewer concepts in the library design because related syntactic requirements are grouped to create coherent, meaningful abstractions. Requirements can also be expressed more tersely, needing fewer concepts to express a set of requirements that describe how types are used in an algorithm. The use of abstract concepts also allows an algorithm to have more conforming implementations, giving a library author an opportunity to modify (i.e. maintain) a template's implementation without impacting its requirements. The obvious downside to this style is that it over-constrains templates; there may be types that conform to a minimal set of operations used by a template, but not the full set of operations required by the concept. The concepts presented in *Elements of Programming* approach this end of the spectrum."

Similarly to N3351, P1813 aims to hold itself in-between these two extremes.

1.3 Organisation [intro.organisation]

Similarly to N3351, P1813 is broken into a section for declaring algorithms with concept requirements, and a section for defining concepts. This document is intended to be read in sequentially, with many sections depending on exposition from previous sections.

Unlike N3351, P1813 does not introduce concept definitions at their first point-of-use: it instead sequentially defines them in the pre-wording-but-looks-like-wording Clause 3. P1813 also contains an appendix for proving mathematical assertions.

1.4 Assumed knowledge [intro.assumed.knowledge]

The following sections assume familiarity with the concepts library ([concepts]), the iterator concepts ([iterator.concepts]), the indirect callable requirements ([indirectcallable]), the common algorithm requirements ([alg.req]), the range requirements ([range.req]), and the way in which algorithms are specified in namespace `std::ranges` ([algorithms]).

Readers should consult *Design of concept libraries for C++*[18] prior to reading the remainder of P1813. Readers are also encouraged to consult *Elements of Programming*[16] and N3351 as necessary.

1.5 Implementation [intro.implementation]

This design has partially been implemented in cmcstl2. The original design and the ideas articulated in this document have slightly diverged, but not to the point where the author is convinced that the design has become un-implementable.

The author also hopes to implement this in range-v3 for broader coverage.

1.6 Target vehicle [intro.target.vehicle]

P1813 targets C++23.

1.7 Acknowledgements [intro.acknowledgements]

The author would like to thank Andrew Sutton, Ben Deane, Nicole Mazzuca, Nathaniel Shead, and Steve Downey reviewing this document, and providing valuable feedback. The author would also like to thank Arien Judge for reviewing the proofs in Annex 4.

2 Algorithms [algorithms]

"We start with algorithms because it is algorithms we want to specify cleanly, precisely, completely, and readably. If we can specify algorithms well, our concepts and the language mechanisms we use to specify the concepts are adequate. If not, no amount of sophistication in language mechanisms will help us."

—*N3351, §2*

"Generic Programming pro tip: Although Concepts are constraints on types, you don't find them by looking at the types in your system. You find them by studying the algorithms."

—*Eric Niebler, Twitter*

2.1 Where do the numeric algorithms belong? [algorithms.home]

The numeric algorithms have lived in `<numeric>` since the original implementation of STL, yet many developers frequently question why they are not found in `<algorithm>`. With standard module units on the horizon, it might seem pointless to discuss the validity of choosing to separate the numeric algorithms from the rest of their kin. This section aims to provide some guidance for when the Library Evolution group ultimately drafts the design for which entities reside in what module units.

[Note to reviewers: This section is pending reviewer input.]

2.2 Sequenced numeric algorithms [algorithms.sequenced]

The 'sequenced numeric algorithms' are the algorithms found in `<numeric>`, introduced in the STL; most of which made their way into C++98. This family of algorithms performs computations in a sequential manner, from left-to-right, or from the first element in the sequence to the last. For some binary operation `bop`, and two expressions `x` and `y`, the expression `bop(x, y)` need only be equality-preserving ([concepts.equality]); the expression `bop(x, y)` doesn't need to be *associative*, nor does it need to be *commutative*. That is, `bop(x, bop(y, z))` is not required to return the same result as `bop(bop(x, y), z)`, and `bop(x, y)` does not need to return the same result as `bop(y, x)`.

1 [*Example*: Addition is an associative operation: $1 + (2 + 3) = 6$ and $(1 + 2) + 3 = 6$ also. Subtraction is not an associative operation: $1 - (2 - 3) = 2$ and $(1 - 2) - 3 = -4$. — *end example*]

2 [*Example*: Addition is an commutative operation: $1 + 2 = 3$ and $2 + 1 = 3$ also. Subtraction is not a commutative operation: $1 - 2 = -1$ and $2 - 1 = 1$. — *end example*]

3 [*Note*: An operation does not need to be both associative and commutative; nor does it need to be neither. [*Example*: Matrix multiplication is associative, but not commutative[19]. — *end example*] — *end note*]

2.2.1 Accumulate [algorithms.accumulate]

`accumulate` is an algorithm that performs a fold operation, or in other words, takes a sequence of values and reduces them into a single value according to some operation. This is a generalisation of a summation. The algorithm — modelled after what's currently in the International Standard — has a fairly straightforward declaration.

```
template<input_iterator I, sentinel_for<I> S, movable T, class Proj = identity,
         indirect_magma<const T*, projected<I, Proj>, T*> BOp = ranges::plus>
constexpr accumulate_result<I, T>
  accumulate(I first, S last, T init, BOp bop = {}, Proj proj = {});

template<input_range R, movable T, class Proj = identity,
         indirect_magma<const T*, projected<iterator_t<R>, Proj>, T*> BOp = ranges::plus>
constexpr accumulate_result<safe_iterator_t<R>, T>
  accumulate(R&& r, T init, BOp bop = {}, Proj proj = {});
```

1 A *magma* is a binary operation `bop` over a set of elements S, where the result of `bop(x, y)` is also in the set, or alternatively, `bop` is closed under S[2]. Because different types may represent the same set of

elements (e.g. all of `int`, `long long`, and `double` can be used to represent a subset of integers), `BOp` does not need to be a homogeneous binary operation. For equational reasoning purposes, the types are expected to have a common type, and so `bop(0, vector0)` does not model a `magma`. Similarly, `bop(x, y)`, where `x` and `y` are possibly different types is expected to share a type common to both `x` and `y`. The type of `bop(x, y)` must be the same as the type of `bop(y, x)`. magma also requires `BOp` to model `regular_invocable`. Finally, a magma is only concerned with closure: they do not impose any requirements on associativity, nor on commutativity, so although the types of `bop(x, y)` and `bop(y, x)` need to match, there is no requirement for their values to match.

It might also be nice to use `accumulate` without an initial value, similarly to C++17's `std::reduce` ([reduce]). It would certainly be convenient to use `accumulate(r)` or even `accumulate(r, ranges::times{})`, where `r` is an arbitrary range, and `ranges::times` is a modernisation of `std::multiplies`. The former is fairly trivial to do: we can default `init = T{}` and call it a day, just as `std::reduce` has, but the author feels that this is lacking. An *ideal* `init`-less `accumulate` should permit the caller to specify a range, optionally an operation, and optionally a projection. This requires great care, because `accumulate(r, times{})` when `init = T{}` would always produce a single result: `T{}` (recall that `T{}` is equivalent to zero for fundamental types). The reasons for why this is not desirable should be obvious.

By instead choosing an appropriate way to represent an operation's identity element, `accumulate(r, bop, proj)` becomes a viable candidate to add to our overload set. An *identity element id* is an element in a set S, where $x \cdot id$ is equivalent to x, or $id \cdot x$ is equivalent to x. $x \cdot id$ is called a *right-identity*, because id is on the right-hand-side of x, and $id \cdot x$ is called a *left-identity*. When id is both a left-identity and a right-identity, we call it a *two-sided identity*[6] (mathematicians should note that `std::identity` is a function object ([func.identity])).

2 [*Example*: 0 is the two-sided identity element for addition of real numbers: $x + 0 = x = 0 + x = x$. — *end example*]

3 [*Example*: 1 is the two-sided identity element for multiplication of real numbers: $1(x) = x = x(1) = x$. — *end example*]

With an interface that requires a two-sided identity, we can now declare our additions to the `accumulate` overload set.

```
template<input_iterator I, sentinel_for<I> S, class Proj = identity,
         indirect_monoid<projected<I, Proj>, projected<I, Proj>,
                         iter_value_t<projected<I, Proj>>*> BOp = ranges::plus>
  requires movable<iter_value_t<projected<I, Proj>>>
constexpr accumulate_result<I, iter_value_t<projected<I, Proj>>>
  accumulate(I first, S last, BOp bop = {}, Proj proj = {});

template<input_range R, class Proj = identity,
         indirect_monoid<projected<iterator_t<R>, Proj>,
                         projected<iterator_t<R>, Proj>,
                         iter_value_t<projected<iterator_t<R>, Proj>>* = ranges::plus>
  requires movable<iter_value_t<projected<iterator_t<R>, Proj>>>
constexpr accumulate_result<safe_iterator_t<R>, iter_value_t<projected<iterator_t<R>, Proj>>>
  accumulate(R&& r, BOp bop = {}, Proj proj = {});
```

A *monoid* is a twice-removed refinement over magma: it requires `BOp` be an associative operation (this is a *semigroup*[9]), and it requires that `BOp` have a two-sided identity element[7]. How this is achieved is covered later, but it is a good idea to note now that the notion of identities are defined using a new set of traits (numeric traits). This overload subset designates the return type to be the same as the iterator's value type, so the requirement for `T` to be `movable` must be moved appropriately.

