Ref:	*unspecified*
Date:	2025-01-01
Revises:	*unspecified*
Reply at:	Discussions, issues

mp-units Library Reference Documentations

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

Contents

Foreword

[This page is intentionally left blank.]

Introduction

Clauses and subclauses in this document are annotated with a so-called stable name, presented in square brackets next to the (sub)clause heading (such as "[quantities]" for Clause 5). Stable names aid in the discussion and evolution of this document by serving as stable references to subclauses across editions that are unaffected by changes of subclause numbering.

Aspects of the language syntax of C++ are distinguished typographically by the use of *italic, sans-serif* type or `constant width` type to avoid ambiguities.

1 Scope [scope]

1 This document describes the contents of the *mp-units library*.

2 References [refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — IEC 60050-102:2007/AMD3:2021, *Amendment 3 — International Electrotechnical Vocabulary (IEV) — Part 102: Mathematics — General concepts and linear algebra*

(1.2) — IEC 60050-112:2010/AMD2:2020, *Amendment 2 — International Electrotechnical Vocabulary (IEV) — Part 112: Quantities and units*

(1.3) — ISO 80000 (all parts), *Quantities and units*

(1.4) — The C++ Standards Committee. N4971: *Working Draft, Standard for Programming Language C++*. Edited by Thomas Köppe. Available from: `https://wg21.link/N4971{}`

(1.5) — The C++ Standards Committee. P3094R5: ***`std::basic_fixed_string`***. Edited by Mateusz Pusz. Available from: `https://wg21.link/P3094R5`

(1.6) — The C++ Standards Committee. SD-8: *Standard Library Compatibility*. Edited by Bryce Lelbach. Available from: `https://wg21.link/SD8`

3 Terms and definitions [defs]

1 For the purposes of this document, the terms and definitions given in IEC 60050-102:2007/AMD3:2021, IEC 60050-112:2010/AMD2:2020, ISO 80000-2:2019, and N4971, and the following apply.

2 ISO and IEC maintain terminology databases for use in standardization at the following addresses:

(2.1) — ISO Online browsing platform: available at `https://www.iso.org/obp`

(2.2) — IEC Electropedia: available at `http://www.electropedia.org`

4 Specification [spec]

4.1 External [spec.ext]

1 The specification of the mp-units library subsumes N4971, [description], N4971, [requirements], N4971, [concepts.equality], and SD-8, all assumingly amended for the context of this library.

[*Note 1*: This means that, non exhaustively,

(1.1) — `::mp_units2` is a reserved namespace, and

(1.2) — `std::vector<mp_units::`*`type`*`>` is a program-defined specialization and a library-defined specialization from the point of view of the C++ standard library and the mp-units library, respectively.

— *end note*]

2 The mp-units library is not part of the C++ implementation.

4.2 Categories [spec.cats]

1 Detailed specifications for each of the components in the library are in Clause 5–Clause 5, as shown in Table 1.

Table 1 — Library categories [tab:lib.cats]

Clause	**Category**
Clause 5	Quantities library

2 The quantities library (Clause 5) describes components for dealing with quantities.

4.3 Modules [spec.mods]

1 The mp-units library provides the *mp-units modules*, shown in Table 2.

Table 2 — mp-units modules [tab:modules]

`mp_units`	`mp_units.core`	`mp_units.systems`

4.4 Library-wide requirements [spec.reqs]

4.4.1 Reserved names [spec.res.names]

1 The mp-units library reserves macro names that start with `MP_UNITS`*digit-sequence*$_{opt}$`_`.

5 Quantities and units library [quantities]

5.1 Summary [quantities.summary]

1 This Clause describes components for dealing with quantities and units, as summarized in Table 3.

Table 3 — Quantities and units library summary [tab:quantities.summary]

	Subclause	Module
5.3	Utilities	`mp_units.core`
5.4	Reference	
5.5	Representation	
5.6	Quantity	
5.7	Quantity point	
5.8	Systems	`mp_units.systems`
5.9	`std::chrono` interoperability	

[Editor's note: Following the SG16 recommendation at https://lists.isocpp.org/sg16/2024/10/4490.php, the *universal-character-names* should be replaced by their UTF-8 code points.]

5.2 mp-units module synopses [mp.units.syns]

5.2.1 Module `mp_units` synopsis [mp.units.syn]

```
export module mp_units;

export import mp_units.core;
export import mp_units.systems;
```

5.2.2 Module `mp_units.core` synopsis [mp.units.core.syn]

```
// mostly freestanding
export module mp_units.core;

import std;

export namespace mp_units {

// 5.3, utilities

// 5.3.3, symbol text

enum class character_set : std::int8_t { utf8, portable, default_character_set = utf8 };

template<std::size_t N, std::size_t M>
class symbol_text;

// 5.3.4, symbolic expressions

// 5.3.4.3, types

template<typename T, typename... Ts>
struct per;

template<typename F, int Num, int... Den>
  requires see below
struct power;

// 5.4, reference
// 5.4.2, dimension

// 5.4.2.2, concepts

template<typename T>
concept Dimension = see below;

template<typename T, auto D>
concept DimensionOf = see below;

// 5.4.2.3, types

template<symbol_text Symbol>
struct base_dimension;

template<SymbolicConstant... Expr>
struct derived_dimension;

struct dimension_one;
inline constexpr dimension_one dimension_one{};

// 5.4.2.4, operations

consteval Dimension auto inverse(Dimension auto d);

template<std::intmax_t Num, std::intmax_t Den = 1, Dimension D>
  requires(Den != 0)
consteval Dimension auto pow(D d);
consteval Dimension auto sqrt(Dimension auto d);
consteval Dimension auto cbrt(Dimension auto d);

// 5.4.2.5, symbol formatting

struct dimension_symbol_formatting {
  character_set char_set = character_set::default_character_set;
};

template<typename CharT = char, std::output_iterator<CharT> Out, Dimension D>
constexpr Out dimension_symbol_to(Out out, D d, const dimension_symbol_formatting& fmt = {});

template<dimension_symbol_formatting fmt = {}, typename CharT = char, Dimension D>
consteval std::string_view dimension_symbol(D);

// 5.4.3, quantity specification

// 5.4.3.2, concepts

template<typename T>
concept QuantitySpec = see below;

template<typename T, auto QS>
concept QuantitySpecOf = see below;

// 5.4.3.3, types

// 5.4.3.3.1, named

struct is_kind;
inline constexpr is_kind is_kind{};

template<auto...>
struct quantity_spec;   // not defined
```

```
template<BaseDimension auto Dim, QSProperty auto... Args>
struct quantity_spec<Dim, Args...>;

template<DerivedQuantitySpec auto Eq, QSProperty auto... Args>
struct quantity_spec<Eq, Args...>;

template<NamedQuantitySpec auto QS, QSProperty auto... Args>
struct quantity_spec<QS, Args...>;

template<NamedQuantitySpec auto QS, DerivedQuantitySpec auto Eq, QSProperty auto... Args>
struct quantity_spec<QS, Eq, Args...>;

// 5.4.3.3.2, derived

template<SymbolicConstant... Expr>
struct derived_quantity_spec;

// 5.4.3.3.3, base quantity of dimension one

struct dimensionless;
inline constexpr dimensionless dimensionless{};

// 5.4.3.3.4, kind of

template<QuantitySpec Q>
  requires see below
struct kind_of_;
template<QuantitySpec auto Q>
  requires requires { typename kind_of_<decltype(Q)>; }
inline constexpr kind_of_<decltype(Q)> kind_of{};

// 5.4.3.5, operations

consteval QuantitySpec auto inverse(QuantitySpec auto q);

template<std::intmax_t Num, std::intmax_t Den = 1, QuantitySpec Q>
  requires(Den != 0)
consteval QuantitySpec auto pow(Q q);
consteval QuantitySpec auto sqrt(QuantitySpec auto q);
consteval QuantitySpec auto cbrt(QuantitySpec auto q);

// 5.4.3.6, hierarchy algorithms

// 5.4.3.6.1, conversion

consteval bool implicitly_convertible(QuantitySpec auto from, QuantitySpec auto to);
consteval bool explicitly_convertible(QuantitySpec auto from, QuantitySpec auto to);
consteval bool castable(QuantitySpec auto from, QuantitySpec auto to);
consteval bool interconvertible(QuantitySpec auto qs1, QuantitySpec auto qs2);

// 5.4.3.6.2, get_kind

template<QuantitySpec Q>
consteval see below get_kind(Q);

// 5.4.3.6.3, get_common_quantity_spec

consteval QuantitySpec auto get_common_quantity_spec(QuantitySpec auto... qs)
  requires see below;

// 5.4.4, unit

// 5.4.4.2, magnitude
// 5.4.4.2.2, concepts

template<typename T>
concept MagConstant = see below;

template<typename T>
concept UnitMagnitude = see below;

// 5.4.4.2.3, types

template<symbol_text Symbol, long double Value>
  requires(Value > 0)
struct mag_constant;

// 5.4.4.2.4, operations

template<MagArg auto V>
constexpr UnitMagnitude auto mag = see below;

template<std::intmax_t N, std::intmax_t D>
  requires(N > 0)
constexpr UnitMagnitude auto mag_ratio = see below;

template<MagArg auto Base, int Num, int Den = 1>
constexpr UnitMagnitude auto mag_power = see below;

// constants

inline constexpr struct pi final :
    mag_constant<{u8"\u03C0" /* U+03C0 GREEK SMALL LETTER PI */, "pi"},
                 std::numbers::pi_v<long double>> {
} pi;

inline constexpr auto \u03C0 /* U+03C0 GREEK SMALL LETTER PI */ = pi;

// 5.4.4.3, traits

template<Unit auto U>
constexpr bool space_before_unit_symbol = true;

template<>
inline constexpr bool space_before_unit_symbol<one> = false;

// 5.4.4.4, concepts

template<typename T>
concept Unit = see below;

template<typename T>
concept PrefixableUnit = see below;

template<typename T>
concept AssociatedUnit = see below;

template<typename U, auto QS>
concept UnitOf = see below;

// 5.4.4.5, types

// 5.4.4.5.2, scaled

template<UnitMagnitude auto M, Unit U>
  requires see below
struct scaled_unit;
```

```
// 5.4.4.5.3, named

template<symbol_text Symbol, auto...>
struct named_unit;  // not defined

template<symbol_text Symbol, QuantityKindSpec auto QS>
  requires see below
struct named_unit<Symbol, QS>;

template<symbol_text Symbol, QuantityKindSpec auto QS, PointOrigin auto PO>
  requires see below
struct named_unit<Symbol, QS, PO>;

template<symbol_text Symbol>
  requires see below
struct named_unit<Symbol>;

template<symbol_text Symbol, Unit auto U>
  requires see below
struct named_unit<Symbol, U>;

template<symbol_text Symbol, Unit auto U, PointOrigin auto PO>
  requires see below
struct named_unit<Symbol, U, PO>;

template<symbol_text Symbol, AssociatedUnit auto U, QuantityKindSpec auto QS>
  requires see below
struct named_unit<Symbol, U, QS>;

template<symbol_text Symbol, AssociatedUnit auto U, QuantityKindSpec auto QS,
         PointOrigin auto PO>
  requires see below
struct named_unit<Symbol, U, QS, PO>;

// 5.4.4.5.4, prefixed

template<symbol_text Symbol, UnitMagnitude auto M, PrefixableUnit auto U>
  requires see below
struct prefixed_unit;

// 5.4.4.5.5, common

template<Unit U1, Unit U2, Unit... Rest>
struct common_unit;

// 5.4.4.5.6, derived

template<SymbolicConstant... Expr>
struct derived_unit;

// 5.4.4.5.7, one

struct one;
inline constexpr one one{};

// named derived units of a quantity of dimension one

inline constexpr struct percent final : named_unit<"%", mag_ratio<1, 100> * one> {
} percent;

inline constexpr struct per_mille final :
    named_unit<symbol_text{u8"\u2030" /* U+2030 PER MILLE SIGN */, "%o"},
               mag_ratio<1, 1000> * one> {
} per_mille;
```

```
inline constexpr struct parts_per_million final :
    named_unit<"ppm", mag_ratio<1, 1'000'000> * one> {
} parts_per_million;

inline constexpr auto ppm = parts_per_million;

// 5.4.4.6, operations

consteval Unit auto inverse(Unit auto u);

template<std::intmax_t Num, std::intmax_t Den = 1, Unit U>
  requires see below
consteval Unit auto pow(U u);
consteval Unit auto sqrt(Unit auto u);
consteval Unit auto cbrt(Unit auto u);
consteval Unit auto square(Unit auto u);
consteval Unit auto cubic(Unit auto u);

// 5.4.4.7, comparison

template<Unit From, Unit To>
consteval bool convertible(From from, To to);

// 5.4.4.8, observers

consteval QuantitySpec auto get_quantity_spec(AssociatedUnit auto u);
consteval Unit auto get_unit(AssociatedUnit auto u);

consteval Unit auto get_common_unit(Unit auto... us)
  requires see below;

// 5.4.4.10, symbol formatting

enum class unit_symbol_solidus : std::int8_t {
  one_denominator,
  always,
  never,
  default_denominator = one_denominator
};

enum class unit_symbol_separator : std::int8_t {
  space,
  half_high_dot,
  default_separator = space
};

struct unit_symbol_formatting {
  character_set char_set = character_set::default_character_set;
  unit_symbol_solidus solidus = unit_symbol_solidus::default_denominator;
  unit_symbol_separator separator = unit_symbol_separator::default_separator;
};

template<typename CharT = char, std::output_iterator<CharT> Out, Unit U>
constexpr Out unit_symbol_to(Out out, U u, const unit_symbol_formatting& fmt = {});

template<unit_symbol_formatting fmt = {}, typename CharT = char, Unit U>
consteval std::string_view unit_symbol(U);

// 5.4.5, concepts

template<typename T>
concept Reference = see below;
```

```
template<typename T, auto QS>
concept ReferenceOf = see below;

// 5.4.6, class template reference

template<QuantitySpec Q, Unit U>
struct reference;

// 5.4.7, operations

template<typename FwdRep, Reference R,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  requires(!OffsetUnit<decltype(get_unit(R{}))>)
constexpr quantity<R{}, Rep> operator*(FwdRep&& lhs, R r);

template<typename FwdRep, Reference R,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  requires(!OffsetUnit<decltype(get_unit(R{}))>)
constexpr Quantity auto operator/(FwdRep&& lhs, R);

template<typename FwdQ, Reference R, Quantity Q = std::remove_cvref_t<FwdQ>>
constexpr Quantity auto operator*(FwdQ&& q, R);

template<typename FwdQ, Reference R, Quantity Q = std::remove_cvref_t<FwdQ>>
constexpr Quantity auto operator/(FwdQ&& q, R);

template<Reference R, typename Rep>
  requires RepresentationOf<std::remove_cvref_t<Rep>, get_quantity_spec(R{})>
constexpr auto operator*(R, Rep&&) = delete;

template<Reference R, typename Rep>
  requires RepresentationOf<std::remove_cvref_t<Rep>, get_quantity_spec(R{})>
constexpr auto operator/(R, Rep&&) = delete;

template<Reference R, typename Q>
  requires Quantity<std::remove_cvref_t<Q>>
constexpr auto operator*(R, Q&&) = delete;

template<Reference R, typename Q>
  requires Quantity<std::remove_cvref_t<Q>>
constexpr auto operator/(R, Q&&) = delete;

// 5.4.9, observers

template<typename Q, typename U>
consteval QuantitySpec auto get_quantity_spec(reference<Q, U>);

template<typename Q, typename U>
consteval Unit auto get_unit(reference<Q, U>);

consteval AssociatedUnit auto get_common_reference(AssociatedUnit auto u1,
                                                   AssociatedUnit auto u2,
                                                   AssociatedUnit auto... rest)
  requires see below;

template<Reference R1, Reference R2, Reference... Rest>
consteval Reference auto get_common_reference(R1 r1, R2 r2, Rest... rest)
  requires see below;

// 5.4.11, math

template<Representation Rep, Reference R>
  requires requires { std::numeric_limits<Rep>::epsilon(); }
constexpr quantity<R{}, Rep> epsilon(R r) noexcept;                                     // hosted
```

```
// 5.5, representation

enum class quantity_character { scalar, complex, vector, tensor };

// 5.5.2, traits

// 5.5.2.1, floating-point

template<typename Rep>
constexpr bool treat_as_floating_point = see below;

// 5.5.2.2, quantity character

template<typename T>
constexpr bool disable_scalar = false;
template<>
inline constexpr bool disable_scalar<bool> = true;
template<typename T>
constexpr bool disable_scalar<std::complex<T>> = true;

template<typename T>
constexpr bool disable_complex = false;

template<typename T>
constexpr bool disable_vector = false;

// 5.5.2.3, values

template<typename Rep>
struct representation_values;

// 5.5.3, customization point objects

inline namespace unspecified {

inline constexpr unspecified real = unspecified;
inline constexpr unspecified imag = unspecified;
inline constexpr unspecified modulus = unspecified;

inline constexpr unspecified magnitude = unspecified;

}

// 5.5.4, concepts

template<typename T>
concept Representation = see below;

template<typename T, quantity_character Ch>
concept RepresentationOf = see below;

// 5.6, quantity

// 5.6.2, interoperability

template<typename T>
struct quantity_like_traits;  // not defined

template<typename T>
concept QuantityLike = see below;

// 5.6.3, class template quantity
```

```
template<typename T>
concept Quantity = see below;

template<typename Q, auto QS>
concept QuantityOf = see below;

template<Reference auto R, RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity;

// 5.6.14, construction helper delta

template<Reference R>
struct delta_;

template<Reference auto R>
constexpr delta_<decltype(R)> delta{};

// 5.6.15, non-member conversions

template<Unit auto ToU, see below>
  requires see below
constexpr Quantity auto value_cast(see below q);

template<Representation ToRep, see below>
  requires see below
constexpr quantity<see below, ToRep> value_cast(see below q);

template<Unit auto ToU, Representation ToRep, see below>
  requires see below
constexpr Quantity auto value_cast(see below q);
template<Representation ToRep, Unit auto ToU, see below>
  requires see below
constexpr Quantity auto value_cast(see below q);

template<Quantity ToQ, see below>
  requires see below
constexpr Quantity auto value_cast(see below q);

template<QuantitySpec auto ToQS, see below>
  requires see below
constexpr Quantity auto quantity_cast(see below q);

// 5.6.16, math

template<auto R, typename Rep>
  requires see below
constexpr quantity<R, Rep> abs(const quantity<R, Rep>& q) noexcept;

template<std::intmax_t Num, std::intmax_t Den = 1, auto R, typename Rep>
  requires see below
constexpr quantity<pow<Num, Den>(R), Rep> pow(const quantity<R, Rep>& q) noexcept;      // hosted

template<auto R, typename Rep>
  requires see below
constexpr quantity<sqrt(R), Rep> sqrt(const quantity<R, Rep>& q) noexcept;              // hosted

template<auto R, typename Rep>
  requires see below
constexpr quantity<cbrt(R), Rep> cbrt(const quantity<R, Rep>& q) noexcept;              // hosted

template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires see below
constexpr quantity<R, Rep> exp(const quantity<R, Rep>& q);                              // hosted
```

```
template<auto R, typename Rep>
  requires see below
constexpr bool isfinite(const quantity<R, Rep>& a) noexcept;                            // hosted

template<auto R, typename Rep>
  requires see below
constexpr bool isinf(const quantity<R, Rep>& a) noexcept;                               // hosted

template<auto R, typename Rep>
  requires see below
constexpr bool isnan(const quantity<R, Rep>& a) noexcept;                               // hosted

template<auto R, auto S, auto T, typename Rep1, typename Rep2, typename Rep3>
  requires see below
constexpr see below fma(const quantity<R, Rep1>& a, const quantity<S, Rep2>& x,         // hosted
                        const quantity<T, Rep3>& b) noexcept;

template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires see below
constexpr QuantityOf<get_quantity_spec(R1)> auto fmod(const quantity<R1, Rep1>& x,      // hosted
                                                      const quantity<R2, Rep2>& y) noexcept;

template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires see below
constexpr QuantityOf<get_quantity_spec(R1)> auto remainder(const quantity<R1, Rep1>& x, // hosted
                                                           const quantity<R2, Rep2>& y) noexcept;

template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> floor(                             // hosted
  const quantity<R, Rep>& q) noexcept
  requires see below;

template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> ceil(                              // hosted
  const quantity<R, Rep>& q) noexcept
  requires see below;

template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> round(                             // hosted
  const quantity<R, Rep>& q) noexcept
  requires see below;

template<Unit auto To, auto R, typename Rep>
constexpr QuantityOf<dimensionless / get_quantity_spec(R)> auto inverse(                // hosted
  const quantity<R, Rep>& q)
  requires see below;

template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires see below
constexpr QuantityOf<get_quantity_spec(get_common_reference(R1, R2))> auto hypot(       // hosted
  const quantity<R1, Rep1>& x, const quantity<R2, Rep2>& y) noexcept;

template<auto R1, typename Rep1, auto R2, typename Rep2, auto R3, typename Rep3>
  requires see below
constexpr QuantityOf<get_quantity_spec(get_common_reference(R1, R2, R3))> auto hypot(   // hosted
  const quantity<R1, Rep1>& x, const quantity<R2, Rep2>& y,
  const quantity<R3, Rep3>& z) noexcept;

}

// 5.6.17, std::common_type specializations
```

```
template<mp_units::Quantity Q1, mp_units::Quantity Q2>
  requires see below
struct std::common_type<Q1, Q2>;

template<mp_units::Quantity Q, mp_units::Representation Value>
  requires see below
struct std::common_type<Q, Value>;

template<mp_units::Quantity Q, mp_units::Representation Value>
  requires requires { typename std::common_type<Q, Value>; }
struct std::common_type<Value, Q> : std::common_type<Q, Value> {};

namespace mp_units {

// 5.6.18, random

// 5.6.18.2.1, class template uniform_int_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct uniform_int_distribution;                                                        // hosted

// 5.6.18.2.2, class template uniform_real_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct uniform_real_distribution;                                                       // hosted

// 5.6.18.3.1, class template binomial_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct binomial_distribution;                                                           // hosted

// 5.6.18.3.2, class template negative_binomial_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct negative_binomial_distribution;                                                  // hosted

// 5.6.18.3.3, class template geometric_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct geometric_distribution;                                                          // hosted

// 5.6.18.4.1, class template poisson_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct poisson_distribution;                                                            // hosted

// 5.6.18.4.2, class template exponential_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct exponential_distribution;                                                        // hosted

// 5.6.18.4.3, class template gamma_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct gamma_distribution;                                                              // hosted

// 5.6.18.4.4, class template weibull_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct weibull_distribution;                                                            // hosted
```

```
// 5.6.18.4.5, class template extreme_value_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct extreme_value_distribution;                                              // hosted

// 5.6.18.5.1, class template normal_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct normal_distribution;                                                     // hosted

// 5.6.18.5.2, class template lognormal_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct lognormal_distribution;                                                  // hosted

// 5.6.18.5.3, class template chi_squared_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct chi_squared_distribution;                                                // hosted

// 5.6.18.5.4, class template cauchy_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct cauchy_distribution;                                                     // hosted

// 5.6.18.5.5, class template fisher_f_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct fisher_f_distribution;                                                   // hosted

// 5.6.18.5.6, class template student_t_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct student_t_distribution;                                                  // hosted

// 5.6.18.6.1, class template discrete_distribution
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct discrete_distribution;                                                   // hosted

// 5.6.18.6.3, class template piecewise_constant_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
class piecewise_constant_distribution;                                          // hosted

// 5.6.18.6.4, class template piecewise_linear_distribution
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
class piecewise_linear_distribution;                                            // hosted

// 5.7, quantity point

// 5.7.2, point origin

// 5.7.2.2, concepts

template<typename T>
concept PointOrigin = see below;

template<typename T, auto QS>
concept PointOriginFor = see below;

// 5.7.2.3, types
// 5.7.2.3.1, absolute

template<QuantitySpec auto QS>
struct absolute_point_origin;

// 5.7.2.3.2, relative

template<QuantityPoint auto QP>
struct relative_point_origin;

// 5.7.2.3.3, zeroth

template<QuantitySpec auto QS>
struct zeroth_point_origin_;

template<QuantitySpec auto QS>
constexpr zeroth_point_origin_<QS> zeroth_point_origin{};

// 5.7.2.5.2, default

template<Reference R>
consteval PointOriginFor<get_quantity_spec(R{})> auto default_point_origin(R);

// 5.7.3, interoperability

template<typename T>
struct quantity_point_like_traits;   // not defined

template<typename T>
concept QuantityPointLike = see below;

// 5.7.4, class template quantity_point

template<typename T>
concept QuantityPoint = see below;

template<typename QP, auto V>
concept QuantityPointOf = see below;

template<Reference auto R,
         PointOriginFor<get_quantity_spec(R)> auto PO = default_point_origin(R),
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity_point;

// 5.7.14, construction helper point

template<Reference R>
struct point_;

template<Reference auto R>
constexpr point_<decltype(R)> point{};

// 5.7.15, non-member conversions

template<Unit auto ToU, see below>
  requires see below
constexpr QuantityPoint auto value_cast(see below qp);

template<Representation ToRep, see below>
  requires see below
constexpr quantity_point<see below, see below, ToRep> value_cast(see below qp);
```

```
template<Unit auto ToU, Representation ToRep, see below>
  requires see below
constexpr QuantityPoint auto value_cast(see below qp);
template<Representation ToRep, Unit auto ToU, see below>
  requires see below
constexpr QuantityPoint auto value_cast(see below qp);

template<Quantity ToQ, see below>
  requires see below
constexpr QuantityPoint auto value_cast(see below qp);

template<QuantityPoint ToQP, see below>
  requires see below
constexpr QuantityPoint auto value_cast(see below qp);

template<QuantitySpec auto ToQS, see below>
  requires see below
constexpr QuantityPoint auto quantity_cast(see below qp);

// 5.7.16, math

template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isfinite(q); }
constexpr bool isfinite(const quantity_point<R, PO, Rep>& a) noexcept;                 // hosted

template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isinf(q); }
constexpr bool isinf(const quantity_point<R, PO, Rep>& a) noexcept;                    // hosted

template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isnan(q); }
constexpr bool isnan(const quantity_point<R, PO, Rep>& a) noexcept;                    // hosted

template<auto R, auto S, auto T, auto Origin, typename Rep1, typename Rep2, typename Rep3>
  requires see below
constexpr see below fma(const quantity<R, Rep1>& a, const quantity<S, Rep2>& x,        // hosted
                        const quantity_point<T, Origin, Rep3>& b) noexcept;

}
```