2.2.2 Partial sum [algorithms.partial.sum]

In mathematics, a partial sum is a summation of the first N elements of a sequence[20].

$$S_N = \sum_{k=0}^{N-1} a_k$$

The C++ algorithm `partial_sum` is a generalisation of a partial sum, which writes the kth evaluation of `accumulate` to an output range. The interface is extremely similar to that of `accumulate`.

```
template<input_iterator I, sentinel_for<I> S1, weakly_incrementable O, sentinel_for<O> S2,
         class Proj = identity,
         indirect_magma<projected<I, Proj>, projected<I, Proj>, O> BOp = ranges::plus>
  requires indirectly_copyable_storable<I, O>
constexpr partial_sum_result<I, O>
  partial_sum(I first, S1 last, O result, S2 result_last, BOp bop = {}, Proj proj = {});

template<input_range R, range O, class Proj = identity,
         indirect_magma<projected<iterator_t<R>, Proj>,
                        projected<iterator_t<R>, Proj>,
                        iterator_t<O>> BOp = ranges::plus>
  requires indirectly_copyable_storable<iterator_t<I>, iterator_t<O>>
constexpr partial_sum_result<safe_iterator_t<R>, safe_iterator_t<O>>
  partial_sum(R&& r, O&& result, BOp bop = {}, Proj proj = {});
```

Unlike `accumulate`, `partial_sum` doesn't require an initial value: it instead designates `invoke(proj, *first)` as the initial value. `partial_sum` requires its binary operation model a `magma` over its projected input range for the same reasons as `accumulate`. The output of `partial_sum`'s value-type must be copyable, and movable via a cache ([alg.req.ind.copy]).

[Note to reviewers: The above paragraph is poorly worded. Input on how to rephrase is appreciated.]

To minimise the likelihood of writing to a beyond an output range that is smaller than the input range, both overloads have been slightly altered to take two full ranges instead of a range-and-a-half. The range-and-a-half overloads can be emulated using `unreachable_t`.

2.2.3 Adjacent difference [algorithms.adjacent.difference]

`adjacent_difference` is a specialised transformation over adjacent elements in an input range to compute the inverse of a `partial_sum` (4.1). This yields some interesting properties about `adjacent_difference`'s requirements, as shown below.

```
template<input_iterator I, sentinel_for<I> S1, weakly_incrementable O, sentinel_for<O> S2,
         class Proj = identity,
         indirect_loop<projected<I, Proj>, projected<I, Proj>, O> BOp = ranges::minus>
  requires requires indirectly_copyable_storable<I, O>
constexpr adjacent_difference_result<I, O>
  adjacent_difference(I first, S1 last, O result, S2 result_last, BOp bop = {}, Proj proj = {});

template<input_range R, range O, class Proj = identity,
         indirect_loop<projected<iterator_t<R>, Proj>,
                       projected<iterator_t<R>, Proj>,
                       iterator_t<O>> BOp = ranges::minus>
  requires indirectly_copyable_storable<iterator_t<I>, iterator_t<O>>
constexpr adjacent_difference_result<safe_iterator_t<R>, safe_iterator_t<O>>
  adjacent_difference(R&& r, O&& result, BOp bop = {}, Proj proj = {});
```

1 A *loop* is another twice-removed refinement over a magma. Specifically, it requires that the binary operation have an inverse operation (this is a *quasigroup*[8]), and a two-sided identity. It is necessary for `adjacent_difference` to require a loop, so that we can guarantee that it is the inverse algorithm of `partial_sum`.

It's important to note that despite appearing to have similar use-cases, both the the interface and implementation for `adjacent_difference` are distinct from `transform` ([alg.transform]):

Varying interface `adjacent_difference` requires that its binary operation model `regular_invocable` ([concept.regularinvocable]), while `transform` only requires its binary operation model `invocable` ([concept.invocable]) ([indirectcallable.indirectinvocable]).

Varying implementation `transform` applies its operands in the order of left-to-right, à la `op(*first1, *first2)`, while `adjacent_difference` applies its operands in the *opposite* order, à la `bop(prev, *first)`.

[Note to reviewers: Despite *loop* being the technical term for this algebraic structure, the author does not encourage the using the name `loop` *directly*, due to the likelihood of it being confused with the computer science term 'loop'. See 3.2.6 for possible alternative (ugly) names, and 3.2.7 for a possible (and preferred) redesign.]

2.2.4 Inner Product [algorithms.inner.product]

`inner_product` generalises an algebraic inner product into a map-reduce operation.

```
template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
         movable T, class Proj1 = identity, class Proj2 = identity,
         class BOp1 = ranges::plus, class BOp2 = ranges::times>
  requires indirect_weak_magmaring<BOp1, BOp2, const T*,
                                   projected<I1, Proj1>, projected<I2, Proj2>, T*>
constexpr inner_product_result<I1, I2, T>
 inner_product(I1 first1, S1 last1, I2 first2, S2 last2, T init,
               BOp1 bop1 = {}, BOp2 bop2 = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, class BOp1 = ranges::plus, class BOp2 = ranges::times,
         movable T, class Proj1 = identity, class Proj2 = identity>
  requires indirect_weak_magmaring<BOp1, BOp2, const T*,
                                   projected<iterator_t<R1>, Proj1>,
                                   projected<iterator_t<R2>, Proj2>, T*>
constexpr inner_product_result<safe_iterator_t<R1>, safe_iterator_t<R2>, T>
  inner_product(R1&& r1, R2&& r2, T init, BOp1 bop1 = {}, BOp2 bop2 = {},
                Proj1 proj1 = {}, Proj2 proj2 = {});
```

1 A `weak_magmaring` is an extreme generalisation of the well-known semiring algebraic structure that establishes a relationship between two magmas. Specifically, it describes that a magma `BOp2` is *distributive* over `BOp1`. Given three objects of possibly distinct — but related — types, `x`, `y`, and `z`, the expression `bop2(x, bop1(y, z))` is equivalent to `bop1(bop2(x, y), bop2(x, z))`.

2 [*Example*: Multiplication is distributive over addition: $x(y + z) = xy + yz$. — *end example*]

3 Mathematicians note that weak-magmaring is a generalisation of a near-semiring, named by the author, to fit the requirements. The author asked around on StackExchange[10] before naming this algebraic structure, but it seems that the structure is too general to be of interest outside of this use-case. The naming decision stems from that fact that a near-semiring weakens $(S, \cdot)$ from a monoid to a semigroup, and a weak-magmaring weakens $(S, \cdot)$ from a semiring to a magma. A more appropriate name might exist: near-semirings still require $(S, +)$ to model a monoid, but a near-magma weakens this requirement to a magma as well.

4 Similarly to `adjacent_difference`, `inner_product` is not quite the same as C++17's `transform_reduce`, which is expected to be far more permissive with its operations.

Care has been taken to ensure that `inner_product` is not over-constraining, and that only types that directly interact are required to have a common type. This means the following code doesn't meet the requirements for `inner_product`.

```
auto words_to_ints = [](string_view const word) -> int {
   // ...
};
auto const data1 = vector{"one"s, "two"s, "three"s, "four"s, "five"s};
auto const data2 = vector{"six"s, "seven"s, "eigth"s, "nine"s, "ten"s};
return inner_product(data1, data2, 0, ranges::plus{}, words_to_ints);
// error: words_to_ints doesn't model magma<string, string>, since
// common_with<invoke_result_t<words_to_ints, string, string>, int> is false.
```

A user that wants to perform this operation should instead use the following:

```
auto as_words = view::transform(words_to_ints);
return accumulate(view::zip_with(data1 | as_words, data2 | as_words, ranges::times{}));
```

[Note to reviewers: The author painfully is aware that `zip_with_view` is yet to be standardised: this use-case exasperates the need for such a library feature.]

Similarly to `accumulate`, by refining our requirements, it's possible to eliminate the need for an initial value, thereby making this possible:

```
auto ints = view::iota(0);
auto slice = [](auto const drop, auto const take) {
   return view::drop(drop) | view::take(take);
};
return inner_product(ints | slice(100, 10), ints | slice(10, 10));
```

```
template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
         class BOp1 = ranges::plus, class BOp2 = ranges::times,
         class Proj1 = identity, class Proj2 = identity>
requires indirect_near_semiring<BOp1, BOp2,
                                const iter_value_t<projected<I1, Proj1>>*,
                                projected<I1, Proj1>,
                                projected<I2, Proj2>,
                                iter_value_t<projected<I1, Proj1>>*>
constexpr inner_product_result<I1, I2, iter_value_t<projected<I1, Proj1>>>
inner_product(I1 first1, S1 last1, I2 first2, S2 last2, BOp1 bop1 = {}, BOp2 bop2 = {},
              Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, class Proj1 = identity, class Proj2 = identity,
         class BOp1 = ranges::plus, class BOp2 = ranges::times>
requires indirect_near_semiring<BOp1, BOp2,
                                const iter_value_t<projected<iterator_t<R1>, Proj1>>*,
                                projected<iterator_t<R1>, Proj1>,
                                projected<iterator_t<R2>, Proj2>,
                                iter_value_t<projected<iterator_t<R1>, Proj1>>*>
constexpr inner_product_result<safe_iterator_t<R1>, safe_iterator_t<R2>,
                               iter_value_t<projected<iterator_t<R1>, Proj1>>>
inner_product(R1&& r1, R2&& r2, BOp1 bop1 = {}, BOp2 bop2 = {},
              Proj1 proj1 = {}, Proj2 proj2 = {});
```

A *near-semiring* is a refinement of a weak-magmaring, and naturally arises from studying functions on monoids[1]. A near-semiring requires that `BOp1` model a monoid, and that `BOp2` model a semigroup. As a near-semiring refines a weak-magmaring, it subsumes the distributive property. It also introduces the notion of an *annhiliating element*[4]. In mathematics, an annihilating element is a special element in a set for certain operations, such that when applied with any other element in the set, the result of the operation is the annihliating element. It is the complete opposite of an identity element.

[*Example*: Scalar multiplication's annihilating element is 0: $0x = 0$ and $x0 = 0$. — *end example*]

Semigroup theory refers to annihilating elements as the *zero element*, as there is only one notion of zero.

[Note to reviewers: While a zero element is not strictly a necessity for `inner_product`, it is a fundamental property of a near-semiring, and so it has been included in the requirements for a `near_semiring`.]

2.2.5 Iota [algorithms.iota]

[Note to reviewers: As the C++20 WP contains `iota_view`, it is unclear to the author whether or not there is a place for an algorithm `iota`. This subsection will be filled out, either in favour or against, after receiving guidance.]

2.2.6 Power [algorithms.power]

[Note to reviewers: While reviewing the history of the original STL implementation, the author noted that there existed an extension algorithm called `power`. The current revision of this document does not explore this algorithm, but a future revision may.]

2.3 Unsequenced numeric algorithms [algorithms.unsequenced]

The 'unsequenced numeric algorithms' are the `<numeric>` algorithms introduced in C++17. These are a further generalisation of the sequenced numeric algorithms, and may perform computations out-of-order. As such, in order to guarantee equality-preservation, these algorithms will require their operations be both associative and commutative.

2.3.1 Reduce [algorithms.reduce]

`reduce` is the unsequenced counterpart to `accumulate`. Its declaration is fairly similar to that of `accumulate`, except for the refinements introduced by this section.

```
template<input_iterator I, sentinel_for<I> S, movable T, class Proj = identity,
         indirect_commutative_semigroup<const T*, projected<I, Proj>, T*> BOp = ranges::plus>
constexpr reduce_result<I, T>
  reduce(I first, S last, T init, BOp bop = {}, Proj proj = {});
```

```
template<input_range R, movable T, class Proj = identity,
         indirect_commutative_semigroup<const T*,
                                        projected<iterator_t<R>, Proj>, T*> BOp = ranges::plus>
constexpr reduce_result<safe_iterator_t<R>, T>
  reduce(R&& r, T init, BOp bop = {}, Proj proj = {});

template<input_iterator I, sentinel_for<I> S, class Proj = identity,
         indirect_commutative_monoid<projected<I, Proj>, projected<I, Proj>,
                                     iter_value_t<projected<I, Proj>>*> BOp = ranges::plus>
requires movable<iter_value_t<projected<I, Proj>>>
constexpr reduce_result<I, iter_value_t<projected<I, Proj>>>
  reduce(I first, S last, BOp bop = {}, Proj proj = {});

template<input_range R, class Proj = identity,
         indirect_commutative_monoid<projected<iterator_t<R>, Proj>,
                                     projected<iterator_t<R>, Proj>,
                                     iter_value_t<projected<iterator_t<R>, Proj>>* = ranges::plus>
requires movable<iter_value_t<projected<iterator_t<R>, Proj>>>
constexpr reduce_result<safe_iterator_t<R>, iter_value_t<projected<iterator_t<R>, Proj>>>
  reduce(R&& r, BOp bop = {}, Proj proj = {});
```

`commutative_semigroup` and `commutative_monoid` respectively refine `semigroup` and `monoid` so that `BOp` is a commutative operation. This is achieved by introducing a `commutative_operation` concept, which requires that for two distinct values `x` and `y`, `bop(x, y)` is has same result as `bop(y, x)`.

[Note to reviewers: This document does not yet define the concepts `commutative_semigroup` and co, but one can 'imagine' them being equivalent to `semigroup<T, U> && commutative_operation<T, U>`, etc.]

2.3.2 Inclusive scan [algorithms.inclusive.scan]

This revision does not explore the requirements for `inclusive_scan`.

2.3.3 Exclusive scan [algorithms.exclusive.scan]

This revision does not explore the requirements for `exclusive_scan`.

2.3.4 Transform reduce [algorithms.transform.reduce]

This revision does not explore the requirements for `transform_reduce`.

3 Algorithm support [support]

> "Generic Programming pro tip #2: The "basis operations" of a well-designed concept or concept hierarchy is the minimal set of operations that are both sufficient and necessary for efficiently implementing all algorithms of interest within a particular domain."
>
> —*Eric Niebler, Twitter*

The following subsections articulate the concept designs and any supporting material (such as traits).

[Note to reviewers: This section's 'wording' is not intended to be reviewed for wording (hence why the chapter isn't titled 'Proposed Wording').]

3.1 Numeric traits [support.traits]

This section provides exposition for the traits that are used by algebraic concepts. The author is aware that the design is not necessarily the most appropriate, and is open to suggestions for improvement.

3.1.1 Identity traits [support.traits.identity]

1 An identity element *id* is a special element in a set S, such that for all other elements x in S, given a magma $\cdot$, at least one of $x \cdot id = x$ or $id \cdot x = x$ holds.

2 If both $x \cdot id = x$ and $id \cdot x = x$ hold, then *id* is unique (4.2).

3.1.1.1 Left identity [support.traits.identity.left]

1 `left_identity` is a type that represents the notion of a *left-identity*.

```
namespace std {
  template<class BOp, class T, class U = T>
  struct left_identity {};

  template<class BOp, class T, class U = T>
  using left_identity_t = decltype(declval<left_identity<BOp, T, U>>()());
}
```

2 A program may specialise `left_identity`, as described in the example below. No diagnostic is required for specialisations that do not follow this implementation. No diagnostic is required for explicit specialisations of the template parameters `T` or `U`. [*Example*:

```
struct binary_op {
  int operator()(int, int) const;
};

namespace std {
  template<class T, class U>
  requires magma<binary_op, T, U>
  struct left_identity<binary_op, T, U> {
    constexpr common_type_t<T, U> operator()() const
    { /* implementation-defined */ }
  };
}
```

— *end example*]

3.1.1.2 Right identity [support.traits.identity.right]

1 `right_identity` is a type that represents the notion of a *right-identity*.

```
namespace std {
  template<class BOp, class T, class U = T>
  struct right_identity {};
```

```
  template<class BOp, class T, class U = T>
  using right_identity_t = decltype(declval<right_identity<BOp, T, U>>()());
}
```

2 A program may specialise `right_identity`, as described in the example below. No diagnostic is required for specialisations that do not follow this implementation. No diagnostic is required for explicit specialisations of the template parameters `T` or `U`. [*Example*:

```
struct binary_op {
  int operator()(int, int) const;
};

namespace std {
  template<class T, class U>
  requires magma<binary_op, T, U>
  struct right_identity<binary_op, T, U> {
    constexpr common_type_t<T, U> operator()() const
    { /* implementation-defined */ }
  };
}
```

— *end example*]

3.1.1.3 Two-sided identity [support.traits.identity.two.sided]

1 `two_sided_identity` is a type that represents the notion of a *two-sided identity*.

```
namespace std {
  template<class BOp, class T, class U>
  concept has-two-sided-identity = // exposition only
    requires(BOp bop, const T& t, const U& u) {
      typename left_identity_t<BOp, T, U>;
      typename left_identity_t<BOp, U, T>;
      typename right_identity_t<BOp, T, U>;
      typename right_identity_t<BOp, U, T>;

      requires same_as<left_identity_t<BOp, T, U>, left_identity_t<BOp, U, T>>;
      requires same_as<right_identity_t<BOp, T, U>, right_identity_t<BOp, U, T>>;
      requires same_as<left_identity_t<BOp, T, U>, right_identity_t<BOp, T, U>>;

    };
}
```

2 Let `left1` be an object of type `left_identity<BOp, T, U>`, `left2` be an object of type `left_identity<BOp, U, T>`, `right1` be an object of type `right_identity<BOp, T, U>`, and `right2` be an object of type `right_identity<BOp, U, T>`.