5.2.3 Module mp_units.systems synopsis [mp.units.systems.syn]

```
export module mp_units.systems;

export import mp_units.core;
import std;

export namespace mp_units {

// 5.9, std::chrono interoperability

template<typename Rep, typename Period>
struct quantity_like_traits<std::chrono::duration<Rep, Period>>;

template<typename Clock>
struct chrono_point_origin_;
template<typename Clock>
constexpr chrono_point_origin_<Clock> chrono_point_origin{};

template<typename Clock, typename Rep, typename Period>
struct quantity_point_like_traits<
  std::chrono::time_point<Clock, std::chrono::duration<Rep, Period>>>;

}
```

5.3 Utilities [qty.utils]

5.3.1 Non-types [qty.utils.non.types]

```
template<typename T, template<see below> typename U>
consteval bool is-specialization-of();               // exposition only
template<typename T, template<see below> typename U>
consteval bool is-derived-from-specialization-of();  // exposition only
```

1 *Returns*:

(1.1) — For the first signature, `true` of `T` is a specialization of `U`, and `false` otherwise.

(1.2) — For the second signature, `true` if `T` has exactly one public base class that is a specialization of `U` and has no other base class that is a specialization of `U`, and `false` otherwise.

2 *Remarks*: An implementation provides enough overloads for all arguments to `U`.

5.3.2 Ratio [qty.ratio]

```
namespace mp_units {

struct ratio {  // exposition only
  std::intmax_t num;
  std::intmax_t den;

  consteval ratio(std::intmax_t n, std::intmax_t d = 1);

  friend consteval bool operator==(ratio, ratio) = default;
  friend consteval auto operator<=>(ratio lhs, ratio rhs) { return (lhs - rhs).num <=> 0; }

  friend consteval ratio operator-(ratio r) { return {-r.num, r.den}; }

  friend consteval ratio operator+(ratio lhs, ratio rhs)
  {
    return {lhs.num * rhs.den + lhs.den * rhs.num, lhs.den * rhs.den};
  }

  friend consteval ratio operator-(ratio lhs, ratio rhs) { return lhs + (-rhs); }

  friend consteval ratio operator*(ratio lhs, ratio rhs);

  friend consteval ratio operator/(ratio lhs, ratio rhs)
  {
    return lhs * ratio{rhs.den, rhs.num};
  }
};

consteval bool is-integral(ratio r) { return r.num % r.den == 0; }

consteval ratio common-ratio(ratio r1, ratio r2);

}
```

1 *`ratio`* represents the rational number `num`/`den`.

2 Unless otherwise specified, in the following descriptions, let `R(r)` be `std::ratio<N, D>`, where `N` and `D` are the values of `r.num` and `r.den`.

```
consteval ratio(std::intmax_t n, std::intmax_t d = 1);
```

3 Let `N` and `D` be the values of `n` and `d`. Let `R` be `std::ratio<N, D>`.

4 *Effects*: Equivalent to `R`.

5 *Postconditions*: `num == R::num && den == R::den` is `true`.

```
friend consteval ratio operator*(ratio lhs, ratio rhs);
```

6 Let `Res` be `std::ratio_multiply<R(lhs), R(rhs)>`.

7 *Effects*: Equivalent to: `return {Res::num, Res::den};`

```
consteval ratio common-ratio(ratio r1, ratio r2);
```

8 Let `Res` be equal to

```
std::common_type<std::chrono::duration<int, R(r1)>,
                 std::chrono::duration<int, R(r2)>>::type::period
```

9 *Effects*: Equivalent to: `return {Res::num, Res::den};`

5.3.3 Symbol text [qty.sym.txt]

```
namespace mp_units {

template<std::size_t N, std::size_t M>
class symbol_text {
public:
  std::fixed_u8string<N> utf8;      // exposition only
  std::fixed_string<M> portable;   // exposition only

  // constructors
  constexpr symbol_text(char portable);
  consteval symbol_text(const char (&portable)[N + 1]);
  constexpr symbol_text(const std::fixed_string<N>& portable);
  consteval symbol_text(const char8_t (&utf8)[N + 1], const char (&portable)[M + 1]);
  constexpr symbol_text(const std::fixed_u8string<N>& utf8,
                        const std::fixed_string<M>& portable);

  // observers
  constexpr const auto& utf8() const { return utf8; }
  constexpr const auto& portable() const { return portable; }
  constexpr bool empty() const { return utf8().empty(); }

  // string operations
  template<std::size_t N2, std::size_t M2>
  friend constexpr symbol_text<N + N2, M + M2> operator+(const symbol_text& lhs,
                                                         const symbol_text<N2, M2>& rhs);

  // comparison
  template<std::size_t N2, std::size_t M2>
  friend constexpr bool operator==(const symbol_text& lhs,
                                   const symbol_text<N2, M2>& rhs) noexcept;
  template<std::size_t N2, std::size_t M2>
  friend constexpr auto operator<=>(const symbol_text& lhs,
                                    const symbol_text<N2, M2>& rhs) noexcept;
};

symbol_text(char) -> symbol_text<1, 1>;

template<std::size_t N>
symbol_text(const char (&)[N]) -> symbol_text<N - 1, N - 1>;

template<std::size_t N>
symbol_text(const std::fixed_string<N>&) -> symbol_text<N, N>;

template<std::size_t N, std::size_t M>
symbol_text(const char8_t (&)[N], const char (&)[M]) -> symbol_text<N - 1, M - 1>;

template<std::size_t N, std::size_t M>
symbol_text(const std::fixed_u8string<N>&, const std::fixed_string<M>&) -> symbol_text<N, M>;

}
```

1 `symbol_text` represents a symbol text. *`utf8`* stores its UTF-8 representation, and *`portable`* stores its portable representation. `symbol_text<N, M>` is a structural type (N4971, [temp.param]).

2 In the descriptions that follow, it is a *Precondition* that

(2.1) — values of `char` are in the basic literal character set (N4971, [lex.charset]), and

(2.2) — for a parameter of the form `const CharT (&txt)[M]`, `(txt[M - 1] == CharT())` is `true`.

```
constexpr symbol_text(char portable);
consteval symbol_text(const char (&portable)[N + 1]);
constexpr symbol_text(const std::fixed_string<N>& portable);
consteval symbol_text(const char8_t (&utf8)[N + 1], const char (&portable)[M + 1]);
constexpr symbol_text(const std::fixed_u8string<N>& utf8, const std::fixed_string<M>& portable);
```

3 For the constructors without a parameter named `utf8`, let `utf8` be:

```
std::bit_cast<std::fixed_u8string<N>>(std::basic_fixed_string(portable))
```

4 *Effects*: Equivalent to the *mem-initializer-list*:

```
utf8{utf8}, portable{portable}
```

```
template<std::size_t N2, std::size_t M2>
friend constexpr symbol_text<N + N2, M + M2> operator+(const symbol_text& lhs,
                                                       const symbol_text<N2, M2>& rhs);
```

5 *Effects*: Equivalent to:

```
return symbol_text<N + N2, M + M2>(lhs.utf8() + rhs.utf8(),
                                   lhs.portable() + rhs.portable());
```

```
template<std::size_t N2, std::size_t M2>
friend constexpr bool operator==(const symbol_text& lhs,
                                 const symbol_text<N2, M2>& rhs) noexcept;
template<std::size_t N2, std::size_t M2>
friend constexpr auto operator<=>(const symbol_text& lhs,
                                  const symbol_text<N2, M2>& rhs) noexcept;
```

6 Let `@` be the *operator*.

7 *Effects*: Equivalent to:

```
return std::make_tuple(std::cref(lhs.utf8()), std::cref(lhs.portable())) @
       std::make_tuple(std::cref(rhs.utf8()), std::cref(rhs.portable()));
```

5.3.4 Symbolic expressions [qty.sym.expr]

5.3.4.1 General [qty.sym.expr.general]

1 Subclause 5.3.4 specifies the components used to maintain ordered, simplified, and readable argument lists in the names of specializations.

[*Example 1*:

```
using namespace si::unit_symbols;
int x = kg * km / square(h);  // error: cannot construct from
  // derived_unit<si::kilo_<si::gram>, si::kilo_<si::metre>, per<power<non_si::hour, 2>>>
```

The library ensures `decltype(kg * km / square(h))` is styled-like as commented in diagnostics, provided that, in the implementation-defined total order of types, `decltype(kg)` is less than `decltype(km)`. — *end example*]

5.3.4.2 Concept *SymbolicConstant* [qty.sym.expr.concepts]

```
template<typename T>
concept SymbolicConstant =  // exposition only
  std::is_empty_v<T> && std::is_final_v<T> && std::is_trivially_default_constructible_v<T> &&
  std::is_trivially_copy_constructible_v<T> && std::is_trivially_move_constructible_v<T> &&
  std::is_trivially_destructible_v<T>;
```

1 The concept *`SymbolicConstant`* is used to constrain the types that are used in symbolic expressions.

5.3.4.3 Types [qty.sym.expr.types]

```
namespace mp_units {

template<typename T, typename... Ts>
struct per final {};
}
```

1 `per` is used to store arguments with negative exponents. A specialization of `per` represents the product of the inverse of its template arguments. A program that instantiates a specialization of `per` that is not a possible result of the library specifications is ill-formed, no diagnostic required.

```
namespace mp_units {

template<typename F, int Num, int... Den>
  requires see below
struct power final {
  using factor = F;                              // exposition only
  static constexpr ratio exponent{Num, Den...};  // exposition only
};

}
```

2 `power` represents a power (IEC 60050, 102-02-08) of the form $\texttt{F}^{\texttt{Num/Den}}$.

[*Note 1*: `Den` is optional to shorten the type name when `Den` is `1`. — *end note*]

A program that instantiates a specialization of `power` that is not a possible result of the library specifications is ill-formed, no diagnostic required.

3 Let `r` be *`ratio`*`{Num, Den...}`. Let *`is-valid-ratio`* be `true` if `r` is a valid constant expression, and `false` otherwise. The expression in the *requires-clause* is equivalent to:

```
is-valid-ratio && (r > ratio{0}) && (r != ratio{1})
```

5.3.4.4 Algorithms [qty.sym.expr.algos]

```
template<typename T>
using expr-type = see below;  // exposition only
```

1 *`expr-type`*`<T>` denotes `U` if `T` is of the form `power<U, Ints...>`, and `T` otherwise.

```
template<typename T, typename U>
consteval bool type-less-impl();  // exposition only
```

2 *Returns*: `true` if `T` is less than `U` in an implementation-defined total order for types, and `false` otherwise.

```
template<typename Lhs, typename Rhs>
struct type-less :  // exposition only
    std::bool_constant<is-specialization-of<Rhs, power>() ||
                       type-less-impl<expr-type<Lhs>, expr-type<Rhs>>()> {};
```

3 *`type-less`* meets the requirements of the `Pred` parameter of the symbolic expression algorithms below.

```
template<typename... Ts>
struct type-list {};      // exposition only

template<typename OneType, typename... Ts>
struct expr-fractions {   // exposition only
  using num = see below;  // exposition only
  using den = see below;  // exposition only
}
```

4 *`expr-fractions`* divides a symbolic expression to numerator and denominator parts. Let `EF` be a specialization of *`expr-fractions`*.

(4.1) — If `EF` is of the form *`expr-fractions`*`<OneType, Ts..., per<Us...>>`, then

(4.1.1) — `EF::`*`num`* denotes *`type-list`*`<Ts...>`, and

(4.1.2) — `EF::`*`den`* denotes *`type-list`*`<Us...>`.

(4.2) — Otherwise, `EF` is of the form *`expr-fractions`*`<OneType, Ts...>`, and

(4.2.1) — `EF::`*`num`* denotes *`type-list`*`<Ts...>`, and

(4.2.2) — `EF::`*`den`* denotes *`type-list`*`<>`.

5 The symbolic expression algorithms perform operations on symbolic constants. A symbolic constant is a type that is a model of *`SymbolicConstant`*.

[*Example 1*: The dimension `dim_length`, the quantity `time`, and the unit `one` are symbolic constants. — *end example*]

The algorithms also support powers with a symbolic constant base and a rational exponent, products thereof, and fractions thereof.

```
template<template<typename...> typename To, typename OneType,
         template<typename, typename> typename Pred = type-less, typename Lhs, typename Rhs>
consteval auto expr-multiply(Lhs, Rhs);         // exposition only

template<template<typename...> typename To, typename OneType,
         template<typename, typename> typename Pred = type-less, typename Lhs, typename Rhs>
consteval auto expr-divide(Lhs lhs, Rhs rhs);   // exposition only

template<template<typename...> typename To, typename OneType, typename T>
consteval auto expr-invert(T);                  // exposition only

template<std::intmax_t Num, std::intmax_t Den, template<typename...> typename To,
         typename OneType, template<typename, typename> typename Pred = type-less, typename T>
  requires(Den != 0)
consteval auto expr-pow(T);                     // exposition only
```

6 *Mandates*:

(6.1) — `OneType` is the neutral element (IEC 60050, 102-01-19) of the operation, and

(6.2) — `Pred` is a *Cpp17BinaryTypeTrait* (N4971, [meta.rqmts]) with a base characteristic of `std::bool_constant<B>`. `Pred<T, U>` implements a total order for types; `B` is `true` if `T` is ordered before `U`, and `false` otherwise.

7 *Effects*: First, inputs to the operations are obtained from the types of the function parameters. If the type of a function parameter is:

(7.1) — A specialization of `To`, then its input is the product of its template arguments, and the following also apply.

(7.2) — A specialization of `per`, then its input is the product of the inverse of its template arguments, and the following also apply.

(7.3) — A specialization of the form `power<F, Num>`, then its input is $\texttt{F}^{\texttt{Num}}$, or a specialization of the form `power<F, Num, Den>`, then its input is $\texttt{F}^{\texttt{Num/Den}}$, and the following also applies.

(7.4) — Otherwise, the input is the symbolic constant itself.

[*Example 2*: Item by item, this algorithm step goes from the C++ parameter type `decltype(km / square(h))`, styled in diagnostics like `derived_unit<si::kilo_<si::metre>, per<power<non_si::hour, 2>>`,

(7.5) — to `decltype(km)` × `per<power<decltype(h), 2>` (product of `To`'s arguments),

(7.6) — to `decltype(km)` × 1/`power<decltype(h), 2>` (product of inverse of `per`'s arguments),

(7.7) — to `decltype(km)` × 1/$\texttt{decltype(h)}^2$ (powers as powers),

(7.8) — to $\mathrm{a} \times 1/\mathrm{b}^2$ where a = `decltype(km)` and b = `decltype(h)` (symbolic substitution) in the mathematical domain.

— *end example*]

8 Then, the operation takes place:

(8.1) — *`expr-multiply`* multiplies its inputs,

(8.2) — *`expr-divide`* divides the input of its first parameter by the input of its second parameter,

(8.3) — *`expr-invert`* divides 1 by its input, and

(8.4) — *`expr-pow`* raises its input to the `Num`/`Den`.

9 Finally, let r be the result of the operation simplified as follows:

(9.1) — All terms are part of the same fraction (if any).

(9.2) — There is at most a single term with a given symbolic constant.

(9.3) — There are no negative exponents.

(9.4) — 1 is only present as r and as a numerator with a denominator not equal to 1.

[*Example 3*: Item by item:
$x \times 1/y \times 1/x^2$

= $x/(yx^2)$ (single fraction)
= x^{-1}/y (unique symbolic constants)
= $1/(x^1y)$ (positive exponents)
= $1/(xy)$ (non-redundant 1s)
— *end example*]

10 *Returns*: r is mapped to the return type:

(10.1) — If $r = 1$, returns `OneType{}`.

(10.2) — Otherwise, if r is a symbolic constant, returns r.

(10.3) — Otherwise, first applies the following mappings to the terms of r:

(10.3.1) — $x^{n/d}$ is mapped to `power<`x`, `n`, `d`>`, and x^n is mapped to `power<`x`, `n`>`, and

(10.3.2) — 1 is mapped to `OneType{}`.

(10.4) — Then, a denominator x of r (if any) is mapped to `per<`x`>`.

(10.5) — Then, sorts r without `per` (if any) and the template arguments of `per` (if any) according to `Pred`.

(10.6) — Finally, returns `To<`r`>{}`, where `per` (if any) is the last argument.

11 *Remarks*: A valid template argument list for `To` and `per` is formed by interspersing commas between each mapped term. If a mapping to `std::intmax_t` is not representable, the program is ill-formed.

12 *`expr-map`* maps the contents of one symbolic expression to another resulting in a different type list.

```
template<template<typename> typename Proj, template<typename...> typename To, typename OneType,
         template<typename, typename> typename Pred = type-less, typename T>
consteval auto expr-map(T);   // exposition only
```

13 Let

(13.1) — *`expr-type-map`*`<U>` be `power<Proj<F>, Ints...>` if `U` is of the form `power<F, Ints...>`, and `Proj<U>` otherwise,

(13.2) — *`map-power`*`(u)` be `pow<Ints...>(F{})` if `decltype(u)` is of the form `power<F, Ints...>`, and `u` otherwise, and

(13.3) — `Nums` and `Dens` be packs denoting the template arguments of `T::`*`nums`* and `T::`*`dens`*, respectively.

14 *Returns*:

```
(OneType{} * ... * map-power(expr-type-map<Nums>{})) /
(OneType{} * ... * map-power(expr-type-map<Dens>{}))
```

5.4 Reference [qty.ref]

5.4.1 General [qty.ref.general]

1 Subclause 5.4 specifies the components for describing the reference of a quantity (IEC 60050, 112-01-01).

5.4.2 Dimension [qty.dim]

5.4.2.1 General [qty.dim.general]

1 Subclause 5.4.2 specifies the components for defining the dimension of a quantity (IEC 60050, 112-01-11).

5.4.2.2 Concepts [qty.dim.concepts]

```
template<typename T>
concept Dimension = SymbolicConstant<T> && std::derived_from<T, dimension-interface>;

template<typename T>
concept BaseDimension =   // exposition only
  Dimension<T> && (is-derived-from-specialization-of<T, base_dimension>());

template<typename T, auto D>
concept DimensionOf = Dimension<T> && Dimension<decltype(D)> && (T{} == D);
```

5.4.2.3 Types [qty.dim.types]

```
  namespace mp_units {
```

```
template<symbol_text Symbol>
struct base_dimension : dimension-interface {
  static constexpr auto symbol = Symbol;  // exposition only
};

}
```

1 base_dimension is used to define the dimension of a base quantity (IEC 60050, 112-01-08). Symbol is its symbolic representation.

[*Example 1*:

```
inline constexpr struct dim_length final : base_dimension<"L"> {} dim_length;
```

— *end example*]

```
namespace mp_units {

template<typename... Expr>
struct derived-dimension-impl  // exposition only
    : expr-fractions<struct dimension_one, Expr...> {};

template<SymbolicConstant... Expr>
struct derived_dimension final : dimension-interface, derived-dimension-impl<Expr...> {};

}
```

2 derived_dimension is used by the library to represent the dimension of a derived quantity (IEC 60050, 112-01-10).