3 The expressions `left1() == left2()`, `right1() == right2()`, and `left1() == right1()` are all `true`.

4 If `t != left1()` is `true` and `u != right1()` is `true`, then the expressions `t == invoke(bop, t, right())` and `u == invoke(bop, left(), u)` are both `true`.

```
  template<class BOp, class T, class U = T>
  requires has-two-sided-identity<BOp, T, U>
  struct two_sided_identity {
    constexpr common_type_t<T, U> operator()() const;
  };

  template<class BOp, class T, class U>
  using two_sided_identity_t =
    decltype(two_sided_identity{}(declval<BOp&>(), declval<T>(), declval<U>()));
```

```
constexpr common_type_t<T, U> operator()(BOp bop, T&& t, U&& u) const;
```

5 *Expects:* `left_identity<BOp, T, U>() == right_identity<BOp, T, U>()` is `true`.

6 *Effects:* Equivalent to:

```
        return left_identity<BOp, T, U>{}();
```

3.1.2 Zero traits [support.traits.zero]

1 A zero element z is a special element in a set S, such that for all other elements x in S, given a magma $\cdot$, at least one of $x \cdot z = z$ or $z \cdot x = z$ holds.

2 If both $x \cdot z = z$ and $z \cdot x = z$ hold, then z is unique (4.3).

3.1.2.1 Left zero [support.traits.zero.left]

1 `left_zero` is a type that represents the notion of a *left-zero*.

```
namespace std {
  template<class BOp, class T, class U = T>
  struct left_zero {};

  template<class BOp, class T, class U = T>
  using left_zero_t = decltype(declval<left_zero<BOp, T, U>>()());
}
```

2 A program may specialise `left_zero`, as described in the example below. No diagnostic is required for specialisations that do not follow this implementation. No diagnostic is required for explicit specialisations of the template parameters `T` or `U`. [*Example*:

```
struct binary_op {
  int operator()(int, int) const;
};

namespace std {
  template<class T, class U>
  requires magma<binary_op, T, U>
  struct left_zero<binary_op, T, U> {
    constexpr common_type_t<T, U> operator()() const
    { /* implementation-defined */ }
  };
}
```

— *end example*]

3.1.2.2 Right zero [support.traits.zero.right]

1 `right_zero` is a type that represents the notion of a *right-zero*.

```
namespace std {
  template<class BOp, class T, class U = T>
  struct right_zero {};

  template<class BOp, class T, class U = T>
  using right_zero_t = decltype(declval<right_zero<BOp, T, U>>()());
}
```

2 A program may specialise `right_zero`, as described in the example below. No diagnostic is required for specialisations that do not follow this implementation. No diagnostic is required for explicit specialisations of the template parameters `T` or `U`. [*Example*:

```
struct binary_op {
  int operator()(int, int) const;
};

namespace std {
  template<class T, class U>
  requires magma<binary_op, T, U>
  struct right_zero<binary_op, T, U> {
    constexpr common_type_t<T, U> operator()() const
    { /* implementation-defined */ }
  };
}
```

— *end example*]

3.1.2.3 Two-sided zero [support.traits.zero.two.sided]

1 `two_sided_zero` is a type that represents the notion of a *two-sided zero*.

```
namespace std {
  template<class BOp, class T, class U>
  concept has-two-sided-zero = // exposition only
    requires(BOp bop, const T& t, const U& u) {
       typename left_zero_t<BOp, T, U>;
       typename left_zero_t<BOp, U, T>;
       typename right_zero_t<BOp, T, U>;
       typename right_zero_t<BOp, U, T>;

       requires same_as<left_zero_t<BOp, T, U>, left_zero_t<BOp, U, T>>;
       requires same_as<right_zero_t<BOp, T, U>, right_zero_t<BOp, U, T>>;
       requires same_as<left_zero_t<BOp, T, U>, right_zero_t<BOp, T, U>>;
    };
}
```

2 Let `left1` be an object of type `left_zero<BOp, T, U>`, `left2` be an object of type `left_zero<BOp, U, T>`, `right1` be an object of type `right_zero<BOp, T, U>`, and `right2` be an object of type `right_zero<BOp, U, T>`.

3 The expressions `left1() == left2()`, `right1() == right2()`, and `left1() == right1()` are all `true`.

4 If `t != left1()` is `true` and `u != right1()` is `true`, then the expressions `right1() == invoke(bop, t, right1())` and `left1() == invoke(bop, left1(), u)` are both `true`.

```
template<class BOp, class T, class U>
requires has-two-sided-zero<BOp, T, U>
struct two_sided_zero {
   constexpr common_type_t<T, U> operator()(BOp bop, T&& t, U&& u) const;
};

template<class BOp, class T, class U = T>
using two_sided_zero_t =
  decltype(two_sided_zero{}(declval<BOp&>(), declval<T>(), declval<U>()));
```

```
constexpr common_type_t<T, U> operator()(BOp bop, T&& t, U&& u) const;
```

5 *Expects:* `left_zero<BOp, T, U>() == right_zero<BOp, T, U>()` is `true`.

6 *Effects:* Equivalent to:

```
return left_zero<BOp, T, U>();}
```

3.1.3 Inverse traits [support.traits.inverse]

[Note to reviewers: The design for `inverse_traits` needs to be thoroughly revised, as it is overly restrictive, incomplete, and wrong.]

1 `inverse_traits` denotes a type that takes an object modelling a `magma` over an arbitrary domain as input, and returns an object whose type models a `magma` over the same domain, where the returned object is the mathematical inverse operation of the input.

```
template<class BOp>
struct inverse_traits {};

template<class BOp>
using inverse_operation_t = typename inverse_traits::type;
```

2 Users may specialise this type, provided it is equivalent to:

```
struct plus;
struct minus;
```

```
namespace std {
  template<>
  struct inverse_traits<plus> {
     using type = minus;
     constexpr type operator()() const { return type{}; }
  };

  template<>
  struct inverse_traits<minus> {
     using type = plus;
     constexpr type operator()() const { return type{}; }
  };
}
```

3 Let `x` be an object of type `T`, `y` be an object of type `U`, `bop` be an object of type `BOp`, where `BOp` models `magma<T, U>`, and `inv` be an object of type `Inv`, where `Inv` models `magma<T, U>`.

4 *Mandates:*

(4.1) — If `is_same_v<inverse_type_t<BOp>, Inv>` is `true`, then `is_same_v<inverse_type_t<Inv>, BOp>` is also `true`, no diagnostic required.

(4.2) — `is_same_v<inverse_type_t<BOp>, BOp>` is `false`, no diagnostic required.

5 *Expects:*

(5.1) — `invoke(inv, invoke(bop, x, y), y) == x` is `true`.

(5.2) — `invoke(inv, invoke(bop, x, y), x) == y` is `true`.

[Note to reviewers: This type could be relaxed to permit the inverse of other operations. For example, the inverse of $-x$ is $-(-x)$, and the inverse of `swap(x, y)` is `swap(x, y)`.]

3.2 Algebraic concepts [support.concepts]

This section describes the concepts required by the numeric algorithms, and are not present in the C++20 standard library.

3.2.1 Concept commutative operation [support.concepts.commutative.op]

1 A commutative operation is a binary operation where the order of its operands does not change the evaluation of the expression.