[*Example 2*:

```
constexpr auto dim_acceleration = isq::speed.dimension / isq::dim_time;
int x = dim_acceleration;  // error: cannot construct from
  // derived_dimension<isq::dim_length, per<power<isq::dim_time, 2>>>
```

— *end example*]

A program that instantiates a specialization of derived_dimension that is not a possible result of the library specifications is ill-formed, no diagnostic required.

```
namespace mp_units {

struct dimension_one final : dimension-interface, derived-dimension-impl<> {};

}
```

3 dimension_one represents the dimension of a quantity of dimension one (IEC 60050, 112-01-13).

5.4.2.4 Operations [qty.dim.ops]

```
namespace mp_units {

struct dimension-interface {  // exposition only
  template<Dimension Lhs, Dimension Rhs>
  friend consteval Dimension auto operator*(Lhs, Rhs);

  template<Dimension Lhs, Dimension Rhs>
  friend consteval Dimension auto operator/(Lhs, Rhs);

  template<Dimension Lhs, Dimension Rhs>
  friend consteval bool operator==(Lhs, Rhs);
};

}
```

```
template<Dimension Lhs, Dimension Rhs>
friend consteval Dimension auto operator*(Lhs, Rhs);
```

1 *Returns*: expr-multiply<derived_dimension, struct dimension_one>(Lhs{}, Rhs{}).

```
template<Dimension Lhs, Dimension Rhs>
friend consteval Dimension auto operator/(Lhs, Rhs);
```

2 *Returns*: *`expr-divide`*`<derived_dimension, struct dimension_one>(Lhs{}, Rhs{})`.

```
template<Dimension Lhs, Dimension Rhs>
friend consteval bool operator==(Lhs, Rhs);
```

3 *Returns*: `std::is_same_v<Lhs, Rhs>`.

```
consteval Dimension auto inverse(Dimension auto d);
```

4 *Returns*: `dimension_one / d`.

```
template<std::intmax_t Num, std::intmax_t Den = 1, Dimension D>
  requires(Den != 0)
consteval Dimension auto pow(D d);
```

5 *Returns*: *`expr-pow`*`<Num, Den, derived_dimension, struct dimension_one>(d)`.

```
consteval Dimension auto sqrt(Dimension auto d);
```

6 *Returns*: `pow<1, 2>(d)`.

```
consteval Dimension auto cbrt(Dimension auto d);
```

7 *Returns*: `pow<1, 3>(d)`.

5.4.2.5 Symbol formatting [qty.dim.sym.fmt]

```
template<std::size_t N>
constexpr auto get-symbol-exponent(const fixed_string<N>& sign, ratio r);   // exposition only
```

1 Let *`num-to-txt`*`(x)` be an expression equal to an instantiation of `symbo_text` with the symbol text of `x` in base 10.

2 *Effects*: Determines `txt` as follows:

(2.1) — If `r.den != 1` is `true`, let `txt` be

```
symbol_text("^") + symbol_text(sign) + symbol_text("(") + num-to-txt(r.num) +
  symbol_text("/") + num-to-txt(r.den) + symbol_text(")")
```

(2.2) — Otherwise, let `txt` be an instantiation of `symbol_text` such that:

(2.2.1) — `txt.portable()` equals

```
"^" + sign + num-to-txt(Num).portable()
```

(2.2.2) — `txt.utf8()` equals `txt.portable()` after replacing its elements according to Table 4.

3 *Returns*: `txt`.

Table 4 — UTF-8 exponent transformation [tab:qty.exp.portable.to.utf8]

Input value	Replacement value
’^’	None (erased)
’-’	U+207B SUPERSCRIPT MINUS
’0’	U+2070 SUPERSCRIPT ZERO
’1’	U+00B9 SUPERSCRIPT ONE
’2’	U+00B2 SUPERSCRIPT TWO
’3’	U+00B3 SUPERSCRIPT THREE
’4’	U+2074 SUPERSCRIPT FOUR
’5’	U+2075 SUPERSCRIPT FIVE
’6’	U+2076 SUPERSCRIPT SIX
’7’	U+2077 SUPERSCRIPT SEVEN
’8’	U+2078 SUPERSCRIPT EIGHT
’9’	U+2079 SUPERSCRIPT NINE

```
template<typename CharT, std::size_t N, std::size_t M, std::output_iterator<CharT> Out>
constexpr Out copy-symbol-text(const symbol_text<N, M>& txt,  // exposition only
                               character_set char_set, Out out);
```

4 *Effects*: Copies `txt` to `out` according to `char_set` as follows:

(4.1) — If `char_set == character_set::utf8` is `true`, then:

(4.1.1) — If `std::is_same_v<CharT, char8_t>` is `true`, equivalent to:

```
out = std::ranges::copy(txt.utf8(), out)
```

(4.1.2) — Otherwise, if `std::is_same_v<CharT, char>` is `true`, then:

(4.1.2.1) — If

```
std::text_encoding::literal().mib() != std::text_encoding::id::UTF8
```

is `true`, equivalent to:

```
out = std::ranges::copy(txt.portable(), out)
```

(4.1.2.2) — Otherwise, equivalent to:

```
for (const char8_t ch : txt.utf8()) *out++ = static_cast<char>(ch);
```

(4.1.3) — Otherwise, no effect.

[*Note 1*: UTF-8 text can't be copied to `CharT` output. — *end note*]

(4.2) — Otherwise, if `std::is_same_v<CharT, char>` is `true`, equivalent to:

```
out = std::ranges::copy(txt.portable(), out)
```

(4.3) — Otherwise, no effect.

[*Note 2*: Portable text can't be copied to `CharT` output. — *end note*]

5 *Returns*: `out`.

6 *Throws*: `std::invalid_argument` if there is no effect.

```
template<typename CharT = char, std::output_iterator<CharT> Out, Dimension D>
constexpr Out dimension_symbol_to(Out out, D d, const dimension_symbol_formatting& fmt = {});
```

7 Let `E` be the type of the dimension whose symbol text is being copied.

8 *Effects*: Copies the symbol text of the dimension `D` to `out` according to `fmt` as follows:

(8.1) — If `is-derived-from-specialization-of<D, expr-fractions>()` is `true`, let `Nums` and `Dens` be packs denoting the template arguments of `D::nums` and `D::dens`, respectively.

(8.1.1) — If `sizeof...(Nums) == 0 && sizeof...(Dens) == 0` is `true`, equivalent to `*out++ = '1'`.

(8.1.2) — Otherwise, copies the symbol text of the numerators `(..., void(Nums))`, in that order, as follows. Then, copies the symbol text of the denominators `(..., void(Dens))`, in that order, as follows.

(8.2) — If `E` is a specialization of `power`, copies the symbol text of `E::factor`, as follows. Then, copies the symbol text of `E::exponent` $\neq 1$, determined as follows:

(8.2.1) — If `E` is a denominator, let `sign` be `"-"`. Otherwise, let `sign` be `""`.

(8.2.2) — Let `txt` be equivalent to `get-symbol-exponent(sign, E::exponent)`.

(8.3) — If `E::symbol` is a valid expression, let `txt` be that.

(8.4) — Each time `txt` is determined, equivalent to:

```
out = copy-symbol-text(txt, fmt.char_set, out)
```

9 *Returns*: `out`.

```
template<dimension_symbol_formatting fmt = {}, typename CharT = char, Dimension D>
consteval std::string_view dimension_symbol(D);
```

10 *Returns*: A value `sv` such that [`sv.data()`,`sv.data() + sv.size()`) has static storage duration and the following assertion holds:

```
std::basic_string<CharT> s;
dimension_symbol_to<CharT>(std::back_inserter(s), D{}, fmt);
assert(sv == s);
```

11 [*Example 1*:

```
import mp_units;

using namespace mp_units;

static_assert(dimension_symbol(dimension_one) == "1");
static_assert(dimension_symbol(isq::dim_length) == "L");
static_assert(dimension_symbol(isq::dim_thermodynamic_temperature) == "\u0398");
static_assert(
  dimension_symbol<{character_set::portable}>(isq::dim_thermodynamic_temperature) == "O");
static_assert(dimension_symbol(isq::speed.dimension) == "LT\u207B\u00B9");
static_assert(dimension_symbol<{character_set::portable}>(isq::speed.dimension) == "LT^-1");
static_assert(dimension_symbol(pow<1, 2>(isq::dim_length)) == "L^(1/2)");
```

— *end example*]

5.4.3 Quantity specification [qty.spec]

5.4.3.1 General [qty.spec.general]

1 Subclause 5.4.3 specifies the components for defining a quantity (IEC 60050, 112-01-01).

5.4.3.2 Concepts [qty.spec.concepts]

```
template<typename T>
concept QuantitySpec = SymbolicConstant<T> && std::derived_from<T, quantity-spec-interface>;

template<typename T>
concept QuantityKindSpec =                                  // exposition only
  QuantitySpec<T> && is-specialization-of<T, kind_of_>();

template<typename T>
concept NamedQuantitySpec =                                 // exposition only
  QuantitySpec<T> && is-derived-from-specialization-of<T, quantity_spec>() &&
  (!QuantityKindSpec<T>);

template<typename T>
concept DerivedQuantitySpec =                               // exposition only
  QuantitySpec<T> &&
  (is-specialization-of<T, derived_quantity_spec>() ||
   (QuantityKindSpec<T> &&
    is-specialization-of<decltype(auto(T::quantity-spec)), derived_quantity_spec>()));

template<auto Child, auto Parent>
concept ChildQuantitySpecOf = (is-child-of(Child, Parent));  // exposition only

template<auto To, auto From>
concept NestedQuantityKindSpecOf =                          // exposition only
  QuantitySpec<decltype(From)> && QuantitySpec<decltype(To)> &&
  (get_kind(From) != get_kind(To)) && ChildQuantitySpecOf<To, get_kind(From).quantity-spec>;

template<auto From, auto To>
concept QuantitySpecConvertibleTo =                         // exposition only
  QuantitySpec<decltype(From)> && QuantitySpec<decltype(To)> && implicitly_convertible(From, To);

template<auto From, auto To>
concept QuantitySpecExplicitlyConvertibleTo =               // exposition only
  QuantitySpec<decltype(From)> && QuantitySpec<decltype(To)> && explicitly_convertible(From, To);

template<auto From, auto To>
concept QuantitySpecCastableTo =                            // exposition only
  QuantitySpec<decltype(From)> && QuantitySpec<decltype(To)> && castable(From, To);

template<typename T, auto QS>
concept QuantitySpecOf =
  QuantitySpec<T> && QuantitySpec<decltype(QS)> && QuantitySpecConvertibleTo<T{}, QS> &&
  !NestedQuantityKindSpecOf<T{}, QS> &&
  (QuantityKindSpec<T> || !NestedQuantityKindSpecOf<QS, T{}>);

template<typename T>
concept QSProperty = (!QuantitySpec<T>);                    // exposition only
```

5.4.3.3 Types [qty.spec.types]

5.4.3.3.1 Named [named.qty]

```
namespace mp_units {

struct is_kind {};

template<BaseDimension auto Dim, QSProperty auto... Args>
struct quantity_spec<Dim, Args...> : quantity-spec-interface {
  using base-type = quantity_spec;
  static constexpr BaseDimension auto dimension = Dim;
  static constexpr quantity_character character = see below;
};

template<DerivedQuantitySpec auto Eq, QSProperty auto... Args>
struct quantity_spec<Eq, Args...> : quantity-spec-interface {
  using base-type = quantity_spec;
  static constexpr auto equation = Eq;
  static constexpr Dimension auto dimension = Eq.dimension;
  static constexpr quantity_character character = see below;
};

template<NamedQuantitySpec auto QS, QSProperty auto... Args>
struct quantity_spec<QS, Args...> : quantity-spec-interface {
  using base-type = quantity_spec;
  static constexpr auto parent = QS;
  static constexpr auto equation = parent.equation;   // exposition only, present only
    // if the qualified-id parent.equation is valid and denotes an object
  static constexpr Dimension auto dimension = parent.dimension;
  static constexpr quantity_character character = see below;
};

template<NamedQuantitySpec auto QS, DerivedQuantitySpec auto Eq, QSProperty auto... Args>
  requires QuantitySpecExplicitlyConvertibleTo<Eq, QS>
struct quantity_spec<QS, Eq, Args...> : quantity-spec-interface {
  using base-type = quantity_spec;
  static constexpr auto parent = QS;
  static constexpr auto equation = Eq;
  static constexpr Dimension auto dimension = parent.dimension;
  static constexpr quantity_character character = see below;
};

}
```

1 A *named quantity* is a type that models `NamedQuantitySpec`. A specialization of `quantity_spec` is used as a base type when defining a named quantity.

2 In the following descriptions, let `Q` be a named quantity defined with an alluded signature. The identifier of `Q` represents its quantity name (IEC 60050, 112-01-02).

3 Let `Ch` be an enumerator value of `quantity_character`. The possible arguments to `quantity_spec` are

(3.1) — (a base quantity dimension, `Ch`$_{opt}$),

(3.2) — (a quantity calculus, `Ch`$_{opt}$),

(3.3) — (a named quantity, `Ch`$_{opt}$, `is_kind`$_{opt}$), and

(3.4) — (a named quantity, a quantity calculus, `Ch`$_{opt}$, `is_kind`$_{opt}$).

4 If the first argument is a base quantity dimension, then `Q` is that base quantity (IEC 60050, 112-01-08). If an argument is a quantity calculus (IEC 60050, 112-01-30) C, then `Q` is implicitly convertible from C. If the first argument is a named quantity, then `Q` is of its kind (IEC 60050, 112-01-04).

5 The member `character` represents the set of the numerical value of `Q` (5.5.2.2) and is equal to

(5.1) — `Ch` if specified,

(5.2) — otherwise, `quantity_character::scalar` for the first signature, and

(5.3) — otherwise, `(BC).character`, where `BC` is the argument preceding `Ch` in the signatures above.

6 `is_kind` specifies `Q` to start a new hierarchy tree of a kind.

7 Optional arguments may appear in any order.

8 [*Example 1*:

```
// The first signature defines a base quantity.
inline constexpr struct length final : quantity_spec<dim_length> {
} length;  // Length is a base quantity.

// The second signature defines a derived quantity.
inline constexpr struct area final : quantity_spec<pow<2>(length)> {
} area;  // An area equals length by length.

// The third and fourth signatures add a leaf to a hierarchy of kinds.
inline constexpr struct width final : quantity_spec<length> {
} width;  // Width is a kind of length.

// The fourth signature also refines the calculus required for implicit conversions.
inline constexpr struct angular_measure final :
    quantity_spec<dimensionless, arc_length / radius, is_kind> {
} angular_measure;  // Requires an arc length per radius, not just any quantity of dimension one.
```

— *end example*]

5.4.3.3.2 Derived [derived.qty]

```
namespace mp_units {

template<NamedQuantitySpec Q>
using to-dimension = decltype(auto(Q::dimension));  // exposition only

template<typename... Expr>
struct derived-quantity-spec-impl :                  // exposition only
    quantity-spec-interface,
    expr-fractions<struct dimensionless, Expr...> {
  using base-type = derived-quantity-spec-impl;
  using base = expr-fractions<struct dimensionless, Expr...>;

  static constexpr Dimension auto dimension =
    expr-map<to-dimension, derived_dimension, struct dimension_one>(base{});
  static constexpr quantity_character character = see below;
};

template<SymbolicConstant... Expr>
struct derived_quantity_spec final : derived-quantity-spec-impl<Expr...> {};

}
```

1 `derived_quantity_spec` is used by the library to represent the result of a quantity calculus not equal to a named quantity.

[*Example 1*:

```
constexpr auto area = pow<2>(isq::length);
int x = area;  // error: cannot construct from derived_quantity_spec<power<isq::length, 2>>
```

— *end example*]

A program that instantiates a specialization of `derived_quantity_spec` that is not a possible result of the library specifications is ill-formed, no diagnostic required.

2 Let

(2.1) — `Nums` and `Dens` be packs denoting the template arguments of *`base`*`::`*`nums`* and *`base`*`::`*`dens`*, respectively,

(2.2) — *`QUANTITY-CHARACTER-OF`*`(Pack)` be

```
std::max({quantity_character::scalar, expr-type<Pack>::character...})
```

and

(2.3) — `num_char` be *`QUANTITY-CHARACTER-OF`*`(Nums)` and `den_char` be *`QUANTITY-CHARACTER-OF`*`(Dens)`.

The member `character` is equal to `quantity_character::scalar` if `num_char == den_char` is `true`, and `std::max(num_char, den_char)` otherwise.

5.4.3.3.3 Base quantity of dimension one [dimless.qty]

```
namespace mp_units {

struct dimensionless final : quantity_spec<derived_quantity_spec<>> {};

}
```

1 `dimensionless` represents the base quantity of dimension one (IEC 60050, 112-01-13).

5.4.3.3.4 Kind of [kind.of.qty]

```
namespace mp_units {

template<QuantitySpec Q>
  requires(!QuantityKindSpec<Q>) && (get-kind-tree-root(Q{}) == Q{})
struct kind_of_ final : Q::base-type {
  using base-type = kind_of_;                   // exposition only
  static constexpr auto quantity-spec = Q{};    // exposition only
};

}
```

1 `kind_of<Q>` represents a kind of quantity (IEC 60050, 112-01-04) `Q`.

5.4.3.4 Utilities [qty.spec.utils]

```
template<QuantitySpec auto... From, QuantitySpec Q>
consteval QuantitySpec auto clone-kind-of(Q q);     // exposition only
```

1 *Effects*: Equivalent to:

```
if constexpr ((... && QuantityKindSpec<decltype(From)>))
  return kind_of<Q{}>;
else
  return q;
```

```
template<QuantitySpec Q>
consteval auto remove-kind(Q q);                    // exposition only
```

2 *Effects*: Equivalent to:

```
if constexpr (QuantityKindSpec<Q>)
  return Q::quantity-spec;
else
  return q;
```

```
template<QuantitySpec QS, Unit U>
  requires(!AssociatedUnit<U>) || UnitOf<U, QS{}>
consteval Reference auto make-reference(QS, U u);   // exposition only
```

3 *Effects*: Equivalent to:

```
if constexpr (requires { requires get_quantity_spec(U{}) == QS{}; })
  return u;
```

```
    else
      return reference<QS, U>{};
```

5.4.3.5 Operations [qty.spec.ops]

```
namespace mp_units {

struct quantity-spec-interface {  // exposition only
  template<QuantitySpec Lhs, QuantitySpec Rhs>
  friend consteval QuantitySpec auto operator*(Lhs lhs, Rhs rhs);

  template<QuantitySpec Lhs, QuantitySpec Rhs>
  friend consteval QuantitySpec auto operator/(Lhs lhs, Rhs rhs);

  template<typename Self, UnitOf<Self{}> U>
  consteval Reference auto operator[](this Self self, U u);

  template<typename Self, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
    requires QuantitySpecExplicitlyConvertibleTo<Q::quantity_spec, Self{}>
  constexpr Quantity auto operator()(this Self self, FwdQ&& q);

  template<QuantitySpec Lhs, QuantitySpec Rhs>
  friend consteval bool operator==(Lhs, Rhs);
};

}
```

```
template<QuantitySpec Lhs, QuantitySpec Rhs>
friend consteval QuantitySpec auto operator*(Lhs lhs, Rhs rhs);
```

1 *Returns*:

```
clone-kind-of<Lhs{}, Rhs{}>(expr-multiply<derived_quantity_spec, struct dimensionless>(
  remove-kind(lhs), remove-kind(rhs)))
```

```
template<QuantitySpec Lhs, QuantitySpec Rhs>
friend consteval QuantitySpec auto operator/(Lhs lhs, Rhs rhs);
```

2 *Returns*:

```
clone-kind-of<Lhs{}, Rhs{}>(expr-divide<derived_quantity_spec, struct dimensionless>(
  remove-kind(lhs), remove-kind(rhs)))
```

```
template<typename Self, UnitOf<Self{}> U>
consteval Reference auto operator[](this Self self, U u);
```

3 *Returns*: `make-reference(self, u)`.

```
template<typename Self, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
  requires QuantitySpecExplicitlyConvertibleTo<Q::quantity_spec, Self{}>
constexpr Quantity auto operator()(this Self self, FwdQ&& q);
```

4 *Returns*:

```
quantity{std::forward<FwdQ>(q).numerical-value, make-reference(self, Q::unit)}
```

```
template<QuantitySpec Lhs, QuantitySpec Rhs>
friend consteval bool operator==(Lhs, Rhs);
```

5 *Returns*: `std::is_same_v<Lhs, Rhs>`.

```
consteval QuantitySpec auto inverse(QuantitySpec auto q);
```

6 *Returns*: `dimensionless / q`.

```
template<std::intmax_t Num, std::intmax_t Den = 1, QuantitySpec Q>
  requires(Den != 0)
consteval QuantitySpec auto pow(Q q);
```

7 *Returns*:

```
clone-kind-of<Q{}>(
    expr-pow<Num, Den, derived_quantity_spec, struct dimensionless>(remove-kind(q)));
```

```
consteval QuantitySpec auto sqrt(QuantitySpec auto q);
```

8 *Returns*: `pow<1, 2>(q)`.

```
consteval QuantitySpec auto cbrt(QuantitySpec auto q);
```

9 *Returns*: `pow<1, 3>(q)`.

5.4.3.6 Hierarchy algorithms [qty.spec.hier.algos]

5.4.3.6.1 Conversion [qty.spec.conv]

```
consteval bool implicitly_convertible(QuantitySpec auto from, QuantitySpec auto to);
```

1 *Returns*: TBD.

```
consteval bool explicitly_convertible(QuantitySpec auto from, QuantitySpec auto to);
```

2 *Returns*: TBD.

```
consteval bool castable(QuantitySpec auto from, QuantitySpec auto to);
```

3 *Returns*: TBD.

```
consteval bool interconvertible(QuantitySpec auto qs1, QuantitySpec auto qs2);
```

4 *Returns*: `implicitly_convertible(qs1, qs2) && implicitly_convertible(qs2, qs1)`.

5.4.3.6.2 Get kind [qty.get.kind]

```
template<QuantitySpec Q>
consteval QuantitySpec auto get-kind-tree-root(Q q);   // exposition only
```

1 *Returns*:

(1.1) — If *`QuantityKindSpec`*`<Q>` is `true`, returns *`remove-kind`*`(q)`.

(1.2) — Otherwise, if *`is-derived-from-specialization-of`*`<Q, quantity_spec>()` is `true`, and the specialization of `Q::quantity_spec` has a template argument equal to `is_kind`, returns `q`.

(1.3) — Otherwise, if `Q::`*`parent`* is a valid expression, returns *`get-kind-tree-root`*`(Q::`*`parent`*`)`.

(1.4) — Otherwise, if *`DerivedQuantitySpec`*`<Q>` is `true`, returns

```
expr-map<to-kind, derived_quantity_spec, struct dimensionless>(q)
```

where *`to-kind`* is defined as follows:

```
template<QuantitySpec Q>
using to-kind = decltype(get-kind-tree-root(Q{}));   // exposition only
```

(1.5) — Otherwise, returns `q`.

```
template<QuantitySpec Q>
consteval QuantityKindSpec auto get_kind(Q);
```

2 *Returns*: `kind_of<`*`get-kind-tree-root`*`(Q{})>`.

5.4.3.6.3 Get common quantity specification [get.common.qty.spec]

```
consteval QuantitySpec auto get_common_quantity_spec(QuantitySpec auto... qs)
  requires see below;
```

1 Let

(1.1) — `q1` be `qs...[0]`,

(1.2) — `q2` be `qs...[1]`,

(1.3) — `Q1` be `decltype(q1)`,

(1.4) — `Q2` be `decltype(q2)`, and

(1.5) — `rest` be a pack denoting the elements of `qs` without `q1` and `q2`.

2 *Effects*: Equivalent to:

```
if constexpr (sizeof...(qs) == 1)
  return q1;
else if constexpr (sizeof...(qs) == 2) {
  using QQ1 = decltype(remove-kind(q1));
  using QQ2 = decltype(remove-kind(q2));

  if constexpr (std::is_same_v<Q1, Q2>)
    return q1;
  else if constexpr (NestedQuantityKindSpecOf<Q1{}, Q2{}>)
    return QQ1{};
  else if constexpr (NestedQuantityKindSpecOf<Q2{}, Q1{}>)
    return QQ2{};
  else if constexpr ((QuantityKindSpec<Q1> && !QuantityKindSpec<Q2>) ||
                     (DerivedQuantitySpec<QQ1> && NamedQuantitySpec<QQ2> &&
                      implicitly_convertible(Q1{}, Q2{})))
    return q2;
  else if constexpr ((!QuantityKindSpec<Q1> && QuantityKindSpec<Q2>) ||
                     (NamedQuantitySpec<QQ1> && DerivedQuantitySpec<QQ2> &&
                      implicitly_convertible(Q2{}, Q1{})))
    return q1;
  else if constexpr (constexpr auto common_base = get-common-base<Q1{}, Q2{}>())
    return *common_base;
  else if constexpr (implicitly_convertible(Q1{}, Q2{}))
    return q2;
  else if constexpr (implicitly_convertible(Q2{}, Q1{}))
    return q1;
  else if constexpr (implicitly_convertible(get-kind-tree-root(Q1{}),
                                            get-kind-tree-root(Q2{})))
    return get-kind-tree-root(q2);
  else
    return get-kind-tree-root(q1);
} else
  return get_common_quantity_spec(get_common_quantity_spec(q1, q2), rest...);
```

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(sizeof...(qs) != 0 &&
 (sizeof...(qs) == 1 ||
  (sizeof...(qs) == 2 &&
   (QuantitySpecConvertibleTo<get-kind-tree-root(Q1{}), get-kind-tree-root(Q2{})> ||
    QuantitySpecConvertibleTo<get-kind-tree-root(Q2{}), get-kind-tree-root(Q1{})>)) ||
  requires { get_common_quantity_spec(get_common_quantity_spec(q1, q2), rest...); }))
```

5.4.3.6.4 Get common base [qty.get.common.base]

1 In this subclause and 5.4.3.6.5, let the kind of quantity (IEC 60050, 112-01-04) hierarchy of `q` be the tuple $h(\texttt{q}) = (\texttt{q}, \texttt{q.par}, \texttt{q.par.par}, \ldots)$, where `par` is *`parent`*.

```
template<QuantitySpec auto A, QuantitySpec auto B>
consteval auto get-common-base();   // exposition only
```

2 Let

(2.1) — a_s be the number of elements in $h(\texttt{A})$,

(2.2) — b_s be the number of elements in $h(\texttt{B})$,

(2.3) — s be $\min(a_s, b_s)$,

(2.4) — A be a tuple of the last s elements of $h(\texttt{A})$, and

(2.5) — B be a tuple of the last s elements of $h(\texttt{B})$.

3 *Effects*: Looks for x, the first pair-wise equal element in A and B.

4 *Returns*: `std::optional(`x`)`, if x is found, and `std::optional<`*`unspecified`*`>()` otherwise.

5.4.3.6.5 Is child of [qty.is.child.of]

```
template<QuantitySpec Child, QuantitySpec Parent>
consteval bool is-child-of(Child ch, Parent p);   // exposition only
```

1 *Returns*: If h(`p`) has more elements than h(`ch`), returns `false`. Otherwise, let C be a tuple of the last s elements of h(`ch`), where s is the number of elements in h(`p`). Returns C_0 `== p`.

5.4.4 Unit [qty.unit]

5.4.4.1 General [qty.unit.general]

1 Subclause 5.4.4 specifies the components for defining a unit of measurement (IEC 60050, 112-01-14).

5.4.4.2 Magnitude [qty.unit.mag]

5.4.4.2.1 General [qty.unit.mag.general]

1 Subclause 5.4.4.2 specifies the components used to represent the numerical value (IEC 60050, 112-01-29) of a unit with support for powers (IEC 60050, 102-02-08) of real numbers (IEC 60050, 102-02-05).

5.4.4.2.2 Concepts [qty.unit.mag.concepts]

```
template<typename T>
concept MagConstant = SymbolicConstant<T> && is-derived-from-specialization-of<T, mag_constant>();

template<typename T>
concept UnitMagnitude = (is-specialization-of<T, unit-magnitude>());

template<typename T>
concept MagArg = std::integral<T> || MagConstant<T>;   // exposition only
```

5.4.4.2.3 Types [qty.unit.mag.types]

```
namespace mp_units {

template<symbol_text Symbol, long double Value>
  requires(Value > 0)
struct mag_constant {
  static constexpr auto symbol = Symbol;        // exposition only
  static constexpr long double value = Value;   // exposition only
};

}
```

A specialization of `mag_constant` represents a real number (IEC 60050, 102-02-05). `Symbol` is its symbol, and `Value` is (an approximation of) its value.

```
namespace mp_units {

template<auto... Ms>
struct unit-magnitude {   // exposition only
  // 5.4.4.2.4, operations

  template<UnitMagnitude M>
  friend consteval UnitMagnitude auto operator*(unit-magnitude lhs, M rhs);

  friend consteval auto operator/(unit-magnitude lhs, UnitMagnitude auto rhs);

  template<UnitMagnitude Rhs>
  friend consteval bool operator==(unit-magnitude, Rhs);

  template<int Num, int Den = 1>
  friend consteval auto pow(unit-magnitude);                      // exposition only

  // 5.4.4.2.5, utilities

  friend consteval bool is-positive-integral-power(unit-magnitude);   // exposition only
```

```
    template<auto... Ms2>
    friend consteval auto common-magnitude(unit-magnitude,             // exposition only
                                            unit-magnitude<Ms2...>);

    template<typename CharT, std::output_iterator<CharT> Out>
    friend constexpr Out magnitude-symbol(Out out, unit-magnitude,     // exposition only
                                          const unit_symbol_formatting& fmt);
  };

  }
```

1 A specialization of *unit-magnitude* represents the product of its template arguments.

2 For the purposes of specifying the implementation-defined limits, let the representation of the terms of *unit-magnitude* be the structure

```
  struct {
    ratio exp;
    base-type base;
  };
```

representing the number $\texttt{base}^{\texttt{exp}}$, where *base-type* is a model of *MagArg*.

(2.1) — There is a single term for each *base-type*.

(2.2) — `exp.num` is not expanded into base.

[*Note 1*: $2^3 = 8$ is not permitted. — *end note*]

(2.3) — `exp.den` can reduce the base.

[*Note 2*: $4^{1/2} = 2$ is permitted. — *end note*]

(2.4) — If the result of an operation on `std::intmax_t` values is undefined, the behavior is implementation-defined.

5.4.4.2.4 Operations [qty.unit.mag.ops]

```
template<UnitMagnitude M>
friend consteval UnitMagnitude auto operator*(unit-magnitude lhs, M rhs);
```

1 *Returns*:

(1.1) — If `sizeof...(Ms) == 0` is `true`, returns `rhs`.

(1.2) — Otherwise, if `std::is_same_v<M, unit-magnitude<>>`, returns `lhs`.

(1.3) — Otherwise, returns an unspecified value equal to `lhs` × `rhs`.

```
friend consteval auto operator/(unit-magnitude lhs, UnitMagnitude auto rhs);
```

2 *Returns*: `lhs * pow<-1>(rhs)`.

```
template<UnitMagnitude Rhs>
friend consteval bool operator==(unit-magnitude, Rhs);
```

3 *Returns*: `std::is_same_v<unit-magnitude, Rhs>`.

```
template<int Num, int Den = 1>
friend consteval auto pow(unit-magnitude base);   // exposition only
```

4 *Returns*:

(4.1) — If `Num == 0` is `true`, returns `unit-magnitude<>{}`.

(4.2) — Otherwise, returns an unspecified value equal to $\texttt{base}^{\texttt{Num/Den}}$.

```
template<MagArg auto V>
constexpr UnitMagnitude auto mag = see below;
```

5 *Constraints*: `V` is greater than 0.

6 *Effects*: If `MagConstant<decltype(V)>` is satisfied, initializes `mag` with `unit-magnitude<V>{}`. Otherwise, initializes `mag` with an unspecified value equal to `V`.

```
template<std::intmax_t N, std::intmax_t D>
  requires(N > 0)
constexpr UnitMagnitude auto mag_ratio = see below;
```

7 *Effects*: Initializes `mag_ratio` with an unspecified value equal to `N`/`D`.

```
template<MagArg auto Base, int Num, int Den = 1>
constexpr UnitMagnitude auto mag_power = pow<Num, Den>(mag<Base>);
```

8 *Constraints*: `Base` is greater than 0.

5.4.4.2.5 Utilities [qty.unit.mag.utils]

```
friend consteval bool is-positive-integral-power(unit-magnitude x);             // exposition only
```

1 *Returns*: `false` if `x` has a negative or rational exponent, and `true` otherwise.

```
template<auto... Ms2>
friend consteval auto common-magnitude(unit-magnitude, unit-magnitude<Ms2...>);  // exposition only
```

2 *Returns*: The largest magnitude `C` such that each input magnitude is expressible by only positive powers relative to `C`.

```
template<typename CharT, std::output_iterator<CharT> Out>
friend constexpr Out magnitude-symbol(Out out, unit-magnitude,                  // exposition only
                                      const unit_symbol_formatting& fmt);
```

3 *Effects*: TBD.

4 *Returns*: `out`.

5.4.4.3 Traits [qty.unit.traits]

```
template<Unit auto U>
constexpr bool space_before_unit_symbol = true;
```

1 The formatting functions (5.4.4.10) use `space_before_unit_symbol` to determine whether there is a space between the numerical value and the unit symbol.

2 *Remarks*: Pursuant to N4971, [namespace.std] (4.1), users may specialize `space_before_unit_symbol` for cv-unqualified program-defined types. Such specializations shall be usable in constant expressions (N4971, [expr.const]) and have type `const bool`.

5.4.4.4 Concepts [qty.unit.concepts]

```
template<typename T>
concept Unit = SymbolicConstant<T> && std::derived_from<T, unit-interface>;

template<typename T>
concept PrefixableUnit = Unit<T> && is-derived-from-specialization-of<T, named_unit>();

template<typename T>
concept AssociatedUnit = Unit<U> && has-associated-quantity(U{});

template<typename U, auto QS>
concept UnitOf = AssociatedUnit<U> && QuantitySpec<decltype(QS)> &&
                 QuantitySpecConvertibleTo<get_quantity_spec(U{}), QS> &&
                 (get_kind(QS) == get_kind(get_quantity_spec(U{})) ||
                  !NestedQuantityKindSpecOf<get_quantity_spec(U{}), QS>);

template<auto From, auto To>
concept UnitConvertibleTo =                                     // exposition only
  Unit<decltype(From)> && Unit<decltype(To)> && (convertible(From, To));

template<typename U, auto FromU, auto QS>
concept UnitCompatibleWith =                                    // exposition only
  Unit<U> && Unit<decltype(FromU)> && QuantitySpec<decltype(QS)> &&
  (!AssociatedUnit<U> || UnitOf<U, QS>) && UnitConvertibleTo<FromU, U{}>;
```

```
template<typename T>
concept OffsetUnit = Unit<T> && requires { T::point-origin; };   // exposition only

template<typename From, typename To>
concept PotentiallyConvertibleTo =                               // exposition only
  Unit<From> && Unit<To> &&
  ((AssociatedUnit<From> && AssociatedUnit<To> &&
    implicitly_convertible(get_quantity_spec(From{}), get_quantity_spec(To{}))) ||
   (!AssociatedUnit<From> && !AssociatedUnit<To>));
```

5.4.4.5 Types [qty.unit.types]

5.4.4.5.1 Canonical [qty.canon.unit]

```
namespace mp_units {

template<UnitMagnitude M, Unit U>
struct canonical-unit {   // exposition only
  M mag;
  U reference_unit;
};

}
```

1 `canonical-unit` represents a unit expressed in terms of base units (IEC 60050, 112-01-18).

[*Note 1*: Other types representing units are equal only if they have the same type. `canonical-unit` is used to implement binary relations other than equality. — *end note*]

`reference_unit` is simplified (5.3.4.4).

```
consteval auto get-canonical-unit(Unit auto u);   // exposition only
```

2 *Returns*: The instantiation of `canonical-unit` for `u`.

5.4.4.5.2 Scaled [qty.scaled.unit]

```
namespace mp_units {

template<UnitMagnitude auto M, Unit U>
  requires(M != unit-magnitude<>{} && M != mag<1>)
struct scaled_unit final : unit-interface {
  using base-type = scaled_unit;                        // exposition only
  static constexpr UnitMagnitude auto mag = M;          // exposition only
  static constexpr U reference-unit{};                  // exposition only
  static constexpr auto point-origin = U::point_origin; // exposition only, present only
    // if the qualified-id U::point_origin is valid and denotes an object
};

}
```

1 `scaled_unit<M, U>` is used by the library to represent the unit M × U.

5.4.4.5.3 Named [qty.named.unit]

```
namespace mp_units {

template<symbol_text Symbol, QuantityKindSpec auto QS>
  requires(!Symbol.empty()) && BaseDimension<decltype(auto(QS.dimension))>
struct named_unit<Symbol, QS> : unit-interface {
  using base-type = named_unit;                 // exposition only
  static constexpr auto symbol = Symbol;        // exposition only
  static constexpr auto quantity-spec = QS;     // exposition only
};

template<symbol_text Symbol, QuantityKindSpec auto QS, PointOrigin auto PO>
  requires(!Symbol.empty()) && BaseDimension<decltype(auto(QS.dimension))>
struct named_unit<Symbol, QS, PO> : unit-interface {
  using base-type = named_unit;                 // exposition only
  static constexpr auto symbol = Symbol;        // exposition only
```

```
    static constexpr auto quantity-spec = QS;  // exposition only
    static constexpr auto point-origin = PO;   // exposition only
  };

  template<symbol_text Symbol>
    requires(!Symbol.empty())
  struct named_unit<Symbol> : unit-interface {
    using base-type = named_unit;              // exposition only
    static constexpr auto symbol = Symbol;     // exposition only
  };

  template<symbol_text Symbol, Unit auto U>
    requires(!Symbol.empty())
  struct named_unit<Symbol, U> : decltype(U)::base-type {
    using base-type = named_unit;              // exposition only
    static constexpr auto symbol = Symbol;     // exposition only
  };

  template<symbol_text Symbol, Unit auto U, PointOrigin auto PO>
    requires(!Symbol.empty())
  struct named_unit<Symbol, U, PO> : decltype(U)::base-type {
    using base-type = named_unit;              // exposition only
    static constexpr auto symbol = Symbol;     // exposition only
    static constexpr auto point-origin = PO;   // exposition only
  };

  template<symbol_text Symbol, AssociatedUnit auto U, QuantityKindSpec auto QS>
    requires(!Symbol.empty()) && (QS.dimension == get-associated-quantity(U).dimension)
  struct named_unit<Symbol, U, QS> : decltype(U)::base-type {
    using base-type = named_unit;              // exposition only
    static constexpr auto symbol = Symbol;     // exposition only
    static constexpr auto quantity-spec = QS;  // exposition only
  };

  template<symbol_text Symbol, AssociatedUnit auto U, QuantityKindSpec auto QS,
           PointOrigin auto PO>
    requires(!Symbol.empty()) && (QS.dimension == get-associated-quantity(U).dimension)
  struct named_unit<Symbol, U, QS, PO> : decltype(U)::base-type {
    using base-type = named_unit;              // exposition only
    static constexpr auto symbol = Symbol;     // exposition only
    static constexpr auto quantity-spec = QS;  // exposition only
    static constexpr auto point-origin = PO;   // exposition only
  };

  }
```

1 A *named unit* is a type that models `PrefixableUnit`. A specialization of `named_unit` is used as a base type when defining a named unit.

2 In the following descriptions, let `U` be a named unit defined with an alluded signature. The identifier of `U` represents its unit name (IEC 60050, 112-01-15) or special unit name (IEC 60050, 112-01-16). `Symbol` is its unit symbol (IEC 60050, 112-01-17).

3 The possible arguments to `named_unit` are

(3.1) — (the unit symbol, a kind of base quantity, a point origin$_{opt}$),

(3.2) — (the unit symbol),

(3.3) — (the unit symbol, a unit expression, a point origin$_{opt}$), and

(3.4) — (the unit symbol, a unit expression, a kind of quantity, a point origin$_{opt}$).

4 The first signature defines the unit of a base quantity without a unit prefix (IEC 60050, 112-01-26). The second signature defines a unit that can be reused by several base quantities. The third and fourth signatures with a unit expression argument E define `U` as implicitly convertible from E. The first and fourth signatures with a kind of quantity (IEC 60050, 112-01-04) Q also restrict `U` to Q.

5 A point origin argument specifies the default point origin of `U` (5.7.4).

6 [*Example 1*:

```
// The first signature defines a base unit restricted to a kind of base quantity.
inline constexpr struct second final : named_unit<"s", kind_of<time>> {
} second;

// The third and fourth signatures give a name to the unit argument.
inline constexpr struct minute final : named_unit<"min", mag<60> * second> {
} minute;  // min = 60 s.

// The fourth signature also further restricts the kind of quantity.
inline constexpr struct hertz final : named_unit<"Hz", inverse(second), kind_of<frequency>> {
} hertz;  // Hz can't measure becquerel, activity,
          // or any other quantity with dimension T^-1
          // that isn't a kind of frequency.
```

— *end example*]

5.4.4.5.4 Prefixed [qty.prefixed.unit]

```
namespace mp_units {

template<symbol_text Symbol, UnitMagnitude auto M, PrefixableUnit auto U>
  requires(!Symbol.empty())
struct prefixed_unit : decltype(M * U)::base-type {
  using base-type = prefixed_unit;                     // exposition only
  static constexpr auto symbol = Symbol + U.symbol;   // exposition only
};

}
```

1 `prefixed_unit<Symbol, M, U>` represents the unit `U` with a unit prefix (IEC 60050, 112-01-26). `Symbol` is the symbol of the unit prefix. `M` is the factor of the unit prefix. A specialization of `prefixed_unit` is used as a base type when defining a unit prefix.

[*Example 1*:

```
template<PrefixableUnit auto U>
struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U> {};

template<PrefixableUnit auto U>
constexpr kilo_<U> kilo;

inline constexpr auto kilogram = kilo<si::gram>;
```

— *end example*]

5.4.4.5.5 Common [qty.common.unit]

```
namespace mp_units {

template<Unit U1, Unit U2, Unit... Rest>
struct common_unit final : decltype(get-common-scaled-unit(U1{}, U2{}, Rest{}...))::base-type
{
  using base-type = common_unit;        // exposition only
  static constexpr auto common-unit =   // exposition only
    get-common-scaled-unit(U1{}, U2{}, Rest{}...);
};

}
```

1 `common_unit` is used by the library to encapsulate a conversion factor between units (IEC 60050, 112-01-33) common to the operands of quantity addition.

[*Example 1*: The result of `1 * km + 1 * mi` has a common unit [8/125] m encapsulated by `common_unit<mi, km>`. — *end example*]

A program that instantiates a specialization of `common_unit` that is not a possible result of the library specifications is ill-formed, no diagnostic required.

```
template<Unit U1, Unit U2, Unit... Rest>
consteval Unit auto get-common-scaled-unit(U1, U2, Rest... rest)  // exposition only
  requires see below;
```

2 *Effects*: Equivalent to:

```
constexpr auto res = [] {
  constexpr auto canonical_lhs = get-canonical-unit(U1{});
  constexpr auto canonical_rhs = get-canonical-unit(U2{});
  constexpr auto common_mag = common-magnitude(canonical_lhs.mag, canonical_rhs.mag);
  if constexpr (common_mag == mag<1>)
    return canonical_lhs.reference_unit;
  else
    return scaled_unit<common_mag, decltype(auto(canonical_lhs.reference_unit))>{};
}();
if constexpr (sizeof...(rest) == 0)
  return res;
else
  return get-common-scaled-unit(res, rest...);
```

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(convertible(U1{}, U2{}) && (sizeof...(Rest) == 0 || requires {
   get-common-scaled-unit(get-common-scaled-unit(u1, u2), rest...);
 }))
```

5.4.4.5.6 Derived [qty.derived.unit]

```
namespace mp_units {

template<typename... Expr>
struct derived-unit-impl :              // exposition only
    unit-interface,
    expr-fractions<struct one, Expr...> {
  using base-type = derived-unit-impl;  // exposition only
};

template<SymbolicConstant... Expr>
struct derived_unit final : derived-unit-impl<Expr...> {};

}
```

1 `derived_unit` is used by the library to represent a derived unit (IEC 60050, 112-01-19).

[*Example 1*:

```
using namespace si::unit_symbols;
int x = m * m;  // error: cannot construct from derived_unit<power<si::metre, 2>>
int y = m * s;  // error: cannot construct from derived_unit<si::metre, si::second>
int z = m / s;  // error: cannot construct from derived_unit<si::metre, per<si::second>>
```

— *end example*]

A program that instantiates a specialization of `derived_unit` that is not a possible result of the library specifications is ill-formed, no diagnostic required.

5.4.4.5.7 One [qty.unit.one]

```
namespace mp_units {

struct one final : derived-unit-impl<> {};

}
```

1 `one` represents the base unit (IEC 60050, 112-01-18) of a quantity of dimension one (IEC 60050, 112-01-13).