2 [*Example*: Integral arithmetic is commutative. — *end example*]

3 [*Example*: Matrix multiplication is not commutative. — *end example*]

```
template<class BOp, class T, class U>
concept commutative_operation =
  regular_invocable<BOp, T, U> &&
  regular_invocable<BOp, U, T> &&
  common_with<T, U> &&
  equality_comparable_with<T, U>;
```

4 Let `bop` be an object of type `BOp`, `t` be an object of type `T`, and `u` be an object of type `U`, where `t != u`.

5 The result of `invoke(bop, t, u)` is expression-equivalent to `invoke(bop, u, t)`.

3.2.2 Concept magma [support.concepts.magma]

1 A *magma* is a set S associated with a binary operation $\cdot$, such that S is *closed under* $\cdot$.

2 [*Example*: $(\mathbb{Z}, +)$ is a magma, since we can add any two integers and find that the result is also an integer. — *end example*]

3 [*Example*: $(\mathbb{Z}, /)$ is not a magma, since $\frac{2}{3}$ is not an integer. — *end example*]

[Note to reviewers: The term *semigroupoid* is an older name for the notion of a magma, but this seems to have been co-opted by category theory.]

```
template<class BOp, class T, class U>
concept magma =
  common_with<T, U> &&
  regular_invocable<BOp, T, T> &&
  regular_invocable<BOp, U, U> &&
  regular_invocable<BOp, T, U> &&
  regular_invocable<BOp, U, T> &&
  common_with<invoke_result_t<BOp&, T, U>, T> &&
  common_with<invoke_result_t<BOp&, T, U>, U> &&
  same_as<invoke_result_t<BOp&, T, U>, invoke_result_t<BOp&, U, T>>;
```

4 Let `bop` be an object of type `BOp`, `t` be an object of type `T`, and `u` be an object of type `U`.

5 The value `invoke(bop, t, u)` must return a result that is representable by `common_type_t<T, U>`.

The decision to require common types for a over `magma<T, U>` is similar to the reason that `equality_-comparable_with` requires `common_reference_with`: this ensures that when an algorithm requires a `magma`, we are able to *equationally reason* about those requirements. It's possible to overload `operator+(int, vector<int> const&)`, but that doesn't follow the canonical usage of +. Does `1 + vector{1, 2, 3}` mean 'concatenate `vector{1, 2, 3}` to the end of a temporary `vector{1}`'? Is it a shorthand for `accumulate(vector{1, 2, 3}, 1)`? The intention is unclear, and so `std::plus<>` (sic) should not model `magma<int, vector<int>>`.

[Note to reviewers: This implies that the author is not a fan of + as a way of concatenating strings. It's not ideal, but since this is already a standard practice, the author has chosen not to — probably fruitlessly — wage war on this specific case.]

Similarly, if the result type is not related to its parameters, the operation lacks the ability to be equationally reasoned about. While a programmer might have a good local reason to write a matrix multiplication declared as `mat16 operator*(mat4 const& x, mat4 const&)`, this does not adhere to the usual rules of mathematics, and is out-of-scope for generic programming.

3.2.3 Concept semigroup [support.concepts.semigroup]

1 A *semigroup* $(S, \cdot)$ refines the concept of a magma, by requiring $\cdot$ to be an *associative* binary operation.

2 [*Example*: $(\mathbb{R}, +)$ is a semigroup, since $(1.2 + 2.3) + \pi = 1.2 + (2.3 + \pi)$. — *end example*]

3 [*Example*: $(\mathbb{R}, -)$ is not a semigroup, since $(1.2 - 2.3) - \pi \neq 1.2 - (2.3 - \pi)$. — *end example*]

```
template<class BOp, class T, class U>
concept semigroup = magma<BOp, T, U>;
```

4 Let `bop` be an object of type `BOp`, `t` be an object of type `T`, and `u` be an object of type `U`.

5 `invoke(bop, t, invoke(bop, t, u))` is expression-equivalent to `invoke(bop, invoke(bop, t, u), u)`.

6 [*Note*: The difference between `magma` and `semigroup` is purely semantic. — *end note*]

3.2.4 Concept monoid [support.concepts.monoid]

1 A *monoid* $(S, \cdot)$ refines the concept of a semigroup by requiring $\cdot$ to have a two-sided identity element.

```
template<class BOp, class T, class U = T>
concept monoid = semigroup<BOp, T, U> && requires {
  typename two_sided_identity_t<BOp, remove_cvref_t<T>, remove_cvref_t<U>>;
};
```

3.2.5 Concept quasigroup [support.concepts.quasigroup]

1 A *quasigroup* $(S, \cdot)$ refines the concept of a magma by requiring $\cdot$ have an inverse operation.

[*Example*: $(\mathbb{Z}, +)$ is a quasigroup, since subtraction is the inverse operation of addition. — *end example*]

[*Example*: $(\mathbb{Z}, \mathrm{rem})$ is not a quasigroup, since there is no inverse operation for remainder. — *end example*]

```
template<class BOp, class T, class U>
concept quasigroup = magma<BOp, T, U> and requires {
   typename inverse_operation_t<BOp, T, U>;
   typename inverse_operation_t<inverse_operation_t<BOp, T, U>, T, U>;
```

```
    requires same_as<BOp, inverse_operation_t<inverse_operation_t<BOp, T, U>, T, U>>;
};
```

2 Let `t` be an object of type `T`, `u` be an object of type `U`, `bop` be an object of type `BOp`.

3 There exists an object `inv` of type `Inv`, where `Inv` models `magma<T, U>`, and the expressions

(3.1) — `invoke(inv, invoke(bop, t, u), t) == u`

(3.2) — `invoke(inv, invoke(bop, t, u), u) == t`

are both `true`. [*Note*: This implies that `Inv` also models `quasigroup<T, U>`, with respect to `inv`. — *end note*]

[Note to reviewers: The author wonders if having both `semigroup` and `quasigroup` will be confusing for some people. The names `associative_magma` (for `semigroup`) and `invertible_magma` (for `quasigroup`) have been considered as alternatives.]

3.2.6 Concept loop [support.concepts.loop]

1 A *loop* $(S, \cdot)$ refines the concept of a quasigroup by requiring $\cdot$ to have a two-sided identity element.

```
template<class BOp, class T, class U>
concept loop = quasigroup<BOp, T, U> && requires {
  typename two_sided_identity_t<BOp, remove_cvref_t<T>, remove_cvref_t<U>>;
};
```

2 `invoke(inv, t, invoke(bop, t, u))` is expression-equivalent to `invoke(inv, two_sided_identity(bop, t, u), t)`.

3 `invoke(inv, u, invoke(bop, t, u))` is expression-equivalent to `invoke(inv, two_sided_identity(bop, t, u), u)`.

[Note to reviewers: To avoid confusion between computer science loops and abstract algebra loops, the author is considering renaming `loop` to `abstract_loop` or `algebraic_loop`.

The solution should be considered for the notion of a module (for obvious reasons), ring (to avoid confusion with types such as `ring_buffer`), and to a lesser extent, group ('group' is a *very* abstract term). This proposal introduces neither a `module` concept, nor a `ring` concept (but it is plausible that a subsequent paper propose these).]

3.2.7 Concept group [support.concepts.group]

1 A *group* is a refinement of both a semigroup and a quasigroup[5].

```
template<class BOp, class T, class U>
concept group = semigroup<BOp, T, U> && quasigroup<BOp, T, U>;
```

[Note to reviewers: The author is open to simplifying the group heirarchy by following the design presented in EoP, where a group refines a monoid by requiring inverse operation. This would eliminate the confusion between 'semigroup' and 'quasigroup', and won't introduce any concept-like diamond problems.]

3.2.8 Concept abelian group [support.concepts.abelian.group]

1 A *abelian group* refines a group such that the operation is commutative[3].

```
template<class BOp, class T, class U>
concept abelian_group = group<BOp, T, U> && commutative_operation<BOp, T, U>;
```

[Note to reviewers: An alternative, and possibly more appropriate name for this concept is `commutative_group`.]

[Note to reviewers: `group` and `abelian_group` are the only concepts introduced in this proposal that aren't required by an algorithm, but given their 'simple' definitions, it would be remiss to omit them. The same cannot be said for the ring hierarchy, as there are more gaps between what is proposed in P1813 and the full definition of a ring.]

3.2.9 Concept weak-magmaring [support.concepts.weak.magmaring]

1 A *weak-magmaring* $(S, \cdot)$ is a generalisation of the notion of a near-semiring, where:

(1.1) — $(S, +)$ is a magma.

(1.2) — $(S, \cdot)$ is a magma.

(1.3) — $\cdot$ is distributive over $+$

(1.3.1) — $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

(1.3.2) — $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$

[*Note*: In the definition of a weak-magmaring, $+$ does not refer to canonical addition, and $\cdot$ does not refer to canonical multiplication. — *end note*]

```
template<class BOp1, class BOp2, class T, class U, class V>
concept weak_magmaring = magma<BOp2, U, V> && magma<BOp1, T, invoke_result_t<BOp2&, U, V>>;
```

2 Let `bop1` be an object of type `BOp1`, `bop2` be an object of type `BOp2`, `t` be an object of type `T`, `u` be an object of type `U`, and `v` be an object of type `V`.