5.4.4.6 Operations [qty.unit.ops]

```
namespace mp_units {
  struct unit-interface {   // exposition only
    template<UnitMagnitude M, Unit U>
    friend consteval Unit auto operator*(M, U u);

    friend consteval Unit auto operator*(Unit auto, UnitMagnitude auto) = delete;

    template<UnitMagnitude M, Unit U>
    friend consteval Unit auto operator/(M mag, U u);

    template<Unit Lhs, Unit Rhs>
    friend consteval Unit auto operator*(Lhs lhs, Rhs rhs);

    template<Unit Lhs, Unit Rhs>
    friend consteval Unit auto operator/(Lhs lhs, Rhs rhs);

    // 5.4.4.7, comparison

    template<Unit Lhs, Unit Rhs>
    friend consteval bool operator==(Lhs, Rhs);

    template<Unit Lhs, Unit Rhs>
    friend consteval bool equivalent(Lhs lhs, Rhs rhs);
  };

  }
```

```
template<UnitMagnitude M, Unit U>
friend consteval Unit auto operator*(M, U u);
```

1 *Effects*: Equivalent to:

```
if constexpr (std::is_same_v<M, decltype(auto(mag<1>))>)
  return u;
else if constexpr (is-specialization-of<U, scaled_unit>()) {
  if constexpr (M{} * U::mag == mag<1>)
    return U::reference-unit;
  else
    return scaled_unit<M{} * U::mag, decltype(auto(U::reference-unit))>{};
} else
  return scaled_unit<M{}, U>{};
```

```
friend consteval Unit auto operator*(Unit auto, UnitMagnitude auto) = delete;
```

2 *Recommended practice*: Suggest swapping the operands.

```
template<UnitMagnitude M, Unit U>
friend consteval Unit auto operator/(M mag, U u);
```

3 *Returns*: `mag * inverse(u)`.

```
template<Unit Lhs, Unit Rhs>
friend consteval Unit auto operator*(Lhs lhs, Rhs rhs);
```

4 *Returns*: `expr-multiply<derived_unit, struct one>(lhs, rhs)`.

```
template<Unit Lhs, Unit Rhs>
friend consteval Unit auto operator/(Lhs lhs, Rhs rhs);
```

5 *Returns*: `expr-divide<derived_unit, struct one>(lhs, rhs)`.

```
consteval Unit auto inverse(Unit auto u);
```

6 *Returns*: `one / u`.

```
template<std::intmax_t Num, std::intmax_t Den = 1, Unit U>
  requires(Den != 0)
consteval Unit auto pow(U u);
```

7 *Returns*: `expr-pow<Num, Den, derived_unit, struct one>(u)`.

```
consteval Unit auto sqrt(Unit auto u);
```

8 *Returns*: `pow<1, 2>(u)`.

```
consteval Unit auto cbrt(Unit auto u);
```

9 *Returns*: `pow<1, 3>(u)`.

```
consteval Unit auto square(Unit auto u);
```

10 *Returns*: `pow<2>(u)`.

```
consteval Unit auto cubic(Unit auto u);
```

11 *Returns*: `pow<3>(u)`.

5.4.4.7 Comparison [qty.unit.cmp]

```
template<Unit Lhs, Unit Rhs>
friend consteval bool operator==(Lhs, Rhs);
```

1 *Returns*: `std::is_same_v<Lhs, Rhs>`.

```
template<Unit Lhs, Unit Rhs>
friend consteval bool equivalent(Lhs lhs, Rhs rhs);
```

2 *Effects*: Equivalent to:

```
const auto lhs_canonical = get-canonical-unit(lhs);
const auto rhs_canonical = get-canonical-unit(rhs);
return lhs_canonical.mag == rhs_canonical.mag &&
       lhs_canonical.reference_unit == rhs_canonical.reference_unit;
```

```
template<Unit From, Unit To>
consteval bool convertible(From from, To to);
```

3 *Effects*: Equivalent to:

```
if constexpr (std::is_same_v<From, To>)
  return true;
else if constexpr (PotentiallyConvertibleTo<From, To>)
  return std::is_same_v<decltype(get-canonical-unit(from).reference_unit),
                        decltype(get-canonical-unit(to).reference_unit)>;
else
  return false;
```

5.4.4.8 Observers [qty.unit.obs]

```
consteval QuantitySpec auto get_quantity_spec(AssociatedUnit auto u);
```

1 *Returns*: `kind_of<get-associated-quantity(u)>`.

```
consteval Unit auto get_unit(AssociatedUnit auto u);
```

2 *Returns*: `u`.

```
consteval Unit auto get_common_unit(Unit auto... us)
  requires see below;
```

3 Let

(3.1) — `u1` be `us...[0]`,

(3.2) — `u2` be `us...[1]`,

(3.3) — `U1` be `decltype(u1)`,

(3.4) — `U2` be `decltype(u2)`, and

(3.5) — `rest` be a pack denoting the elements of `us` without `u1` and `u2`.

4 *Effects*: Equivalent to:

```
if constexpr (sizeof...(us) == 1)
  return u1;
```

```
    else if constexpr (sizeof...(us) == 2) {
      if constexpr (is-derived-from-specialization-of<U1, common_unit>()) {
        return TBD.;
      } else if constexpr (is-derived-from-specialization-of<U2, common_unit>())
        return get_common_unit(u2, u1);
      else if constexpr (std::is_same_v<U1, U2>)
        return u1;
      else if constexpr (equivalent(U1{}, U2{})) {
        if constexpr (std::derived_from<U1, typename U2::base-type>)
          return u1;
        else if constexpr (std::derived_from<U2, typename U1::base-type>)
          return u2;
        else
          return std::conditional_t<type-less-impl<U1, U2>(), U1, U2>{};
      } else {
        constexpr auto canonical_lhs = get-canonical-unit(U1{});
        constexpr auto canonical_rhs = get-canonical-unit(U2{});

        if constexpr (is-positive-integral-power(canonical_lhs.mag / canonical_rhs.mag))
          return u2;
        else if constexpr (is-positive-integral-power(canonical_rhs.mag / canonical_lhs.mag))
          return u1;
        else {
          if constexpr (type-less<U1, U2>{})
            return common_unit<U1, U2>{};
          else
            return common_unit<U2, U1>{};
        }
      }
    } else
      return get_common_unit(get_common_unit(u1, u2), rest...);
```

5 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(sizeof...(us) != 0 && (sizeof...(us) == 1 ||  //
                        (sizeof...(us) == 2 && convertible(U1{}, U2{})) ||
                        requires { get_common_unit(get_common_unit(u1, u2), rest...); }))
```

5.4.4.9 Associated quantity [assoc.qty]

```
template<Unit U>
consteval bool has-associated-quantity(U);   // exposition only
```

1 *Returns*:

(1.1) — If `U::quantity-spec` is a valid expression, returns `true`.

(1.2) — Otherwise, if `U::reference-unit` is a valid expression, returns

```
has-associated-quantity(U::reference-unit)
```

(1.3) — Otherwise, if `is-derived-from-specialization-of<U, expr-fractions>()` is `true`, let `Nums` and `Dens` be packs denoting the template arguments of `U::nums` and `U::dens`, respectively. Returns

```
(... && has-associated-quantity(expr-type<Nums>{})) &&
  (... && has-associated-quantity(expr-type<Dens>{}))
```

(1.4) — Otherwise, returns `false`.

```
template<AssociatedUnit U>
consteval auto get-associated-quantity(U u);   // exposition only
```

2 *Returns*:

(2.1) — If `U` is of the form `common_unit<Us...>`, returns

```
get_common_quantity_spec(get-associated-quantity(Us{})...)
```

(2.2) — Otherwise, if `U::quantity-spec` is a valid expression, returns

```
remove-kind(U::quantity-spec)
```

(2.3) — Otherwise, if `U::reference-unit` is a valid expression, returns

```
get-associated-quantity(U::reference-unit)
```

(2.4) — Otherwise, if `is-derived-from-specialization-of<U, expr-fractions>()` is `true`, returns

```
expr-map<to-quantity-spec, derived_quantity_spec, struct dimensionless>(u)
```

where `to-quantity-spec` is defined as follows:

```
template<AssociatedUnit U>
using to-quantity-spec = decltype(get-associated-quantity(U{}));  // exposition only
```

5.4.4.10 Symbol formatting [qty.unit.sym.fmt]

```
template<typename CharT, std::output_iterator<CharT> Out>
constexpr Out copy-separator(Out out, const unit_symbol_formatting& fmt);  // exposition only
```

1 *Effects*: Equivalent to:

```
if (fmt.separator == unit_symbol_separator::half_high_dot) {
  const std::string_view dot = "\u22C5" /* U+22C5 DOT OPERATOR */;
  out = std::ranges::copy(dot, out);
} else {
  *out++ = ' ';
}
return out;
```

2 *Throws*: `std::invalid_argument` if `fmt.separator == unit_symbol_separator::half_high_dot && fmt.char_set != character_set::utf8` is `true`.

[*Note 1*: `unit_symbol_separator::half_high_dot` can be only used with `character_set::utf8`. — *end note*]

```
template<typename... Us, Unit U>
consteval Unit auto get-common-unit-in(common_unit<Us...>, U u);  // exposition only
```

3 *Effects*: Equivalent to:

```
auto get_magnitude = [&]() {
  if constexpr (requires { common_unit<Us...>::mag; })
    return common_unit<Us...>::mag;
  else
    return mag<1>;
};
constexpr auto canonical_u = get_canonical_unit(u);
constexpr UnitMagnitude auto cmag = get_magnitude() / canonical_u.mag;
if constexpr (cmag == mag<1>)
  return u;
else
  return scaled_unit<cmag, U>{};
```

```
template<typename CharT = char, std::output_iterator<CharT> Out, Unit U>
constexpr Out unit_symbol_to(Out out, U u, const unit_symbol_formatting& fmt = {});
```

4 Let `E` be the type of the unit whose symbol text is being copied.

5 *Effects*: Copies the symbol text of the unit `U` to `out` according to `fmt` as follows:

(5.1) — If `E` is of the form `common_unit<First, Tail...>`, then:

(5.1.1) — Let `copy_unit(x)` be equivalent to copying the symbol text of

```
get-common-unit-in(common_unit<First, Tail...>{}, x)
```

as follows.

(5.1.2) — Equivalent to:

```
out = std::ranges::copy(std::string_view("EQUIV{"), out);
copy_unit(First{});
(...,
  (out = std::ranges::copy(std::string_view(", "), out), copy_unit(Tail{})));
*out++ = '}';
```

(5.2) — If `E` is of the form `scaled_unit<M, V>`, then:

(5.2.1) — First, equivalent to:

```
*out++ = '(';
out = magnitude-symbol<CharT>(out, M, fmt);  // see 5.4.4.2.5
if constexpr (space_before_unit_symbol<scaled_unit<M, V>::reference-unit>)
  *out++ = ' ';
```

(5.2.2) — Then, copies the symbol text of `E::`*`reference-unit`* as follows.

(5.2.3) — Then, equivalent to `*out++ = ')'`.

(5.3) — If *`is-derived-from-specialization-of`*`<U,` *`expr-fractions`*`>()` is `true`, let `Nums` and `Dens` be packs denoting the template arguments of `U::`*`nums`* and `U::`*`dens`*, respectively.

(5.3.1) — If `sizeof...(Nums) == 0 && sizeof...(Dens) == 0` is `true`, no effect.

(5.3.2) — Otherwise, performs the following algorithm. To intersperse separators between steps is to evaluate an expression equivalent to *`copy-separator`*`<CharT>(out, fmt)`. First, copies the symbol text of the numerators `(..., void(Nums))`, in that order, as follows, with separators interspersed in-between. Then, equivalent to:

```
using enum unit_symbol_solidus;
if (fmt.solidus == always ||
    (fmt.solidus == one_denominator && sizeof...(Dens) == 1)) {
  if constexpr (sizeof...(Nums) == 0) *out++ = '1';
  *out++ = '/';
  if (sizeof...(Dens) > 1) *out++ = '(';
} else if constexpr (sizeof...(Nums) > 0) {
  out = copy-separator<CharT>(out, fmt);
}
```

Then, copies the symbol text of the denominators `(..., void(Dens))`, in that order, as follows, with separators interspersed in-between. Then, equivalent to:

```
using enum unit_symbol_solidus;
if (fmt.solidus == always && sizeof...(Dens) > 1) *out++ = ')';
```

(5.4) — If `E` is a specialization of `power`, copies the symbol text of `E::`*`factor`*, as follows. Then, copies the symbol text of `E::`*`exponent`* $\neq 1$, determined as follows:

(5.4.1) — If `E` is part of a denominator, let `sign` be `"-"`. Otherwise, let `sign` be `""`.

(5.4.2) — Let `txt` be equivalent to *`get-symbol-exponent`*`(sign, E::`*`exponent`*`)` (see 5.4.2.5).

(5.5) — If `E::`*`symbol`* is a valid expression, let `txt` be that.

(5.6) — Each time `txt` is determined, equivalent to:

```
out = copy-symbol-text(txt, fmt.char_set, out)  // see 5.4.2.5
```

6 *Returns*: `out`.

```
template<unit_symbol_formatting fmt = {}, typename CharT = char, Unit U>
consteval std::string_view unit_symbol(U);
```

7 *Returns*: A value `sv` such that [`sv.data()`, `sv.data() + sv.size()`) has static storage duration and the following assertion holds:

```
std::basic_string<CharT> s;
unit_symbol_to<CharT>(std::back_inserter(s), U{}, fmt);
assert(sv == s);
```

8 [*Example 1*:

```
import mp_units;

using namespace mp_units;
using enum character_set;
using enum unit_symbol_solidus;
using enum unit_symbol_separator;

static_assert(unit_symbol(metre) == "m");
static_assert(unit_symbol(degree_Celsius) == "\u2103");
```

```
    static_assert(unit_symbol<{.char_set = portable}>(degree_Celsius) == "`C");
    static_assert(unit_symbol(kilogram) == "kg");

    static_assert(unit_symbol(mag<100> * metre) == "(100 m)");
    static_assert(unit_symbol(mag<1000> * metre) == "(10\u00B3 m)");
    static_assert(unit_symbol<{.char_set = portable}>(mag<1000> * metre) == "(10^3 m)");
    static_assert(unit_symbol(mag<6000> * metre) == "(6 \u00D7 10\u00B3 m)");
    static_assert(unit_symbol(mag<10600> * metre) == "(10600 m)");
    static_assert(unit_symbol(mag_ratio<1, 18> * metre / second) == "(1/18 m)/s");
    static_assert(unit_symbol(mag_ratio<1, 18> * (metre / second)) == "(1/18 m/s)");

    static_assert(unit_symbol(mag<\u03C0> * one) == "(\u03C0)");
    static_assert(unit_symbol(mag<\u03C0> * metre) == "(\u03C0 m)");
    static_assert(unit_symbol(mag<2> * mag<\u03C0> * metre) == "(2 \u03C0 m)");
    static_assert(unit_symbol<{.separator = half_high_dot}>(mag<2> * mag<\u03C0> * metre) ==
           "(2\u22C5\u03C0 m)");
    static_assert(unit_symbol(mag<1> / mag<\u03C0> * one) == "(1/\u03C0)");
    static_assert(unit_symbol<usf{.solidus = never}>(mag<1> / mag<\u03C0> * one) ==
           "(\u03C0\u207B\u00B9)");
    static_assert(unit_symbol<usf{.char_set = portable, .solidus = never}>(
            mag<1> / mag<\u03C0> * one) == "(pi^-1)");

    static_assert(unit_symbol(mag<1> / (mag<2> * mag<\u03C0>)*metre) ==
           "(2\u207B\u00B9 \u03C0\u207B\u00B9 m)");
    static_assert(unit_symbol<usf{.solidus = always}>(mag<1> / (mag<2> * mag<\u03C0>)*metre) ==
           "(1/(2 \u03C0) m)");
    static_assert(unit_symbol(mag_ratio<1, 2> * mag<\u03C0> * metre) == "(\u03C0/2 m)");
    static_assert(unit_symbol(mag_power<\u03C0, 2> * one) == "(\u03C0\u00B2)");
    static_assert(unit_symbol(mag_power<\u03C0, 1, 2> * one) == "(\u03C0^(1/2))");

    static_assert(unit_symbol(get_common_unit(kilo<metre>, mile)) ==
           "EQUIV{(1/25146 mi), (1/15625 km)}");

    static_assert(unit_symbol(kilo<metre> * metre) == "km m");
    static_assert(unit_symbol<usf{.separator = half_high_dot}>(kilo<metre> * metre) ==
           "km\u22C5m");
    static_assert(unit_symbol(metre / metre) == "");
    static_assert(unit_symbol(kilo<metre> / metre) == "km/m");
    static_assert(unit_symbol<usf{.solidus = never}>(kilo<metre> / metre) ==
           "km m\u207B\u00B9");
    static_assert(unit_symbol<usf{.char_set = portable, .solidus = never}>(kilo<metre> /
                                       metre) == "km m^-1");
    static_assert(unit_symbol(metre / second) == "m/s");
    static_assert(unit_symbol<usf{.solidus = never}>(metre / second) == "m s\u207B\u00B9");
    static_assert(unit_symbol<usf{.solidus = never, .separator = half_high_dot}>(metre /
                                          second) ==
           "m\u22C5s\u207B\u00B9");
    static_assert(unit_symbol<usf{.solidus = never, .separator = half_high_dot}>(
            kilogram * metre / square(second)) == "kg\u22C5m\u22C5s\u207B\u00B2");
    static_assert(unit_symbol<usf{.solidus = always}>(one / metre / square(second)) ==
           "1/(m s\u00B2)");
    static_assert(unit_symbol(pow<1, 2>(metre / second)) == "m^(1/2)/s^(1/2)");
    static_assert(unit_symbol<usf{.solidus = never}>(pow<1, 2>(metre / second)) ==
           "m^(1/2) s^-(1/2)");
    static_assert(unit_symbol(litre / (mag<100> * kilo<metre>)) == "L/(100 km)");
    static_assert(unit_symbol((mag<10> * metre) / (mag<20> * second)) == "(10 m)/(20 s)");
    static_assert(unit_symbol(pow<2>(mag<3600> * second)) == "(3600 s)\u00B2");
```

Some of the assertions depend on the implementation-defined total order of types. — *end example*]

5.4.5 Concepts [qty.ref.concepts]

```
template<typename T>
concept Reference = AssociatedUnit<T> || (is-specialization-of<T, reference>());
```

A type `T` that satisfies `Reference` represents the reference of a quantity (IEC 60050, 112-01-01).

```
template<typename T, auto QS>
concept ReferenceOf = Reference<T> && QuantitySpecOf<decltype(get_quantity_spec(T{})), QS>;
```

5.4.6 Class template reference [qty.ref.syn]

```
namespace mp_units {

template<QuantitySpec auto Q, Unit auto U>
using reference-t = reference<decltype(Q), decltype(U)>;  // exposition only

template<QuantitySpec Q, Unit U>
struct reference {
  // 5.4.7, operations

  template<typename Q2, typename U2>
  friend consteval auto operator*(reference, reference<Q2, U2>)
    -> reference-t<Q{} * Q2{}, U{} * U2{}>;

  template<AssociatedUnit U2>
  friend consteval auto operator*(reference, U2)
    -> reference-t<Q{} * get_quantity_spec(U2{}), U{} * U2{}>;

  template<AssociatedUnit U1>
  friend consteval auto operator*(U1, reference)
    -> reference-t<get_quantity_spec(U1{}) * Q{}, U1{} * U{}>;

  template<typename Q2, typename U2>
  friend consteval auto operator/(reference, reference<Q2, U2>)
    -> reference-t<Q{} / Q2{}, U{} / U2{}>;

  template<AssociatedUnit U2>
  friend consteval auto operator/(reference, U2)
    -> reference-t<Q{} / get_quantity_spec(U2{}), U{} / U2{}>;

  template<AssociatedUnit U1>
  friend consteval auto operator/(U1, reference)
    -> reference-t<get_quantity_spec(U1{}) / Q{}, U1{} / U{}>;

  friend consteval auto inverse(reference) -> reference-t<inverse(Q{}), inverse(U{})>;

  template<std::intmax_t Num, std::intmax_t Den = 1>
    requires(Den != 0)
  friend consteval auto pow(reference) -> reference-t<pow<Num, Den>(Q{}), pow<Num, Den>(U{})>;
  friend consteval auto sqrt(reference) -> reference-t<sqrt(Q{}), sqrt(U{})>;
  friend consteval auto cbrt(reference) -> reference-t<cbrt(Q{}), cbrt(U{})>;

  // 5.4.8, comparison

  template<typename Q2, typename U2>
  friend consteval bool operator==(reference, reference<Q2, U2>);

  template<AssociatedUnit U2>
  friend consteval bool operator==(reference, U2 u2);

  template<typename Q2, typename U2>
  friend consteval bool convertible(reference, reference<Q2, U2>);

  template<AssociatedUnit U2>
  friend consteval bool convertible(reference, U2 u2);

  template<AssociatedUnit U1>
  friend consteval bool convertible(U1 u1, reference);
};
}
```

1 `reference<Q, U>` represents the reference of a quantity (IEC 60050, 112-01-01). The unit of measurement `U` (IEC 60050, 112-01-14) is used to measure a value of the quantity `Q` (IEC 60050, 112-01-28).

[*Note 1*: `reference` is typically implicitly instantiated when specifying that a unit measures a more specific quantity.

[*Example 1*:

```
using namespace si::unit_symbols;
auto x = 1 * m;                  // measures a length
auto y = 1 * isq::width[m];      // measures a width
auto z = 1 * isq::diameter[m];   // measures a diameter
```

— *end example*]

— *end note*]

5.4.7 Operations [qty.ref.ops]

1 Each member function with a *trailing-return-type* of `-> reference-t<T...>` returns `{}`.

```
template<typename FwdRep, Reference R,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  requires(!OffsetUnit<decltype(get_unit(R{}))>)
constexpr quantity<R{}, Rep> operator*(FwdRep&& lhs, R r);
```

2 *Effects*: Equivalent to: `return quantity{std::forward<FwdRep>(lhs), r};`

```
template<typename FwdRep, Reference R,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  requires(!OffsetUnit<decltype(get_unit(R{}))>)
constexpr Quantity auto operator/(FwdRep&& lhs, R);
```

3 *Effects*: Equivalent to: `return quantity{std::forward<FwdRep>(lhs), inverse(R{})};`

```
template<typename FwdQ, Reference R, Quantity Q = std::remove_cvref_t<FwdQ>>
constexpr Quantity auto operator*(FwdQ&& q, R);
template<typename FwdQ, Reference R, Quantity Q = std::remove_cvref_t<FwdQ>>
constexpr Quantity auto operator/(FwdQ&& q, R);
```

4 Let `@` be the *operator*.

5 *Effects*: Equivalent to:

```
return quantity{std::forward<FwdQ>(q).numerical-value, Q::reference @ R{}};
```

```
template<Reference R, typename Rep>
  requires RepresentationOf<std::remove_cvref_t<Rep>, get_quantity_spec(R{})>
constexpr auto operator*(R, Rep&&) = delete;

template<Reference R, typename Rep>
  requires RepresentationOf<std::remove_cvref_t<Rep>, get_quantity_spec(R{})>
constexpr auto operator/(R, Rep&&) = delete;

template<Reference R, typename Q>
  requires Quantity<std::remove_cvref_t<Q>>
constexpr auto operator*(R, Q&&) = delete;

template<Reference R, typename Q>
  requires Quantity<std::remove_cvref_t<Q>>
constexpr auto operator/(R, Q&&) = delete;
```

6 *Recommended practice*: Suggest swapping the operands.

5.4.8 Comparison [qty.ref.cmp]

```
template<typename Q2, typename U2>
friend consteval bool operator==(reference, reference<Q2, U2>);
```

1 *Returns*: `Q{} == Q2{} && U{} == U2{}`.

```
template<AssociatedUnit U2>
friend consteval bool operator==(reference, U2 u2);
```

2 *Returns*: `Q{} == get_quantity_spec(u2) && U{} == u2`.

```
template<typename Q2, typename U2>
friend consteval bool convertible(reference, reference<Q2, U2>);
```

3 *Returns*: `implicitly_convertible(Q{}, Q2{}) && convertible(U{}, U2{})`.

```
template<AssociatedUnit U2>
friend consteval bool convertible(reference, U2 u2);
```

4 *Returns*: `implicitly_convertible(Q{}, get_quantity_spec(u2)) && convertible(U{}, u2)`.

```
template<AssociatedUnit U1>
friend consteval bool convertible(U1 u1, reference);
```

5 *Returns*: `implicitly_convertible(get_quantity_spec(u1), Q{}) && convertible(u1, U{})`.

5.4.9 Observers [qty.ref.obs]

```
template<typename Q, typename U>
consteval QuantitySpec auto get_quantity_spec(reference<Q, U>);
```

1 *Returns*: `Q{}`.

```
template<typename Q, typename U>
consteval Unit auto get_unit(reference<Q, U>);
```

2 *Returns*: `U{}`.

```
consteval AssociatedUnit auto get_common_reference(AssociatedUnit auto u1,
                                                   AssociatedUnit auto u2,
                                                   AssociatedUnit auto... rest)
  requires see below;
```

3 *Returns*: `get_common_unit(u1, u2, rest...)`.

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires {
  get_common_quantity_spec(get_quantity_spec(u1), get_quantity_spec(u2),
                           get_quantity_spec(rest)...);
  { get_common_unit(u1, u2, rest...) } -> AssociatedUnit;
}
```

```
template<Reference R1, Reference R2, Reference... Rest>
consteval Reference auto get_common_reference(R1 r1, R2 r2, Rest... rest)
  requires see below;
```

5 *Returns*:

```
reference-t<get_common_quantity_spec(get_quantity_spec(R1{}), get_quantity_spec(R2{}),
                                     get_quantity_spec(rest)...),
            get_common_unit(get_unit(R1{}), get_unit(R2{}), get_unit(rest)...)>{};
```

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires {
  get_common_quantity_spec(get_quantity_spec(r1), get_quantity_spec(r2),
                           get_quantity_spec(rest)...);
  get_common_unit(get_unit(r1), get_unit(r2), get_unit(rest)...);
}
```

5.4.10 Utilities [qty.ref.utils]

```
template<AssociatedUnit auto To, AssociatedUnit From>
consteval decltype(To) clone-reference-with(From);
template<Unit auto To, QuantitySpec QS, Unit U>
consteval reference<QS, decltype(To)> clone-reference-with(reference<QS, U>);
```

1 *Returns*: `{}`.

5.4.11 Math [qty.ref.math]

```
template<Representation Rep, Reference R>
  requires requires { std::numeric_limits<Rep>::epsilon(); }
constexpr quantity<R{}, Rep> epsilon(R r) noexcept;                                    // hosted
```

1 *Effects*: Equivalent to: `return {static_cast<Rep>(std::numeric_limits<Rep>::epsilon()), r};`

5.5 Representation [qty.rep]

5.5.1 General [qty.rep.general]

1 Subclause 5.5 specifies the components used to constrain the numerical value of a quantity (IEC 60050, 112-01-29).

5.5.2 Traits [qty.rep.traits]

5.5.2.1 Floating-point [qty.fp.traits]

```
template<typename T>
struct actual-value-type : cond-value-type<T> {}; // see N4971, [readable.traits]

template<typename T>
  requires(!std::is_pointer_v<T> && !std::is_array_v<T>) &&
          requires { typename std::indirectly_readable_traits<T>::value_type; }
struct actual-value-type<T> : std::indirectly_readable_traits<T> {};

template<typename T>
using actual-value-type-t = actual-value-type<T>::value_type;

template<typename Rep>
constexpr bool treat_as_floating_point =
  std::chrono::treat_as_floating_point_v<actual-value-type-t<Rep>>;
```

1 `quantity` and `quantity_point` use `treat_as_floating_point` to help determine whether implicit conversions are allowed among them.

2 *Remarks*: Pursuant to N4971, [namespace.std] (4.1), users may specialize `treat_as_floating_point` for cv-unqualified program-defined types. Such specializations shall be usable in constant expressions (N4971, [expr.const]) and have type `const bool`.

5.5.2.2 Quantity character [qty.char.traits]

```
template<typename T>
constexpr bool disable_scalar = false;
template<typename T>
constexpr bool disable_complex = false;
template<typename T>
constexpr bool disable_vector = false;
```

1 Some quantities are defined as having a numerical value (IEC 60050, 112-01-29) of a specific set (IEC 60050, 102-01-02). The representation concepts use these traits to help determine the sets `T` represents.

2 *Remarks*: Pursuant to N4971, [namespace.std] (4.1), users may specialize these templates for cv-unqualified program-defined types. Such specializations shall be usable in constant expressions (N4971, [expr.const]) and have type `const bool`.

3 [*Note 1*: These templates prevent use of representation types with the library that satisfy but do not in fact model their corresponding concept. — *end note*]

5.5.2.3 Values [qty.val.traits]

1 `quantity` and `quantity_point` use `representation_values` to construct special values of its representation type.

```
namespace mp_units {

template<typename Rep>
struct representation_values : std::chrono::duration_values<Rep> {
  static constexpr Rep one() noexcept;
};
```

```
  }
```

2 The requirements on `std::chrono::duration_values<Rep>` (N4971, [time.traits.duration.values]) also apply to `representation_values<Rep>`.

```
static constexpr Rep one() noexcept;
```

3 *Returns*: `Rep(1)`.

4 *Remarks*: The value returned shall be the neutral element for multiplication (IEC 60050, 102-01-19).

5.5.3 Customization point objects [qty.rep.cpos]

5.5.3.1 General [qty.rep.cpos.general]

1 Within subclause 5.5.3, *reified object* is as defined in N4971, [range.access.general].

5.5.3.2 `mp_units::real` [qty.real.cpo]

1 The name `mp_units::real` denotes a customization point object (N4971, [customization.point.object]).

2 Given a subexpression `E` with type `T`, let `t` be an lvalue that denotes the reified object for `E`. Then:

(2.1) — If `T` does not model *`WeaklyRegular`*, `mp_units::real(E)` is ill-formed.

(2.2) — If `auto(t.real())` is a valid expression whose type models *`Scalar`*, `mp_units::real(E)` is expression-equivalent to `auto(t.real())`.

(2.3) — Otherwise, if `T` is a class or enumeration type and `auto(real(t))` is a valid expression whose type models *`Scalar`* where the meaning of `real` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::real(E)` is expression-equivalent to that expression.

(2.4) — Otherwise, `mp_units::real(E)` is ill-formed.

5.5.3.3 `mp_units::imag` [qty.imag.cpo]

1 The name `mp_units::imag` denotes a customization point object (N4971, [customization.point.object]).

2 Given a subexpression `E` with type `T`, let `t` be an lvalue that denotes the reified object for `E`. Then:

(2.1) — If `T` does not model *`WeaklyRegular`*, `mp_units::imag(E)` is ill-formed.

(2.2) — If `auto(t.imag())` is a valid expression whose type models *`Scalar`*, `mp_units::imag(E)` is expression-equivalent to `auto(t.imag())`.

(2.3) — Otherwise, if `T` is a class or enumeration type and `auto(imag(t))` is a valid expression whose type models *`Scalar`* where the meaning of `imag` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::imag(E)` is expression-equivalent to that expression.

(2.4) — Otherwise, `mp_units::imag(E)` is ill-formed.

5.5.3.4 `mp_units::modulus` [qty.modulus.cpo]

1 The name `mp_units::modulus` denotes a customization point object (N4971, [customization.point.object]).

2 Given a subexpression `E` with type `T`, let `t` be an lvalue that denotes the reified object for `E`. Then:

(2.1) — If `T` does not model *`WeaklyRegular`*, `mp_units::modulus(E)` is ill-formed.

(2.2) — If `auto(t.modulus())` is a valid expression whose type models *`Scalar`*, `mp_units::modulus(E)` is expression-equivalent to `auto(t.modulus())`.

(2.3) — Otherwise, if `T` is a class or enumeration type and `auto(modulus(t))` is a valid expression whose type models *`Scalar`* where the meaning of `modulus` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::modulus(E)` is expression-equivalent to that expression.

(2.4) — If `auto(t.abs())` is a valid expression whose type models *`Scalar`*, `mp_units::modulus(E)` is expression-equivalent to `auto(t.abs())`.

(2.5) — Otherwise, if `T` is a class or enumeration type and `auto(abs(t))` is a valid expression whose type models *`Scalar`* where the meaning of `abs` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::modulus(E)` is expression-equivalent to that expression.

(2.6) — Otherwise, `mp_units::modulus(E)` is ill-formed.

5.5.3.5 `mp_units::magnitude` [qty.mag.cpo]

1 The name `mp_units::magnitude` denotes a customization point object (N4971, [customization.point.object]).

2 Given a subexpression `E` with type `T`, let `t` be an lvalue that denotes the reified object for `E`. Then:

(2.1) — If `T` does not model *`WeaklyRegular`*, `mp_units::magnitude(E)` is ill-formed.

(2.2) — If `auto(t.magnitude())` is a valid expression whose type models *`Scalar`*, `mp_units::magnitude(E)` is expression-equivalent to `auto(t.magnitude())`.

(2.3) — Otherwise, if `T` is a class or enumeration type and `auto(magnitude(t))` is a valid expression whose type models *`Scalar`* where the meaning of `magnitude` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::magnitude(E)` is expression-equivalent to that expression.

(2.4) — Otherwise, if `auto(t.abs())` is a valid expression whose type models *`Scalar`*, `mp_units::magnitude(E)` is expression-equivalent to `auto(t.abs())`.

(2.5) — Otherwise, if `T` is a class or enumeration type and `auto(abs(t))` is a valid expression whose type models *`Scalar`* where the meaning of `magnitude` is established as-if by performing argument-dependent lookup only (N4971, [basic.lookup.argdep]), then `mp_units::magnitude(E)` is expression-equivalent to that expression.

(2.6) — Otherwise, `mp_units::magnitude(E)` is ill-formed.

5.5.4 Concepts [qty.rep.concepts]

```
template<typename T>
concept WeaklyRegular = std::copyable<T> && std::equality_comparable<T>;  // exposition only

template<typename T>
concept Scalar = (!disable_scalar<T>) && WeaklyRegular<T> && requires(T a, T b) {  // exposition only
  // scalar operations
  { -a } -> std::common_with<T>;
  { a + b } -> std::common_with<T>;
  { a - b } -> std::common_with<T>;
  { a * b } -> std::common_with<T>;
  { a / b } -> std::common_with<T>;
};
```

1 TBD.

```
template<typename T>
using value-type-t = actual-value-type-t<T>;  // exposition only, see 5.5.2.1

template<typename T>
concept Complex =  // exposition only
  (!disable_complex<T>) && WeaklyRegular<T> && Scalar<value-type-t<T>> &&
  std::constructible_from<T, value-type-t<T>, value-type-t<T>> &&
  requires(T a, T b, value-type-t<T> s) {
    // complex operations
    { -a } -> std::common_with<T>;
    { a + b } -> std::common_with<T>;
    { a - b } -> std::common_with<T>;
    { a * b } -> std::common_with<T>;
    { a / b } -> std::common_with<T>;
    { a * s } -> std::common_with<T>;
    { s * a } -> std::common_with<T>;
    { a / s } -> std::common_with<T>;
    ::mp_units::real(a);
    ::mp_units::imag(a);
    ::mp_units::modulus(a);
  };
```

2 TBD.

```
template<typename T>
concept Vector =  // exposition only
  (!disable_vector<T>) && WeaklyRegular<T> && Scalar<value-type-t<T>> &&
  requires(T a, T b, value-type-t<T> s) {
    // vector operations
    { -a } -> std::common_with<T>;
    { a + b } -> std::common_with<T>;
    { a - b } -> std::common_with<T>;
    { a * s } -> std::common_with<T>;
    { s * a } -> std::common_with<T>;
    { a / s } -> std::common_with<T>;
    ::mp_units::magnitude(a);
  };
```

3 TBD.

```
template<typename T>
using scaling-factor-type-t =  // exposition only
  std::conditional_t<treat_as_floating_point<T>, long double, std::intmax_t>;

template<typename T>
concept ScalarRepresentation =  // exposition only
  (!is-specialization-of<T, quantity>()) && Scalar<T> &&
  requires(T a, T b, scaling-factor-type-t<T> f) {
    // scaling
    { a * f } -> std::common_with<T>;
    { f * a } -> std::common_with<T>;
    { a / f } -> std::common_with<T>;
  };
```

4 TBD.

```
template<typename T>
concept ComplexRepresentation =  // exposition only
  (!is-specialization-of<T, quantity>()) && Complex<T> &&
  requires(T a, T b, scaling-factor-type-t<T> f) {
    // scaling
    { a * T(f) } -> std::common_with<T>;
    { T(f) * a } -> std::common_with<T>;
    { a / T(f) } -> std::common_with<T>;
  };
```

5 TBD.

```
template<typename T>
concept VectorRepresentation =  // exposition only
  (!is-specialization-of<T, quantity>()) && Vector<T>;
```

6 TBD.

```
template<typename T>
concept Representation = ScalarRepresentation<T> || ComplexRepresentation<T> ||
                         VectorRepresentation<T>;
```

7 A type `T` models `Representation` if it represents the numerical value of a quantity (IEC 60050, 112-01-29).

```
template<typename T, quantity_character Ch>
concept IsOfCharacter = (Ch == quantity_character::scalar && Scalar<T>) ||  // exposition only
                        (Ch == quantity_character::complex && Complex<T>) ||
                        (Ch == quantity_character::vector && Vector<T>);

template<typename T, auto V>
concept RepresentationOf =
  Representation<T> && ((QuantitySpec<decltype(V)> &&
                         (QuantityKindSpec<decltype(V)> || IsOfCharacter<T, V.character>)) ||
                        (std::same_as<quantity_character, decltype(V)> && IsOfCharacter<T, V>));
```

8 A type `T` models `RepresentationOf<V>` if `T` models `Representation` and

(8.1) — `V` is a kind of quantity, or

(8.2) — if `V` is a quantity, then `T` represents a value of its character, or

(8.3) — if `V` is a quantity character, then `T` represents a value of `V`.

5.6 Quantity [qty]

5.6.1 General [qty.general]

1 Subclause 5.6 describes the class template `quantity` that represents the value of a quantity (IEC 60050, 112-01-28) that is an element of a vector space (IEC 60050, 102-03-01, IEC 60050, 102-03-04).

5.6.2 Interoperability [qty.like]

1 The interfaces specified in this subclause and subclause 5.7.3 are used by `quantity` and `quantity_point` to specify conversions with other types representing quantities.

[*Note 1*: 5.9 implements them for `std::chrono::duration` and `std::chrono::time_point`. — *end note*]

```
template<typename T, template<typename> typename Traits>
concept qty-like-impl = requires(const T& qty, const Traits<T>::rep& num) {  // exposition only
  { Traits<T>::to_numerical_value(qty) } -> std::same_as<typename Traits<T>::rep>;
  { Traits<T>::from_numerical_value(num) } -> std::same_as<T>;
  requires std::same_as<decltype(Traits<T>::explicit_import), const bool>;
  requires std::same_as<decltype(Traits<T>::explicit_export), const bool>;
  typename std::bool_constant<Traits<T>::explicit_import>;
  typename std::bool_constant<Traits<T>::explicit_export>;
};

template<typename T>
concept QuantityLike = !Quantity<T> && qty-like-impl<T, quantity_like_traits> && requires {
  typename quantity<quantity_like_traits<T>::reference, typename quantity_like_traits<T>::rep>;
};
```

2 In the following descriptions, let

(2.1) — `Traits` be `quantity_like_traits` or `quantity_point_like_traits`,

(2.2) — `Q` be a type for which `Traits<Q>` is specialized,

(2.3) — `qty` be an lvalue of type `const Q`, and

(2.4) — `num` be an lvalue of type `const Traits<Q>::rep`.

3 `Q` models *`qty-like-impl`*`<Traits>` if and only if:

(3.1) — `Traits<Q>::to_numerical_value(qty)` returns the numerical value (IEC 60050, 112-01-29) of `qty`.

(3.2) — `Traits<Q>::from_numerical_value(num)` returns a `Q` with numerical value `num`.

(3.3) — If `Traits` is `quantity_point_like_traits`, both numerical values are offset from `Traits<Q>::point_origin`.

4 If the following expression is `true`, the specified conversion will be explicit.

(4.1) — `Traits<Q>::explicit_import` for the conversion from `Q` to this library's type.

(4.2) — `Traits<Q>::explicit_export` for the conversion from this library's type to `Q`.

5.6.3 Class template quantity [qty.syn]

```
namespace mp_units {

template<typename T>
concept Quantity = (is-derived-from-specialization-of<T, quantity>());      // exposition only

template<typename Q, auto QS>
concept QuantityOf =                                                        // exposition only
  Quantity<Q> && QuantitySpecOf<decltype(auto(Q::quantity_spec)), QS>;
```

```
template<Unit UFrom, Unit UTo>
consteval bool integral-conversion-factor(UFrom from, UTo to)              // exposition only
{
  return is-integral(get-canonical-unit(from).mag / get-canonical-unit(to).mag);
}

template<typename T>
concept IsFloatingPoint = treat_as_floating_point<T>;                      // exposition only

template<typename FromRep, typename ToRep, auto FromUnit = one, auto ToUnit = one>
concept ValuePreservingTo =                                                // exposition only
  Representation<std::remove_cvref_t<FromRep>> && Representation<ToRep> &&
  Unit<decltype(FromUnit)> && Unit<decltype(ToUnit)> && std::assignable_from<ToRep&, FromRep> &&
  (IsFloatingPoint<ToRep> || (!IsFloatingPoint<std::remove_cvref_t<FromRep>> &&
                              (integral-conversion-factor(FromUnit, ToUnit))));

template<typename QFrom, typename QTo>
concept QuantityConvertibleTo =                                            // exposition only
  Quantity<QFrom> && Quantity<QTo> &&
  QuantitySpecConvertibleTo<QFrom::quantity_spec, QTo::quantity_spec> &&
  UnitConvertibleTo<QFrom::unit, QTo::unit> &&
  ValuePreservingTo<typename QFrom::rep, typename QTo::rep, QFrom::unit, QTo::unit> &&
  requires(QFrom q) { sudo-cast<QTo>(q); };                                // see 5.6.15

template<auto QS, typename Func, typename T, typename U>
concept InvokeResultOf =                                                   // exposition only
  QuantitySpec<decltype(QS)> && std::regular_invocable<Func, T, U> &&
  RepresentationOf<std::invoke_result_t<Func, T, U>, QS>;

template<typename Func, typename Q1, typename Q2,
         auto QS = std::invoke_result_t<Func, decltype(auto(Q1::quantity_spec)),
                                        decltype(auto(Q2::quantity_spec))>{}>
concept InvocableQuantities =                                              // exposition only
  QuantitySpec<decltype(QS)> && Quantity<Q1> && Quantity<Q2> &&
  InvokeResultOf<QS, Func, typename Q1::rep, typename Q2::rep>;

template<auto R1, auto R2>
concept HaveCommonReference = requires { get_common_reference(R1, R2); };  // exposition only

template<typename Func, Quantity Q1, Quantity Q2>
using common-quantity-for =                                                // exposition only
  quantity<get_common_reference(Q1::reference, Q2::reference),
           std::invoke_result_t<Func, typename Q1::rep, typename Q2::rep>>;

template<typename Func, typename Q1, typename Q2>
concept CommonlyInvocableQuantities =                                      // exposition only
  Quantity<Q1> && Quantity<Q2> && HaveCommonReference<Q1::reference, Q2::reference> &&
  std::convertible_to<Q1, common-quantity-for<Func, Q1, Q2>> &&
  std::convertible_to<Q2, common-quantity-for<Func, Q1, Q2>> &&
  InvocableQuantities<Func, Q1, Q2,
                      get_common_quantity_spec(Q1::quantity_spec, Q2::quantity_spec)>;

template<auto R1, auto R2, typename Rep1, typename Rep2>
concept SameValueAs =                                                      // exposition only
  (equivalent(get_unit(R1), get_unit(R2))) && std::convertible_to<Rep1, Rep2>;

template<typename T>
using quantity-like-type =                                                 // exposition only
  quantity<quantity_like_traits<T>::reference, typename quantity_like_traits<T>::rep>;

template<typename T, typename U, typename TT = std::remove_reference_t<T>>
concept Mutable = (!std::is_const_v<TT>) && std::derived_from<TT, U>;      // exposition only
```

```
template<Reference auto R, RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity {
public:
  Rep numerical-value;   // exposition only

  // member types and values
  static constexpr Reference auto reference = R;
  static constexpr QuantitySpec auto quantity_spec = get_quantity_spec(reference);
  static constexpr Dimension auto dimension = quantity_spec.dimension;
  static constexpr Unit auto unit = get_unit(reference);
  using rep = Rep;

  // 5.6.4, static member functions
  static constexpr quantity zero() noexcept
    requires see below;
  static constexpr quantity one() noexcept
    requires see below;
  static constexpr quantity min() noexcept
    requires see below;
  static constexpr quantity max() noexcept
    requires see below;

  // 5.6.5, constructors and assignment

  quantity() = default;
  quantity(const quantity&) = default;
  quantity(quantity&&) = default;
  ~quantity() = default;

  template<typename FwdValue, Reference R2>
    requires SameValueAs<R2{}, R, std::remove_cvref_t<FwdValue>, Rep>
  constexpr quantity(FwdValue&& v, R2);

  template<typename FwdValue, Reference R2, typename Value = std::remove_cvref_t<FwdValue>>
    requires(!SameValueAs<R2{}, R, Value, Rep>) &&
            QuantityConvertibleTo<quantity<R2{}, Value>, quantity>
  constexpr quantity(FwdValue&& v, R2);

  template<ValuePreservingTo<Rep> FwdValue>
    requires(unit == ::mp_units::one)
  constexpr quantity(FwdValue&& v);

  template<QuantityConvertibleTo<quantity> Q>
  constexpr explicit(see below) quantity(const Q& q);

  template<QuantityLike Q>
    requires QuantityConvertibleTo<quantity-like-type<Q>, quantity>
  constexpr explicit(see below) quantity(const Q& q);

  quantity& operator=(const quantity&) = default;
  quantity& operator=(quantity&&) = default;

  template<ValuePreservingTo<Rep> FwdValue>
    requires(unit == ::mp_units::one)
  constexpr quantity& operator=(FwdValue&& v);