3 `invoke(bop2, invoke(bop1, t, u), v)` is expression-equivalent to `invoke(bop1, invoke(bop2, t, v), invoke(bop2, u, v))`.

[Note to reviewers: This could be renamed as `distributive_operation` to avoid introducing novel mathematical terms that lack rigourous definitions. The author is not convinced that the concept definition needs to change for this renaming to be possible.]

3.2.10 Concept near-semiring [support.concepts.near.semiring]

1 A *near-semiring* $(S, +, \cdot)$ refines the notion of a weak-magmaring, by refining the substructures, and introducing the notion of a two-sided zero element.

(1.1) — $(S, +)$ is a monoid.

(1.2) — $(S, \cdot)$ is a semigroup.

(1.3) — $0 \cdot a = a \cdot 0 = 0$ for all a in S.

```
template<class BOp1, class BOp2, class T, class U, class V>
concept near_semiring = weak_magmaring<BOp1, BOp2, T, U, V> &&
  monoid<BOp1, T, invoke_result_t<BOp2&, U, V>> && semigroup<BOp2, U, V> && requires {
    typename two_sided_zero_t<BOp2, remove_cvref_t<U>, remove_cvref_t<V>>;
  };
```

3.2.11 Indirect callable requirements [support.concepts.indirect]

The following concepts are convenience concepts, similar to those already in the C++20 WP. With the exception of `indirect_commutative_operation`, all of the proposed concepts in this section require the algebraic structure model `writable` to some object. This makes the indirect algebraic structure concepts more in line with `sortable` and `permutable` than with `indirect_unary_invocable`, etc.

```
template<class BOp, class I1, class I2>
concept indirect_commutative_operation =
  readable<I1> &&
  readable<I2> &&
  commutative_operation<BOp&, iter_value_t<I1>&, iter_value_t<I2>&> &&
  commutative_operation<BOp&, iter_value_t<I1>&, iter_reference_t<I2>> &&
  commutative_operation<BOp&, iter_reference_t<I1>, iter_value_t<I2>&> &&
  commutative_operation<BOp&, iter_reference_t<I1>, iter_reference_t<I2>> &&
  commutative_operation<BOp&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

template<class BOp, class I1, class I2, class O>
concept indirect_magma =
  readable<I1> &&
  readable<I2> &&
  writable<O, indirect_result_t<BOp&, I1, I2>> &&
  magma<BOp&, iter_value_t<I1>&, iter_value_t<I2>&> &&
  magma<BOp&, iter_value_t<I1>&, iter_reference_t<I2>&> &&
  magma<BOp&, iter_reference_t<I1>, iter_value_t<I2>&> &&
  magma<BOp&, iter_reference_t<I1>, iter_reference_t<I2>> &&
  magma<BOp&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;
```

```
template<class BOp, class I1, class I2, class O>
concept indirect_semigroup = indirect_magma<BOp, I1, I2, O> &&
  semigroup<BOp&, iter_value_t<I1>&, iter_value_t<I2>&> &&
  semigroup<BOp&, iter_value_t<I1>&, iter_reference_t<I2>&> &&
  semigroup<BOp&, iter_reference_t<I1>, iter_value_t<I2>&> &&
  semigroup<BOp&, iter_reference_t<I1>, iter_reference_t<I2>> &&
  semigroup<BOp&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

template<class BOp, class I1, class I2, class O>
concept indirect_monoid = indirect_semigroup<BOp, I1, I2, O> &&
  monoid<BOp&, iter_value_t<I1>&, iter_value_t<I2>&> &&
  monoid<BOp&, iter_value_t<I1>&, iter_reference_t<I2>&> &&
  monoid<BOp&, iter_reference_t<I1>, iter_value_t<I2>&> &&
  monoid<BOp&, iter_reference_t<I1>, iter_reference_t<I2>> &&
  monoid<BOp&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

template<class BOp1, class BOp2, class I1, class I2, class I3, class O>
concept indirect_weak_magmaring =
  indirect_magma<BOp2, I2, I3, O> &&
  indirect_magma<BOp1, I1, indirect_result_t<BOp2&, I2, I3>*, O> &&
  weak_magmaring<BOp1&, BOp2&, iter_value_t<I1>&, iter_value_t<I2>&, iter_value_t<I3>&> &&
  weak_magmaring<BOp1&, BOp2&, iter_value_t<I1>&, iter_value_t<I2>&, iter_reference_t<I3>> &&
  weak_magmaring<BOp1&, BOp2&, iter_value_t<I1>&, iter_reference_t<I2>, iter_value_t<I3>&> &&
  weak_magmaring<BOp1&, BOp2&, iter_value_t<I1>&, iter_reference_t<I2>, iter_reference_t<I3>> &&
  weak_magmaring<BOp1&, BOp2&, iter_reference_t<I1>, iter_value_t<I2>&, iter_value_t<I3>&> &&
  weak_magmaring<BOp1&, BOp2&, iter_reference_t<I1>, iter_value_t<I2>&, iter_reference_t<I3>> &&
  weak_magmaring<BOp1&, BOp2&, iter_reference_t<I1>, iter_reference_t<I2>, iter_value_t<I3>&> &&
  weak_magmaring<BOp1&, BOp2&, iter_reference_t<I1>, iter_reference_t<I2>, iter_reference_t<I3>> &&
  weak_magmaring<BOp1&, BOp2&, iter_common_reference_t<I1>, iter_common_reference_t<I2>,
    iter_common_reference_t<I3>>;

template<class BOp1, class BOp2, class I1, class I2, class I3, class O>
concept indirect_near_semiring =
  indirect_weak_magmaring<BOp1, BOp2, I1, I2, I3, O> &&
  near_semiring<BOp1&, BOp2&, iter_value_t<I1>&, iter_value_t<I2>&, iter_value_t<I3>&> &&
  near_semiring<BOp1&, BOp2&, iter_value_t<I1>&, iter_value_t<I2>&, iter_reference_t<I3>> &&
  near_semiring<BOp1&, BOp2&, iter_value_t<I1>&, iter_reference_t<I2>, iter_value_t<I3>&> &&
  near_semiring<BOp1&, BOp2&, iter_value_t<I1>&, iter_reference_t<I2>, iter_reference_t<I3>> &&
  near_semiring<BOp1&, BOp2&, iter_reference_t<I1>, iter_value_t<I2>&, iter_value_t<I3>&> &&
  near_semiring<BOp1&, BOp2&, iter_reference_t<I1>, iter_value_t<I2>&, iter_reference_t<I3>> &&
  near_semiring<BOp1&, BOp2&, iter_reference_t<I1>, iter_reference_t<I2>, iter_value_t<I3>&> &&
  near_semiring<BOp1&, BOp2&, iter_reference_t<I1>, iter_reference_t<I2>, iter_reference_t<I3>> &&
  near_semiring<BOp1&, BOp2&, iter_common_reference_t<I1>, iter_common_reference_t<I2>,
    iter_common_reference_t<I3>>;
```

3.3 Arithmetic function objects [support.arithmetic.ops]

Just as P0896 redesigned the comparison function objects, P1813 seeks to redesign the numeric operation function objects. This will allow us to:

- Forget about — and hopefully — one day eliminate the arithmetic function objects in namespace `std`.
- Apply requirements to each operation to ensure that they're semantically sound (what does it 'mean' to evaluate `1 + vector{1, 2, 3}`?).

3.3.1 Plus [support.arithmetic.plus]

```
namespace std::ranges {
  template<class T, class U>
  concept summable-with = // exposition only
    default_initializable<remove_reference_t<T>> &&
    default_initializable<remove_reference_t<U>> &&
    common_reference_with<T, U> &&
    requires(T&& t, U&& u) {
      { std::forward<T>(t) + std::forward<T>(t) } -> common_with<T>;
      { std::forward<U>(u) + std::forward<U>(u) } -> common_with<U>;
```

```
      { std::forward<T>(t) + std::forward<U>(u) } -> common_with<T>;
      { std::forward<U>(u) + std::forward<T>(t) } -> common_with<U>;
      requires same_as<decltype(std::forward<T>(t) + std::forward<U>(u)),
                       decltype(std::forward<U>(u) + std::forward<T>(t))>;
    };
}
```

1 The expression `t + u` is expression-equivalent to `u + t`.

2 The expressions `t + T{} == t`, `u + T{} == common_type_t<T, U>{}`, and `t + U{} == common_type_t<T, U>{}` are all `true`.

[Note to reviewers: This is not the same as the dreaded `has_plus`; it's more of a `has_plus+`.]

```
  struct plus {
    template<class T, summable-with<T> U>
    constexpr decltype(auto) operator()(T&& t, U&& u) const {
      return std::forward<T>(t) + std::forward<U>(u);
    }

    using is_transparent = std::true_type;
  };

  template<class T, class U>
  requires magma<ranges::plus, T, U>
  struct left_identity<ranges::plus> {
    constexpr common_type_t<T, U> operator()() const { return T{}; }
  };

  template<class T, class U>
  requires magma<ranges::plus, T, U>
  struct right_identity<ranges::plus> {
    constexpr common_type_t<T, U> operator()() const { return U{}; }
  };

  template<>
  struct inverse_traits<ranges::plus> {
    using type = minus;
    constexpr type operator()() const noexcept { return type{}; }
  };
}
```

3.3.2 Negate [support.arithmetic.negate]

```
namespace std::ranges {
  template<class T>
  concept negatable = // exposition only
    summable-with<T, T> &&
    totally_ordered<T> &&
    requires(T&& t) {
      { -std::forward<T>(t) } -> common_with<T>;
    };
```

1 Let `t`, `t1`, and `t2` objects of type `T`.

2 `-(-t)` is expression-equivalent to `t`.

3 The expression `-t == t` is `true` if, and only if, `t == T{}` is also `true`.

4 The expression `-t < t` is `true` if, and only if, `T{} < t` is also `true`.

5 The expression `t + -t` is expression-equivalent to `T{}`.

6 If `t1 < t2` is `true` and `T{} < t2` is `true`, then

(6.1) — `t1 + -t2 < t1` is `true`,

(6.2) — `t1 + -t2 > -t2` is `true`,

(6.3) — `-t1 + t2 < t2` is `true`, and

(6.4) — `-t1 + t2 > t1` is `true`.

```
  struct negate {
    template<negatable T>
    constexpr decltype(auto) operator()(T&& t) const {
      return -std::forward<T>(t);
    }

    using is_transparent = std::true_type;
  };
}
```

[Note to reviewers: `negate` is not a binary operation, but it is plausible for there to be an `inverse_operation<negate>` specialisation where `operator()` returns an object of type `negate`.]

3.3.3 Minus [support.arithmetic.minus]

```
namespace std::ranges {
  template<class T, class U>
  concept differenceable-with = // exposition only
    summable-with<T, U> &&
    negatable<T> &&
    negatable<U> &&
    totally_ordered_with<T, U> &&
    requires(T&& t, U&& u) {
      { std::forward<T>(t) - std::forward<T>(t) } -> common_with<T>;
      { std::forward<U>(u) - std::forward<U>(u) } -> common_with<U>;
      { std::forward<T>(t) - std::forward<U>(u) } -> common_with<T>;
      { std::forward<U>(u) - std::forward<T>(t) } -> common_with<U>;
      requires same_as<decltype(std::forward<T>(t) - std::forward<U>(u)),
                       decltype(std::forward<U>(u) - std::forward<T>(t))>;
    };
```

1 Let `t1` and `t2` be objects of type `T`, and `u1` and `u2` be objects of type `U`, where `t1 != t2` and `u1 != u2`.

2 `t1 - t2` is equivalent to `t1 + -t2`, `u1 - u2` is equivalent to `u1 + -u2`, and `t - u` is equivalent to `t + -u`.

3 `t - t` is expression-equivalent to `T{}`, `u - u` is expression-equivalent to `U{}`, and if `t == u`, then `t - u` is expression-equivalent to `common_type_t<T, U>{}`.

4 `t - (-t)` is equivalent to `t + t`, `u - (-u)` is equivalent to `u + u`, and `t - (-u)` is equivalent to `t + u`.

5 `-t1 - t2` is equivalent to `-(t1 + t2)`, `-u1 - u2` is equivalent to `-(u1 + u2)`, and `-t - u` is expression-equivalent to `-(t + u)`.

6 `t + u - t` is expression-equivalent to `static_cast<common_type_t<T, U>(t)`, and `t + u - u` is expression-equivalent to `static_cast<common_type_t<T, U>(u)`.

[Note to reviewers: TODO: add semantics for subtraction and ordering.]

```
  struct minus {
    template<class T, differenceable-with<T> U>
    constexpr decltype(auto) operator()(T&& t, U&& u) const {
      return std::forward<T>(t) - std::forward<U>(u);
    }

    using is_transparent = std::true_type;
  };

  template<class T, class U>
  struct right_identity<ranges::minus, T, U> : private right_identity<ranges::plus, T, U> {
    using right_identity<ranges::plus, T, U>::operator();
  };
```

```
  template<>
  struct inverse_traits<ranges::minus> {
    using type = ranges::plus;
    constexpr type operator()() const noexcept { return type{}; }
  }
}
```

3.3.4 Times [support.arithmetic.times]

[Note to reviewers: The term `multiplies` is — in the author's opinion — not the best name, and so the author would like to take the opportunity of rename this function object so that one can more naturally describe the computation.

A potential alternative is `product`, this is the result of multiplication, not the operation itself (we'd need to rename `plus` to `sum`, etc., to facilitate that idea).]

```
namespace std::ranges {
  template<class T, class U>
  concept multiplicable-with = // exposition only
    summable-with<T, U> &&
    constructible_from<remove_cvref_t<T>, int> && // specifically T{0} and T{1}
    constructible_from<remove_cvref_t<U>, int> && // specifically U{0} and U{1}
    constructible_from<remove_cvref_t<common_type<T, U>>, int> &&
    common_reference_with<T, U> &&
    requires(T&& t, U&& u) {
      { std::forward<T>(t) * std::forward<T>(t) } -> common_with<T>;
      { std::forward<U>(u) * std::forward<U>(u) } -> common_with<U>;
      { std::forward<T>(t) * std::forward<U>(u) } -> common_with<T>;
      { std::forward<U>(u) * std::forward<T>(t) } -> common_with<U>;
      requires same_as<decltype(std::forward<T>(t) * std::forward<U>(u)),
                       decltype(std::forward<U>(u) * std::forward<T>(t))>;
    };
```

1 `T{0}` is equivalent to `T{}`, and `U{0}` is equivalent to `U{}`.

2 The expressions

- (2.1) `t * T{} == T{}`,
- (2.2) `u * U{} == U{}`,
- (2.3) `t * U{} == U{}`, and
- (2.4) `u * T{} == T{}`

are all `true`.

3 The expressions

- (3.1) `t * T{1} == t`,
- (3.2) `T{1} * t == t`,
- (3.3) `u * U{1} == u`,
- (3.4) `U{1} * u == u`,
- (3.5) `u * T{1} == u`,
- (3.6) `T{1} * u == u`,
- (3.7) `t * U{1} == t`, and
- (3.8) `U{1} * t == t`

are all `true`.

```
  struct times {
    template<class T, multiplicable-with<T> U>
    constexpr decltype(auto) operator()(T&& t, U&& u) const
    { return std::forward<T>(t) * std::forward<U>(u); }

    using is_transparent = std::true_type;
```

```
  };

  template<class T, class U>
  requires magma<times, T, U>
  struct left_identity<times> {
    constexpr common_type_t<T, U> operator()() const { return T{1}; }
  };

  template<class T, class U>
  requires magma<times, T, U>
  struct right_identity<times> {
    constexpr common_type_t<T, U> operator()() const { return U{1}; }
  };

  template<class T, class U>
  requires magma<times, T, U>
  struct left_zero<times> {
    constexpr common_type_t<T, U> operator()() const { return T{}; }
  };

  template<class T, class U>
  requires magma<times, T, U>
  struct right_zero<times> {
    constexpr common_type_t<T, U> operator()() const { return U{}; }
  };

  template<>
  struct inverse_traits<times> {
     using type = divided_by;
     constexpr type operator()() const noexcept { return type{}; }
  };
}
```

3.3.5 Divided by [support.arithmetic.divided_by]

[Note to reviewers: The term `divides` clashes with the predicate *divides*, which is used to indicate that a quotient is an integer. The author recommends renaming this operation to `divided_by`. A potential alternative is `quotient`, but this has the same issues as `product`.]

```
namespace std::ranges {
  template<class T, class U>
  concept divisible-with = // exposition only
    multiplicable-with<T, U> &&
    subtractible-with<T, U> &&
    requires(T&& t, U&& u) {
      { std::forward<T>(t) / std::forward<T>(t) } -> common_with<T>;
      { std::forward<U>(u) / std::forward<U>(u) } -> common_with<U>;
      { std::forward<T>(t) / std::forward<U>(u) } -> common_with<T>;
      { std::forward<U>(u) / std::forward<T>(t) } -> common_with<U>;
      requires same_as<decltype(std::forward<T>(t) / std::forward<U>(u)),
                       decltype(std::forward<U>(u) / std::forward<T>(t))>;;
    };
```

1 Let `t1` and `t2` be objects of type `T`, and `u1` and `u2` be objects of type `U`. It is undefined for `t2 == T0` or `u2 == U0` to be `true` in all of the paragraphs below.