  // 5.6.6, conversions

  template<UnitCompatibleWith<unit, quantity_spec> ToU>
    requires QuantityConvertibleTo<quantity, quantity<make-reference(quantity_spec, ToU{}), Rep>>
  constexpr QuantityOf<quantity_spec> auto in(ToU) const;
```

```
template<RepresentationOf<quantity_spec> ToRep>
  requires QuantityConvertibleTo<quantity, quantity<reference, ToRep>>
constexpr QuantityOf<quantity_spec> auto in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires QuantityConvertibleTo<quantity,
                                 quantity<make-reference(quantity_spec, ToU{}), ToRep>>
constexpr QuantityOf<quantity_spec> auto in(ToU) const;

template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires requires(const quantity q) { value_cast<ToU{}>(q); }
constexpr QuantityOf<quantity_spec> auto force_in(ToU) const;

template<RepresentationOf<quantity_spec> ToRep>
  requires requires(const quantity q) { value_cast<ToRep>(q); }
constexpr QuantityOf<quantity_spec> auto force_in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires requires(const quantity q) { value_cast<ToU{}, ToRep>(q); }
constexpr QuantityOf<quantity_spec> auto force_in(ToU) const;

// 5.6.7, numerical value observers

template<Unit U>
  requires(equivalent(U{}, unit))
constexpr rep& numerical_value_ref_in(U) & noexcept;
template<Unit U>
  requires(equivalent(U{}, unit))
constexpr const rep& numerical_value_ref_in(U) const & noexcept;
template<Unit U>
  requires(equivalent(U{}, unit))
void numerical_value_ref_in(U) const && = delete;

template<UnitCompatibleWith<unit, quantity_spec> U>
  requires QuantityConvertibleTo<quantity, quantity<make-reference(quantity_spec, U{}), Rep>>
constexpr rep numerical_value_in(U) const noexcept;

template<UnitCompatibleWith<unit, quantity_spec> U>
  requires requires(const quantity q) { value_cast<U{}>(q); }
constexpr rep force_numerical_value_in(U) const noexcept;

// 5.6.8, conversion operations

template<typename V_, std::constructible_from<Rep> Value = std::remove_cvref_t<V_>>
  requires(unit == ::mp_units::one)
explicit operator V_() const & noexcept;

template<typename Q_, QuantityLike Q = std::remove_cvref_t<Q_>>
  requires QuantityConvertibleTo<quantity, quantity-like-type<Q>>
constexpr explicit(see below) operator Q_() const noexcept(see below);

// 5.6.9, unary operations

constexpr QuantityOf<quantity_spec> auto operator+() const
  requires see below;
constexpr QuantityOf<quantity_spec> auto operator-() const
  requires see below;

template<Mutable<quantity> Q>
friend constexpr decltype(auto) operator++(Q&& q)
  requires see below;
```

```
template<Mutable<quantity> Q>
friend constexpr decltype(auto) operator--(Q&& q)
  requires see below;

constexpr QuantityOf<quantity_spec> auto operator++(int)
  requires see below;
constexpr QuantityOf<quantity_spec> auto operator--(int)
  requires see below;

// 5.6.10, compound assignment operations

template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator+=(Q&& lhs, const quantity<R2, Rep2>& rhs);
template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator-=(Q&& lhs, const quantity<R2, Rep2>& rhs);
template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator%=(Q&& lhs, const quantity<R2, Rep2>& rhs);

template<Mutable<quantity> Q, ValuePreservingTo<Rep> Value>
  requires see below
friend constexpr decltype(auto) operator*=(Q&& lhs, const Value& rhs);
template<Mutable<quantity> Q, ValuePreservingTo<Rep> Value>
  requires see below
friend constexpr decltype(auto) operator/=(Q&& lhs, const Value& rhs);

template<Mutable<quantity> Q, QuantityOf<dimensionless> Q2>
  requires see below
friend constexpr decltype(auto) operator*=(Q&& lhs, const Q2& rhs);
template<Mutable<quantity> Q, QuantityOf<dimensionless> Q2>
  requires see below
friend constexpr decltype(auto) operator/=(Q&& lhs, const Q2& rhs);

// 5.6.11, arithmetic operations

template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires CommonlyInvocableQuantities<std::plus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator+(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires CommonlyInvocableQuantities<std::minus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator-(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires(!treat_as_floating_point<Rep>) && (!treat_as_floating_point<Rep2>) &&
          CommonlyInvocableQuantities<std::modulus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator%(const Q& lhs, const quantity<R2, Rep2>& rhs);

template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::plus<>, Rep, const Value&>
friend constexpr Quantity auto operator+(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::minus<>, Rep, const Value&>
friend constexpr Quantity auto operator-(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::modulus<>, Rep, const Value&>
friend constexpr Quantity auto operator%(const Q& lhs, const Value& rhs);
```

```
  template<std::derived_from<quantity> Q, Representation Value>
    requires(Q::unit == ::mp_units::one) &&
            InvokeResultOf<quantity_spec, std::plus<>, Rep, const Value&>
  friend constexpr Quantity auto operator+(const Value& lhs, const Q& rhs);
  template<std::derived_from<quantity> Q, Representation Value>
    requires(Q::unit == ::mp_units::one) &&
            InvokeResultOf<quantity_spec, std::minus<>, Rep, const Value&>
  friend constexpr Quantity auto operator-(const Value& lhs, const Q& rhs);
  template<std::derived_from<quantity> Q, Representation Value>
    requires(Q::unit == ::mp_units::one) &&
            InvokeResultOf<quantity_spec, std::modulus<>, Rep, const Value&>
  friend constexpr Quantity auto operator%(const Value& lhs, const Q& rhs);

  template<std::derived_from<quantity> Q, auto R2, typename Rep2>
    requires InvocableQuantities<std::multiplies<>, quantity, quantity<R2, Rep2>>
  friend constexpr Quantity auto operator*(const Q& lhs, const quantity<R2, Rep2>& rhs);
  template<std::derived_from<quantity> Q, auto R2, typename Rep2>
    requires InvocableQuantities<std::divides<>, quantity, quantity<R2, Rep2>>
  friend constexpr Quantity auto operator/(const Q& lhs, const quantity<R2, Rep2>& rhs);

  template<std::derived_from<quantity> Q, typename Value>
    requires(!Quantity<Value>) && (!Reference<Value>) &&
            InvokeResultOf<quantity_spec, std::multiplies<>, Rep, const Value&>
  friend constexpr QuantityOf<quantity_spec> auto operator*(const Q& lhs, const Value& rhs);
  template<std::derived_from<quantity> Q, typename Value>
    requires(!Quantity<Value>) && (!Reference<Value>) &&
            InvokeResultOf<quantity_spec, std::divides<>, Rep, const Value&>
  friend constexpr QuantityOf<quantity_spec> auto operator/(const Q& lhs, const Value& rhs);

  template<typename Value, std::derived_from<quantity> Q>
    requires(!Quantity<Value>) && (!Reference<Value>) &&
            InvokeResultOf<quantity_spec, std::multiplies<>, const Value&, Rep>
  friend constexpr QuantityOf<quantity_spec> auto operator*(const Value& lhs, const Q& rhs);
  template<typename Value, std::derived_from<quantity> Q>
    requires(!Quantity<Value>) && (!Reference<Value>) &&
            InvokeResultOf<quantity_spec, std::divides<>, const Value&, Rep>
  friend constexpr Quantity auto operator/(const Value&, const Q&);

  // 5.6.12, comparison

  template<std::derived_from<quantity> Q, auto R2, typename Rep2>
    requires see below
  friend constexpr bool operator==(const Q& lhs, const quantity<R2, Rep2>& rhs);
  template<std::derived_from<quantity> Q, auto R2, typename Rep2>
    requires see below
  friend constexpr auto operator<=>(const Q& lhs, const quantity<R2, Rep2>& rhs);

  template<std::derived_from<quantity> Q, Representation Value>
    requires see below
  friend constexpr bool operator==(const Q& lhs, const Value& rhs);
  template<std::derived_from<quantity> Q, Representation Value>
    requires see below
  friend constexpr auto operator<=>(const Q& lhs, const Value& rhs);

  // 5.6.13, value comparison
  friend constexpr bool is_eq_zero(const quantity& q) requires see below;
  friend constexpr bool is_neq_zero(const quantity& q) requires see below;
  friend constexpr bool is_lt_zero(const quantity& q) requires see below;
  friend constexpr bool is_gt_zero(const quantity& q) requires see below;
  friend constexpr bool is_lteq_zero(const quantity& q) requires see below;
  friend constexpr bool is_gteq_zero(const quantity& q) requires see below;
};
```

```
  template<Representation Value, Reference R>
  quantity(Value, R) -> quantity<R{}, Value>;

  template<Representation Value>
  quantity(Value) -> quantity<one, Value>;

  template<QuantityLike Q>
  explicit(quantity_like_traits<Q>::explicit_import) quantity(Q)
    -> quantity<quantity_like_traits<Q>::reference, typename quantity_like_traits<Q>::rep>;

  }
```

1 `quantity<R, Rep>` is a structural type (N4971, [temp.param]) if `Rep` is a structural type.

5.6.4 Static member functions [qty.static]

```
static constexpr quantity zero() noexcept
  requires see below;
static constexpr quantity one() noexcept
  requires see below;
static constexpr quantity min() noexcept
  requires see below;
static constexpr quantity max() noexcept
  requires see below;
```

1 Let *F* be one of `zero`, `one`, `min`, and `max`.

2 *Returns*: `{representation_values<rep>::F(), R}`.

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { representation_values<rep>::F(); }
```

5.6.5 Constructors and assignment [qty.cons]

```
template<typename FwdValue, Reference R2>
  requires SameValueAs<R2{}, R, std::remove_cvref_t<FwdValue>, Rep>
constexpr quantity(FwdValue&& v, R2);

template<ValuePreservingTo<Rep> FwdValue>
  requires(unit == ::mp_units::one)
constexpr quantity(FwdValue&& v);
```

1 *Effects*: Initializes *`numerical-value`* with `std::forward<FwdValue>(v)`.

```
template<typename FwdValue, Reference R2, typename Value = std::remove_cvref_t<FwdValue>>
  requires(!SameValueAs<R2{}, R, Value, Rep>) &&
          QuantityConvertibleTo<quantity<R2{}, Value>, quantity>
constexpr quantity(FwdValue&& v, R2);
```

2 *Effects*: Equivalent to `quantity(quantity<R2{}, Value>{std::forward<FwdValue>(v), R2{}})`.

```
template<QuantityConvertibleTo<quantity> Q>
constexpr explicit(!std::convertible_to<typename Q::rep, Rep>) quantity(const Q& q);
```

3 *Effects*: Equivalent to *`sudo-cast`*`<quantity>(q)` (5.6.15).

```
template<QuantityLike Q>
  requires QuantityConvertibleTo<quantity-like-type<Q>, quantity>
constexpr explicit(see below) quantity(const Q& q);
```

4 *Effects*: Equivalent to:

```
quantity(::mp_units::quantity{quantity_like_traits<Q>::to_numerical_value(q),
                              quantity_like_traits<Q>::reference})
```

5 *Remarks*: The expression inside `explicit` is equivalent to:

```
quantity_like_traits<Q>::explicit_import ||
  !std::convertible_to<typename quantity_like_traits<Q>::rep, Rep>
```

```
template<ValuePreservingTo<Rep> FwdValue>
  requires(unit == ::mp_units::one)
constexpr quantity& operator=(FwdValue&& v);
```

6 *Effects*: Equivalent to *`numerical-value`* `= std::forward<FwdValue>(v)`.

7 *Returns*: `*this`.

5.6.6 Conversions [qty.conv]

```
template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires QuantityConvertibleTo<quantity, quantity<make-reference(quantity_spec, ToU{}), Rep>>
constexpr QuantityOf<quantity_spec> auto in(ToU) const;
```

1 *Effects*: Equivalent to: `return quantity<`*`make-reference`*`(quantity_spec, ToU{}), Rep>{*this};`

```
template<RepresentationOf<quantity_spec> ToRep>
  requires QuantityConvertibleTo<quantity, quantity<reference, ToRep>>
constexpr QuantityOf<quantity_spec> auto in() const;
```

2 *Effects*: Equivalent to: `return quantity<reference, ToRep>{*this};`

```
template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires QuantityConvertibleTo<quantity,
                                 quantity<make-reference(quantity_spec, ToU{}), ToRep>>
constexpr QuantityOf<quantity_spec> auto in(ToU) const;
```

3 *Effects*: Equivalent to:

```
return quantity<make-reference(quantity_spec, ToU{}), ToRep>{*this};
```

```
template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires requires(const quantity q) { value_cast<ToU{}>(q); }
constexpr QuantityOf<quantity_spec> auto force_in(ToU) const;
```

4 *Effects*: Equivalent to: `return value_cast<ToU{}>(*this);`

```
template<RepresentationOf<quantity_spec> ToRep>
  requires requires(const quantity q) { value_cast<ToRep>(q); }
constexpr QuantityOf<quantity_spec> auto force_in() const;
```

5 *Effects*: Equivalent to: `return value_cast<ToRep>(*this);`

```
template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires requires(const quantity q) { value_cast<ToU{}, ToRep>(q); }
constexpr QuantityOf<quantity_spec> auto force_in(ToU) const;
```

6 *Effects*: Equivalent to: `return value_cast<ToU{}, ToRep>(*this);`

5.6.7 Numerical value observers [qty.obs]

```
template<Unit U>
  requires(equivalent(U{}, unit))
constexpr rep& numerical_value_ref_in(U) & noexcept;
template<Unit U>
  requires(equivalent(U{}, unit))
constexpr const rep& numerical_value_ref_in(U) const & noexcept;
```

1 *Returns*: *`numerical-value`*.

```
template<UnitCompatibleWith<unit, quantity_spec> U>
  requires QuantityConvertibleTo<quantity, quantity<make-reference(quantity_spec, U{}), Rep>>
constexpr rep numerical_value_in(U) const noexcept;
```

2 *Effects*: Equivalent to: `return (*this).in(U{}).`*`numerical-value`*`;`

```
template<UnitCompatibleWith<unit, quantity_spec> U>
  requires requires(const quantity q) { value_cast<U{}>(q); }
```

```
constexpr rep force_numerical_value_in(U) const noexcept;
```

3 *Effects*: Equivalent to: `return (*this).force_in(U{}).`*`numerical-value`*`;`

5.6.8 Conversion operations [qty.conv.ops]

```
template<typename V_, std::constructible_from<Rep> Value = std::remove_cvref_t<V_>>
  requires(unit == ::mp_units::one)
explicit operator V_() const & noexcept;
```

1 *Returns*: *`numerical-value`*.

```
template<typename Q_, QuantityLike Q = std::remove_cvref_t<Q_>>
  requires QuantityConvertibleTo<quantity, quantity-like-type<Q>>
constexpr explicit(see below) operator Q_() const noexcept(see below);
```

2 *Effects*: Equivalent to:

```
return quantity_like_traits<Q>::from_numerical_value(
  numerical_value_in(get_unit(quantity_like_traits<Q>::reference)));
```

3 *Remarks*: The expression inside `explicit` is equivalent to:

```
quantity_like_traits<Q>::explicit_export ||
  !std::convertible_to<Rep, typename quantity_like_traits<Q>::rep>
```

The exception specification is equivalent to:

```
noexcept(quantity_like_traits<Q>::from_numerical_value(numerical-value)) &&
  std::is_nothrow_copy_constructible_v<rep>
```

5.6.9 Unary operations [qty.unary.ops]

1 In the following descriptions, let `@` be the *operator*.

```
constexpr QuantityOf<quantity_spec> auto operator+() const
  requires see below;
constexpr QuantityOf<quantity_spec> auto operator-() const
  requires see below;
```

2 *Effects*: Equivalent to: `return ::mp_units::quantity{@`*`numerical-value`*`, reference};`

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires(const rep v) {
  { @v } -> std::common_with<rep>;
}
```

```
template<Mutable<quantity> Q>
friend constexpr decltype(auto) operator++(Q&& q)
  requires see below;
template<Mutable<quantity> Q>
friend constexpr decltype(auto) operator--(Q&& q)
  requires see below;
```

4 *Effects*: Equivalent to `@q.`*`numerical-value`*.

5 *Returns*: `std::forward<Q>(q)`.

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires(rep& v) {
  { @v } -> std::same_as<rep&>;
}
```

```
constexpr QuantityOf<quantity_spec> auto operator++(int)
  requires see below;
constexpr QuantityOf<quantity_spec> auto operator--(int)
  requires see below;
```

7 *Effects*: Equivalent to: `return ::mp_units::quantity{`*`numerical-value`*`@, reference};`

8 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
    requires(rep& v) {
      { v@ } -> std::common_with<rep>;
    }
```

5.6.10 Compound assignment operations [qty.assign.ops]

1 In the following descriptions, let `@` be the *operator*.

```
template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator+=(Q&& lhs, const quantity<R2, Rep2>& rhs);
template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator-=(Q&& lhs, const quantity<R2, Rep2>& rhs);
template<Mutable<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator%=(Q&& lhs, const quantity<R2, Rep2>& rhs);
```

2 *Preconditions*: If `@` is `%=`, then `is_neq_zero(rhs)` is `true`.

3 *Effects*: Equivalent to `lhs.`*`numerical-value`* `@ rhs.in(lhs.unit).`*`numerical-value`*.

4 *Returns*: `std::forward<Q>(lhs)`.

5 *Remarks*: Let *C* be

(5.1) — `(!treat_as_floating_point<rep>)` if `@` is `%=`, and

(5.2) — `true` otherwise.

The expression in the *requires-clause* is equivalent to:

```
QuantityConvertibleTo<quantity<R2, Rep2>, quantity> && C &&
requires(rep& a, const Rep2 b) {
  { a @ b } -> std::same_as<rep&>;
}
```

6 *Recommended practice*: If `equivalent(unit, get_unit(rhs.reference))` is `true`, then the expression `rhs.in(lhs.unit)` is replaced with `rhs`.

```
template<Mutable<quantity> Q, ValuePreservingTo<Rep> Value>
  requires see below
friend constexpr decltype(auto) operator*=(Q&& lhs, const Value& rhs);
template<Mutable<quantity> Q, ValuePreservingTo<Rep> Value>
  requires see below
friend constexpr decltype(auto) operator/=(Q&& lhs, const Value& rhs);
```

7 *Preconditions*: If `@` is `/=`, then `rhs != representation_values<Value>::zero()` is `true`.

8 *Effects*: Equivalent to `lhs.`*`numerical-value`* `@ rhs`.

9 *Returns*: `std::forward<Q>(lhs)`.

10 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(!Quantity<Value>) && requires(rep& a, const Value b) {
  { a @ b } -> std::same_as<rep&>;
}
```

```
template<Mutable<quantity> Q, QuantityOf<dimensionless> Q2>
  requires see below
friend constexpr decltype(auto) operator*=(Q&& lhs, const Q2& rhs);
template<Mutable<quantity> Q, QuantityOf<dimensionless> Q2>
  requires see below
friend constexpr decltype(auto) operator/=(Q&& lhs, const Q2& rhs);
```

11 *Effects*: Equivalent to: `return std::forward<Q>(lhs) @ rhs.`*`numerical-value`*`;`

12 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(Q2::unit == ::mp_units::one) && ValuePreservingTo<typename Q2::rep, Rep> &&
requires(rep& a, const Q2::rep b) {
  { a @ b } -> std::same_as<rep&>;
}
```

5.6.11 Arithmetic operations [qty.arith.ops]

1 In the following descriptions, let `@` be the *operator*.

```
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires CommonlyInvocableQuantities<std::plus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator+(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires CommonlyInvocableQuantities<std::minus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator-(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires(!treat_as_floating_point<Rep>) && (!treat_as_floating_point<Rep2>) &&
          CommonlyInvocableQuantities<std::modulus<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator%(const Q& lhs, const quantity<R2, Rep2>& rhs);
```

2 Let F be the first argument to `CommonlyInvocableQuantities`.

3 *Preconditions*: If `@` is `%`, then `is_neq_zero(rhs)` is `true`.

4 *Effects*: Equivalent to:

```
using ret = common-quantity-for<F, quantity, quantity<R2, Rep2>>;
const ret ret_lhs(lhs);
const ret ret_rhs(rhs);
return ::mp_units::quantity{
  ret_lhs.numerical_value_ref_in(ret::unit) @ ret_rhs.numerical_value_ref_in(ret::unit),
  ret::reference};
```

```
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::plus<>, Rep, const Value&>
friend constexpr Quantity auto operator+(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::minus<>, Rep, const Value&>
friend constexpr Quantity auto operator-(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::modulus<>, Rep, const Value&>
friend constexpr Quantity auto operator%(const Q& lhs, const Value& rhs);
```

5 *Effects*: Equivalent to: `return lhs @ ::mp_units::quantity{rhs};`

```
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::plus<>, Rep, const Value&>
friend constexpr Quantity auto operator+(const Value& lhs, const Q& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::minus<>, Rep, const Value&>
friend constexpr Quantity auto operator-(const Value& lhs, const Q& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires(Q::unit == ::mp_units::one) &&
          InvokeResultOf<quantity_spec, std::modulus<>, Rep, const Value&>
friend constexpr Quantity auto operator%(const Value& lhs, const Q& rhs);
```

6 *Effects*: Equivalent to: `return ::mp_units::quantity{lhs} @ rhs;`

```
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires InvocableQuantities<std::multiplies<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator*(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires InvocableQuantities<std::divides<>, quantity, quantity<R2, Rep2>>
friend constexpr Quantity auto operator/(const Q& lhs, const quantity<R2, Rep2>& rhs);
```

7 *Preconditions*: If `@` is `/`, then `is_neq_zero(rhs)` is `true`.

8 *Effects*: Equivalent to:

```
    return ::mp_units::quantity{
      lhs.numerical_value_ref_in(unit) @ rhs.numerical_value_ref_in(rhs.unit), R @ R2};
```

```
template<std::derived_from<quantity> Q, typename Value>
  requires(!Quantity<Value>) && (!Reference<Value>) &&
          InvokeResultOf<quantity_spec, std::multiplies<>, Rep, const Value&>
friend constexpr QuantityOf<quantity_spec> auto operator*(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, typename Value>
  requires(!Quantity<Value>) && (!Reference<Value>) &&
          InvokeResultOf<quantity_spec, std::divides<>, Rep, const Value&>
friend constexpr QuantityOf<quantity_spec> auto operator/(const Q& lhs, const Value& rhs);
```

9 *Preconditions*: If `@` is `/`, then `rhs != representation_values<Value>::zero()` is `true`.

10 *Effects*: Equivalent to:

```
    return ::mp_units::quantity{lhs.numerical_value_ref_in(unit) @ rhs, R};
```

```
template<typename Value, std::derived_from<quantity> Q>
  requires(!Quantity<Value>) && (!Reference<Value>) &&
          InvokeResultOf<quantity_spec, std::multiplies<>, const Value&, Rep>
friend constexpr QuantityOf<quantity_spec> auto operator*(const Value& lhs, const Q& rhs);
template<typename Value, std::derived_from<quantity> Q>
  requires(!Quantity<Value>) && (!Reference<Value>) &&
          InvokeResultOf<quantity_spec, std::divides<>, const Value&, Rep>
friend constexpr Quantity auto operator/(const Value& lhs, const Q& rhs);
```

11 *Preconditions*: If `@` is `/`, then `is_neq_zero(rhs)` is `true`.

12 *Effects*: Equivalent to:

```
    return ::mp_units::quantity{lhs @ rhs.numerical_value_ref_in(unit), ::mp_units::one @ R};
```

5.6.12 Comparison [qty.cmp]

1 In the following descriptions, let `@` be the *operator*.