2 The expressions

(2.1) — `(t1 / t2) * t2 == t1`,

(2.2) — `(t1 * t2) / t2 == t1`,

(2.3) — `(u1 / u2) * u2 == u1`,

(2.4) — `(u1 * u2) / u2 == u1`,

(2.5) — `(t1 / u2) * u2 == t1`,

(2.6) — `(t1 * u2) / u2 == t1`,

(2.7) — `(u1 / t2) * t2 == u1`, and

(2.8) — `(u1 * t2) / t2 == u1`

are all `true`.

3 The expressions `T{} / t2 == T{}`, `U{} / u2 == U{}`, `T{} / u2 == common_type_t<T, U>{}`, and `U{} / t2 == common_type_t<T, U>{}` are all `true`.

```
  struct divided_by {
    template<class T, divisible-with<T> U>
    constexpr decltype(auto) operator()(T&& t, U&& u) const
    { return std::forward<T>(t) / std::forward<U>(u); }
  };

  template<class T, class U>
  requires magma<divided_by, T, U>
  struct right_identity<divided_by> {
    constexpr common_type_t<T, U> operator()() const { return U{1}; }
  };

  template<>
  struct inverse_traits<divided_by> {
    using type = times;
    constexpr type operator()() const noexcept { return type{}; }
  };
}
```

3.3.6 Modulus [support.arithmetic.modulus]

[Note to reviewers: An alternative name to `modulus` is `remainder`, which fits well with `sum`, `difference`, `product`, and `quotient`.]

```
namespace std::ranges {
  template<class T, class Q>
  concept modulo-with = // exposition only
    divisible-with<T, Q> &&
    requires(T&& t, Q&& q) {
      { std::forward<T>(t) % std::forward<T>(t) } -> common_with<T>;
      { std::forward<Q>(q) % std::forward<Q>(q) } -> common_with<Q>;
      { std::forward<T>(t) % std::forward<Q>(q) } -> common_with<T>;
      { std::forward<Q>(q) % std::forward<T>(t) } -> common_with<Q>;
      requires same_as<decltype(std::forward<T>(t) % std::forward<Q>(q)),
                       decltype(std::forward<Q>(q) % std::forward<T>(t))>;
    };
```

1 Let `n` and `r` be objects of type `common_type_t<T, Q>`.

2 The expression `t == q * n + r` is `true` if and only if `t % q == r` is `true`.

```
  struct modulus {
    template<class T, modulo-with<T> U>
    constexpr decltype(auto) operator()(T&& t, U&& u) const
    { return std::forward<T>(t) % std::forward<U>(u); }
  };

  template<class T, class U>
  requires magma<modulus, T, U>
  struct left_zero<modulus> {
    constexpr common_type_t<T, U> operator()() const { return T{}; }
  };
}
```

4 (informative) Proofs [proof]

4.1 Adjacent difference is the inverse of partial sum [proof.adjacent.difference]

4.1.1 Defining adjacent difference [proof.adjacent.difference.defn]

Let $(S, +)$ model a loop for some set S, where $+$ denotes an arbitrary operation, and $-$ denotes its inverse. Let $\mathbf{in}$ be an ordered sequence of elements of the set S. There exists a function $d : ([S], \mathbb{Z}^+) \to S$, such that:

$$d(\mathbf{in}, k) = \begin{cases} \mathbf{in}_1 & \text{when } k = 1 \\ \mathbf{in}_k - \mathbf{in}_{k-1} & \text{when } k > 1 \end{cases}$$

There also exists an ordered sequence $\mathbf{o}$, such that

$$\mathbf{o}_n = d(\mathbf{in}, n).$$

We define $\mathbf{o}$ as the *adjacent difference* of $\mathbf{in}$.

4.1.2 Theorem [proof.adjacent.difference.theorem]

Suppose that $\mathbf{a}$ is an ordered sequence of elements of the set S, and that $\mathbf{s}$ is its partial sum, with respect to $+$. The ajdacent difference of $\mathbf{p}$ is equivalent to $\mathbf{a}$; that is, for all ordered sequences of length n, the adjacent difference of a partial sum of an ordered sequence yields identity.

4.1.3 Proof [proof.adjacent.difference.proof]

Case $n = 0$: Since $\mathbf{s}_0$ and $\mathbf{o}_0$ are not defined, the proof is trivial.
Case $n = 1$: $\mathbf{s}_1 = \mathbf{a}_1$ and $\mathbf{o}_1 = d(\mathbf{s}, 1) = \mathbf{a}_1$, so the proof is trivial.
Case $n > 1$:

$$\begin{aligned} \mathbf{s}_n &= \mathbf{a}_1 + \mathbf{a}_2 + ... + \mathbf{a}_n && (4.1) \\ \mathbf{o} &= [d(\mathbf{s}_1), d(\mathbf{s}_2), ..., d(\mathbf{s}_n)] && (4.2) \\ &= [\mathbf{a}_1, (\mathbf{a}_1 + \mathbf{a}_2) - \mathbf{a}_1, ..., (\mathbf{a}_1 + \mathbf{a}_2 + ... + \mathbf{a}_n) - (\mathbf{a}_1 + \mathbf{a}_2 + ... + \mathbf{a}_{n-1})] && (4.3) \\ &= [\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n] && (4.4) \\ &= \mathbf{a}. && (4.5) \end{aligned}$$

Given that the theorem is true for $n = 0$, $n = 1$, and $n > 1$, the theorem is true for all natural numbers n.

4.2 Proof for uniqueness of a two-sided identity element [proof.identity]

Let $(S, \cdot)$ be a magma. If there exist elements l, r in S, where for all other elements x in S, $l \cdot x = x$ and $x \cdot r = x$, then $l = r$.

4.2.1 Proof (by contradiction) [proof.identity.proof]

Let us first suppose that l and r are distinct. Then, $l \cdot r = r$, since l is a left-identity. But $l \cdot r = l$, since r is a right-identity. This is a contradiction.

Therefore, $l = r$, and the proof is complete.

4.3 Proof for uniqueness of a two-sided zero element [proof.zero]

Let $(S, \cdot)$ be a magma. If there exist elements l, r in S, where for all other elements x in S, $l \cdot x = l$ and $x \cdot r = r$, then $l = r$.

4.3.1 Proof (by contradiction) [proof.identity.proof]

Let us first suppose that l and r are distinct. Then, $l \cdot r = l$, since l is a left-zero. But $l \cdot r = r$, since r is a right-zero. This is a contradiction.

Therefore, $l = r$, and the proof is complete.

Bibliography

[1] Near-semiring. https://en.wikipedia.org/wiki/Near-semiring, Dec 2017.

[2] Magma (algebra). https://en.wikipedia.org/wiki/Magma_(algebra), Dec 2018.

[3] Abelian group. https://en.wikipedia.org/wiki/Abelian_group, Jul 2019.

[4] Absorbing element. https://en.wikipedia.org/wiki/Absorbing_element, Jun 2019.

[5] Group (mathematics). https://en.wikipedia.org/wiki/Group_(mathematics), Jul 2019.

[6] Identity element. https://en.wikipedia.org/wiki/Identity_element, Jun 2019.

[7] Monoid. https://en.wikipedia.org/wiki/Monoid, Jul 2019.

[8] Quasigroup. https://en.wikipedia.org/wiki/Quasigroup, Jul 2019.

[9] Semigroup. https://en.wikipedia.org/wiki/Semigroup, Jun 2019.

[10] Christopher Di Bella. What's more general than a near-semiring? Mathematics Stack Exchange.

[11] Casey Carter and Christopher Di Bella. Rangify the uninitialised memory algorithms! https://wg21.link/p1033, 2018.

[12] Casey Carter and Eric Niebler. Standard library concepts. https://wg21.link/p0896, 2018.

[13] Eric Niebler Casey Carter and Christopher Di Bella. The one ranges proposal. https://wg21.link/p0896, 2018.

[14] Andrew Sutton Eric Niebler, Sean Parent. Ranges for the standard library, revision 1. https://wg21.link/n4128, 2014.

[15] Eric Niebler and Casey Carter. Working draft, c++ extensions for ranges. https://wg21.link/n4685, 2017.

[16] Alexander Stepanov and Paul McJones. Elements of programming. http://elementsofprogramming.com/, 2009.

[17] Bjarne Stroustrup and Andrew Sutton. A concept design for the stl. https://wg21.link/n3351, 2012.

[18] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for c++. http://www.stroustrup.com/sle2011-concepts.pdf, 2011.

[19] Eric W. Weisstein. Matrix multiplication. http://mathworld.wolfram.com/MatrixMultiplication.html.

[20] Eric W. Weisstein. Partial sum. http://mathworld.wolfram.com/PartialSum.html.