```
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr bool operator==(const Q& lhs, const quantity<R2, Rep2>& rhs);
template<std::derived_from<quantity> Q, auto R2, typename Rep2>
  requires see below
friend constexpr auto operator<=>(const Q& lhs, const quantity<R2, Rep2>& rhs);
```

2 Let *C* be `std::equality_comparable` if `@` is `==`, and `std::three_way_comparable` if `@` is `<=>`.

3 *Effects*: Equivalent to:

```
    using ct = std::common_type_t<quantity, quantity<R2, Rep2>>;
    const ct ct_lhs(lhs);
    const ct ct_rhs(rhs);
    return ct_lhs.numerical_value_ref_in(ct::unit) @ ct_rhs.numerical_value_ref_in(ct::unit);
```

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
    requires {
      typename std::common_type_t<quantity, quantity<R2, Rep2>>;
    } && C<typename std::common_type_t<quantity, quantity<R2, Rep2>>::rep>
```

```
template<std::derived_from<quantity> Q, Representation Value>
  requires see below
friend constexpr bool operator==(const Q& lhs, const Value& rhs);
template<std::derived_from<quantity> Q, Representation Value>
  requires see below
friend constexpr auto operator<=>(const Q& lhs, const Value& rhs);
```

5 Let *C* be `std::equality_comparable_with` if `@` is `==`, and `std::three_way_comparable_with` if `@` is `<=>`.

6 *Returns*: `lhs.numerical_value_ref_in(unit) @ rhs`.

7 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
    (Q::unit == ::mp_units::one) && C<Rep, Value>
```

5.6.13 Value comparison [qty.val.cmp]

```
friend constexpr bool is_eq_zero(const quantity& q) requires see below;
friend constexpr bool is_neq_zero(const quantity& q) requires see below;
friend constexpr bool is_lt_zero(const quantity& q) requires see below;
friend constexpr bool is_gt_zero(const quantity& q) requires see below;
friend constexpr bool is_lteq_zero(const quantity& q) requires see below;
friend constexpr bool is_gteq_zero(const quantity& q) requires see below;
```

1 Let `is_`*F*`_zero` be the function name.

2 *Returns*:

(2.1) — If *F* is `eq`, returns `q == zero()`.

(2.2) — Otherwise, if *F* is `neq`, returns `q != zero()`.

(2.3) — Otherwise, returns `is_`*F*`(q <=> zero())` (N4971, [compare.syn]).

3 *Remarks*: Let *C* be `std::equality_comparable_with` if *F* is `eq` or `neq`, and `std::three_way_comparable_with` otherwise. The expression in the *requires-clause* is equivalent to:

```
requires {
  { T::zero() } -> C<quantity>;
}
```

5.6.14 Construction helper `delta` [qty.delta]

```
namespace mp_units {

template<Reference R>
struct delta_ {
  template<typename FwdRep,
           RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  constexpr quantity<R{}, Rep> operator()(FwdRep&& lhs) const;
};

}
```

```
template<typename FwdRep,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
constexpr quantity<R{}, Rep> operator()(FwdRep&& lhs) const;
```

1 *Effects*: Equivalent to: `return quantity{std::forward<FwdRep>(lhs), R{}};`

5.6.15 Non-member conversions [qty.non.mem.conv]

```
template<Quantity To, typename FwdFrom, Quantity From = std::remove_cvref_t<FwdFrom>>
  requires see below
constexpr To sudo-cast(FwdFrom&& q);    // exposition only
```

1 *Returns*: TBD.

2 `value_cast` is an explicit cast that allows truncation.

```
template<Unit auto ToU, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
  requires(convertible(Q::reference, ToU))
constexpr Quantity auto value_cast(FwdQ&& q);
```

3 *Effects*: Equivalent to:

```
return sudo-cast<quantity<make-reference(Q::quantity_spec, ToU), typename Q::rep>>(
  std::forward<FwdQ>(q));
```

```
template<Representation ToRep, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
  requires RepresentationOf<ToRep, Q::quantity_spec> &&
           std::constructible_from<ToRep, typename Q::rep>
constexpr quantity<Q::reference, ToRep> value_cast(FwdQ&& q);
```

4 *Effects*: Equivalent to:

```
    return sudo-cast<quantity<Q::reference, ToRep>>(std::forward<FwdQ>(q));
```

```
template<Unit auto ToU, Representation ToRep, typename FwdQ,
         Quantity Q = std::remove_cvref_t<FwdQ>>
  requires see below
constexpr Quantity auto value_cast(FwdQ&& q);
template<Representation ToRep, Unit auto ToU, typename FwdQ,
         Quantity Q = std::remove_cvref_t<FwdQ>>
  requires see below
constexpr Quantity auto value_cast(FwdQ&& q);
```

5 *Effects*: Equivalent to:

```
return sudo-cast<quantity<make-reference(Q::quantity_spec, ToU), ToRep>>(
  std::forward<FwdQ>(q));
```

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(convertible(Q::reference, ToU)) && RepresentationOf<ToRep, Q::quantity_spec> &&
std::constructible_from<ToRep, typename Q::rep>
```

```
template<Quantity ToQ, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
  requires(convertible(Q::reference, ToQ::unit)) && (ToQ::quantity_spec == Q::quantity_spec) &&
          std::constructible_from<typename ToQ::rep, typename Q::rep>
constexpr Quantity auto value_cast(FwdQ&& q);
```

7 *Effects*: Equivalent to: `return sudo-cast<ToQ>(std::forward<FwdQ>(q));`

8 `quantity_cast` is an explicit cast that allows converting to more specific quantities.

[*Example 1*:

```
auto length = isq::length(42 * m);
auto distance = quantity_cast<isq::distance>(length);
```

— *end example*]

```
template<QuantitySpec auto ToQS, typename FwdQ, Quantity Q = std::remove_cvref_t<FwdQ>>
  requires QuantitySpecCastableTo<Q::quantity_spec, ToQS>
constexpr Quantity auto quantity_cast(FwdQ&& q);
```

9 *Effects*: Equivalent to:

```
return quantity{std::forward<FwdQ>(q).numerical-value, make-reference(ToQS, Q::unit)};
```

5.6.16 Math [qty.math]

```
template<auto R, typename Rep>
  requires requires(Rep v) { abs(v); } || requires(Rep v) { std::abs(v); }
constexpr quantity<R, Rep> abs(const quantity<R, Rep>& q) noexcept;
```

1 *Effects*: Equivalent to:

```
using std::abs;
return {static_cast<Rep>(abs(q.numerical_value_ref_in(q.unit))), R};
```

```
template<std::intmax_t Num, std::intmax_t Den = 1, auto R, typename Rep>
  requires (Den != 0) && requires(Rep v) {
    representation_values<Rep>::one();
    requires requires { pow(v, 1.0); } || requires { std::pow(v, 1.0); };
  }
constexpr quantity<pow<Num, Den>(R), Rep> pow(const quantity<R, Rep>& q) noexcept;      // hosted
```

2 *Effects*: Equivalent to:

```
if constexpr (Num == 0) {
  return quantity<pow<Num, Den>(R), Rep>::one();
} else if constexpr (Num == Den) {
  return q;
} else {
  using std::pow;
  return {static_cast<Rep>(pow(q.numerical_value_ref_in(q.unit),
                               static_cast<double>(Num) / static_cast<double>(Den))),
          pow<Num, Den>(R)};
}
```

```
template<auto R, typename Rep>
  requires requires(Rep v) { sqrt(v); } || requires(Rep v) { std::sqrt(v); }
constexpr quantity<sqrt(R), Rep> sqrt(const quantity<R, Rep>& q) noexcept;            // hosted
```

3 *Effects*: Equivalent to:

```
using std::sqrt;
return {static_cast<Rep>(sqrt(q.numerical_value_ref_in(q.unit))), sqrt(R)};
```

```
template<auto R, typename Rep>
  requires requires(Rep v) { cbrt(v); } || requires(Rep v) { std::cbrt(v); }
constexpr quantity<cbrt(R), Rep> cbrt(const quantity<R, Rep>& q) noexcept;            // hosted
```

4 *Effects*: Equivalent to:

```
using std::cbrt;
return {static_cast<Rep>(cbrt(q.numerical_value_ref_in(q.unit))), cbrt(R)};
```

```
template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires requires(Rep v) { exp(v); } || requires(Rep v) { std::exp(v); }
constexpr quantity<R, Rep> exp(const quantity<R, Rep>& q);                             // hosted
```

5 *Effects*: Equivalent to:

```
using std::exp;
return value_cast<get_unit(R)>(
  quantity{static_cast<Rep>(exp(q.force_numerical_value_in(q.unit))),
           clone-reference-with<one>(R)});
```

```
template<auto R, typename Rep>
  requires requires(Rep v) { isfinite(v); } || requires(Rep v) { std::isfinite(v); }
constexpr bool isfinite(const quantity<R, Rep>& a) noexcept;                           // hosted
```

6 *Effects*: Equivalent to:

```
using std::isfinite;
return isfinite(a.numerical_value_ref_in(a.unit));
```

```
template<auto R, typename Rep>
  requires requires(Rep v) { isinf(v); } || requires(Rep v) { std::isinf(v); }
constexpr bool isinf(const quantity<R, Rep>& a) noexcept;                              // hosted
```

7 *Effects*: Equivalent to:

```
using std::isinf;
return isinf(a.numerical_value_ref_in(a.unit));
```

```
template<auto R, typename Rep>
  requires requires(Rep v) { isnan(v); } || requires(Rep v) { std::isnan(v); }
constexpr bool isnan(const quantity<R, Rep>& a) noexcept;                              // hosted
```

8 *Effects*: Equivalent to:

```
using std::isnan;
return isnan(a.numerical_value_ref_in(a.unit));
```

```
template<auto R, auto S, auto T, typename Rep1, typename Rep2, typename Rep3>
  requires
    requires {
      get_common_quantity_spec(get_quantity_spec(R) * get_quantity_spec(S),
                               get_quantity_spec(T));
    } && (equivalent(get_unit(R) * get_unit(S), get_unit(T))) &&
    requires(Rep1 v1, Rep2 v2, Rep3 v3) {
      requires requires { fma(v1, v2, v3); } || requires { std::fma(v1, v2, v3); };
    }
```

```
constexpr QuantityOf<get_common_quantity_spec(get_quantity_spec(R) * get_quantity_spec(S),
                                              get_quantity_spec(T))> auto
fma(const quantity<R, Rep1>& a, const quantity<S, Rep2>& x,                              // hosted
    const quantity<T, Rep3>& b) noexcept;
```

9 *Effects*: Equivalent to:

```
using std::fma;
return quantity{fma(a.numerical_value_ref_in(a.unit), x.numerical_value_ref_in(x.unit),
                    b.numerical_value_ref_in(b.unit)),
                get_common_reference(R * S, T)};
```

```
template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires requires(Rep1 v1, Rep2 v2) {
    get_common_reference(R1, R2);
    requires requires { fmod(v1, v2); } || requires { std::fmod(v1, v2); };
  }
constexpr QuantityOf<get_quantity_spec(R1)> auto fmod(const quantity<R1, Rep1>& x,      // hosted
                                                      const quantity<R2, Rep2>& y) noexcept;
```

10 *Effects*: Equivalent to:

```
constexpr auto ref = get_common_reference(R1, R2);
constexpr auto unit = get_unit(ref);
using std::fmod;
return quantity{fmod(x.numerical_value_in(unit), y.numerical_value_in(unit)), ref};
```

```
template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires requires(Rep1 v1, Rep2 v2) {
    get_common_reference(R1, R2);
    requires requires { remainder(v1, v2); } || requires { std::remainder(v1, v2); };
  }
constexpr QuantityOf<get_quantity_spec(R1)> auto remainder(const quantity<R1, Rep1>& x, // hosted
                                                           const quantity<R2, Rep2>& y) noexcept;
```

11 *Effects*: Equivalent to:

```
constexpr auto ref = get_common_reference(R1, R2);
constexpr auto unit = get_unit(ref);
using std::remainder;
return quantity{remainder(x.numerical_value_in(unit), y.numerical_value_in(unit)), ref};
```

```
template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> floor(                              // hosted
  const quantity<R, Rep>& q) noexcept
  requires((!treat_as_floating_point<Rep>) || requires(Rep v) { floor(v); } ||
           requires(Rep v) { std::floor(v); }) &&
            (equivalent(To, get_unit(R)) || requires {
              q.force_in(To);
              representation_values<Rep>::one();
            });
```

12 *Effects*: Equivalent to:

```
const auto handle_signed_results = [&]<typename T>(const T& res) {
  if (res > q) {
    return res - T::one();
  }
  return res;
};
if constexpr (treat_as_floating_point<Rep>) {
  using std::floor;
  if constexpr (equivalent(To, get_unit(R))) {
    return {static_cast<Rep>(floor(q.numerical_value_ref_in(q.unit))),
            clone-reference-with<To>(R)};
  } else {
    return handle_signed_results(
      quantity{static_cast<Rep>(floor(q.force_numerical_value_in(To))),
               clone-reference-with<To>(R)});
  }
} else {
  if constexpr (equivalent(To, get_unit(R))) {
    return q.force_in(To);
  } else {
    return handle_signed_results(q.force_in(To));
  }
}
```

```
template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> ceil(                          // hosted
  const quantity<R, Rep>& q) noexcept
  requires((!treat_as_floating_point<Rep>) || requires(Rep v) { ceil(v); } ||
           requires(Rep v) { std::ceil(v); }) &&
           (equivalent(To, get_unit(R)) || requires {
              q.force_in(To);
              representation_values<Rep>::one();
            });
```

13 *Effects*: Equivalent to:

```
const auto handle_signed_results = [&]<typename T>(const T& res) {
  if (res < q) {
    return res + T::one();
  }
  return res;
};
if constexpr (treat_as_floating_point<Rep>) {
  using std::ceil;
  if constexpr (equivalent(To, get_unit(R))) {
    return {static_cast<Rep>(ceil(q.numerical_value_ref_in(q.unit))),
            clone-reference-with<To>(R)};
  } else {
    return handle_signed_results(quantity{
      static_cast<Rep>(ceil(q.force_numerical_value_in(To))), clone-reference-with<To>(R)});
  }
} else {
  if constexpr (equivalent(To, get_unit(R))) {
    return q.force_in(To);
  } else {
    return handle_signed_results(q.force_in(To));
  }
}
```

```
template<Unit auto To, auto R, typename Rep>
constexpr quantity<clone-reference-with<To>(R), Rep> round(                         // hosted
  const quantity<R, Rep>& q) noexcept
  requires((!treat_as_floating_point<Rep>) || requires(Rep v) { round(v); } ||
           requires(Rep v) { std::round(v); }) &&
           (equivalent(To, get_unit(R)) || requires {
              ::mp_units::floor<To>(q);
              representation_values<Rep>::one();
            });
```

14 *Effects*: Equivalent to:

```
if constexpr (equivalent(To, get_unit(R))) {
  if constexpr (treat_as_floating_point<Rep>) {
    using std::round;
    return {static_cast<Rep>(round(q.numerical_value_ref_in(q.unit))),
            clone-reference-with<To>(R)};
  } else {
    return q.force_in(To);
  }
} else {
  const auto res_low = mp_units::floor<To>(q);
```

```
      const auto res_high = res_low + res_low.one();
      const auto diff0 = q - res_low;
      const auto diff1 = res_high - q;
      if (diff0 == diff1) {
        if (static_cast<int>(res_low.numerical_value_ref_in(To)) & 1) {
          return res_high;
        }
        return res_low;
      }
      if (diff0 < diff1) {
        return res_low;
      }
      return res_high;
    }
```

```
template<Unit auto To, auto R, typename Rep>
constexpr QuantityOf<dimensionless / get_quantity_spec(R)> auto inverse(              // hosted
  const quantity<R, Rep>& q)
  requires requires {
    representation_values<Rep>::one();
    value_cast<To>(1 / q);
  };
```

15 *Effects*: Equivalent to:

```
return (representation_values<Rep>::one() * one).force_in(To * quantity<R, Rep>::unit) / q;
```

```
template<auto R1, typename Rep1, auto R2, typename Rep2>
  requires requires(Rep1 v1, Rep2 v2) {
    get_common_reference(R1, R2);
    requires requires { hypot(v1, v2); } || requires { std::hypot(v1, v2); };
  }
constexpr QuantityOf<get_quantity_spec(get_common_reference(R1, R2))> auto hypot(     // hosted
  const quantity<R1, Rep1>& x, const quantity<R2, Rep2>& y) noexcept;
```

16 *Effects*: Equivalent to:

```
constexpr auto ref = get_common_reference(R1, R2);
constexpr auto unit = get_unit(ref);
using std::hypot;
return quantity{hypot(x.numerical_value_in(unit), y.numerical_value_in(unit)), ref};
```

```
template<auto R1, typename Rep1, auto R2, typename Rep2, auto R3, typename Rep3>
  requires requires(Rep1 v1, Rep2 v2, Rep3 v3) {
    get_common_reference(R1, R2, R3);
    requires requires { hypot(v1, v2, v3); } || requires { std::hypot(v1, v2, v3); };
  }
constexpr QuantityOf<get_quantity_spec(get_common_reference(R1, R2, R3))> auto hypot(  // hosted
  const quantity<R1, Rep1>& x, const quantity<R2, Rep2>& y,
  const quantity<R3, Rep3>& z) noexcept;
```

17 *Effects*: Equivalent to:

```
constexpr auto ref = get_common_reference(R1, R2);
constexpr auto unit = get_unit(ref);
using std::hypot;
return quantity{
  hypot(x.numerical_value_in(unit), y.numerical_value_in(unit), z.numerical_value_in(unit)),
  ref};
```

5.6.17 `std::common_type` specializations [qty.common.type]

```
template<mp_units::Quantity Q1, mp_units::Quantity Q2>
  requires requires {
    { mp_units::get_common_reference(Q1::reference, Q2::reference) } -> mp_units::Reference;
    typename std::common_type_t<typename Q1::rep, typename Q2::rep>;
    requires mp_units::RepresentationOf<std::common_type_t<typename Q1::rep, typename Q2::rep>,
                                        mp_units::get_common_quantity_spec(Q1::quantity_spec,
                                                                           Q2::quantity_spec)>;
  }
struct std::common_type<Q1, Q2> {
  using type = mp_units::quantity<mp_units::get_common_reference(Q1::reference, Q2::reference),
                                  std::common_type_t<typename Q1::rep, typename Q2::rep>>;
};

template<mp_units::Quantity Q, mp_units::Representation Value>
  requires(Q::unit == mp_units::one) && requires {
    typename mp_units::quantity<Q::reference, std::common_type_t<typename Q::rep, Value>>;
  }
struct std::common_type<Q, Value> {
  using type = mp_units::quantity<Q::reference, std::common_type_t<typename Q::rep, Value>>;
};
```

5.6.18 Random [qty.rand]

5.6.18.1 Requirements [qty.rand.req]

1 A class D instantiated from a class template described in subclause 5.6.18 meets the following requirements. In this subclase,

(1.1) — d is a value of D, and x is a (possibly const) value of D,

(1.2) — args denotes an *expression-list*,

(1.3) — param$_i$ denotes the i_{th} function parameter, and

(1.4) — unwrapped_param$_i$ denotes param$_i$.numerical_value_ref_in(Q::unit) if param$_i$ is an lvalue reference to const Q, and param$_i$ otherwise.

```
D u;
D u = D();
```

2 *Effects*: Default-initializes the base subobject.

```
D(args)
```

3 *Effects*: Initializes the base subobject with unwrapped_param$_0$, . . . , unwrapped_param$_n$.

```
d(g)
```

4 *Effects*: Equivalent to: return base::operator()(g) * Q::reference;

```
x.f()
```

5 *Effects*: For all other *f*, unless otherwise specified, equivalent to: return base::*f*() * Q::reference;

5.6.18.2 Uniform distributions [qty.rand.uni]

5.6.18.2.1 Class template uniform_int_distribution [qty.rand.uni.int]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct uniform_int_distribution                                                    // hosted
    : public std::uniform_int_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::uniform_int_distribution<rep>;

  uniform_int_distribution();
  uniform_int_distribution(const Q& a, const Q& b);

  template<typename Generator>
  Q operator()(Generator& g);

  Q a() const;
  Q b() const;

  Q min() const;
  Q max() const;
};
}
```

5.6.18.2.2 Class template uniform_real_distribution [qty.rand.uni.real]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct uniform_real_distribution                                                       // hosted
    : public std::uniform_real_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::uniform_real_distribution<rep>;

  uniform_real_distribution();
  uniform_real_distribution(const Q& a, const Q& b);

  template<typename Generator>
  Q operator()(Generator& g);

  Q a() const;
  Q b() const;

  Q min() const;
  Q max() const;
};
}
```

5.6.18.3 Bernoulli distributions [qty.rand.bern]

5.6.18.3.1 Class template binomial_distribution [qty.rand.bern.bin]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct binomial_distribution                                                           // hosted
    : public std::binomial_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::binomial_distribution<rep>;

  binomial_distribution();
  binomial_distribution(const Q& t, double p);

  template<typename Generator>
  Q operator()(Generator& g);

  Q t() const;

  Q min() const;
  Q max() const;
};
}
```

5.6.18.3.2 Class template negative_binomial_distribution [qty.rand.bern.negbin]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct negative_binomial_distribution                                                  // hosted
    : public std::negative_binomial_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::negative_binomial_distribution<rep>;

  negative_binomial_distribution();
  negative_binomial_distribution(const Q& k, double p);
  template<typename Generator>
  Q operator()(Generator& g);

  Q k() const;

  Q min() const;
  Q max() const;
};
}
```

5.6.18.3.3 Class template geometric_distribution [qty.rand.bern.geo]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct geometric_distribution                                       // hosted
    : public std::geometric_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::geometric_distribution<rep>;

  geometric_distribution();
  explicit geometric_distribution(double p);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.4 Poisson distrubutions [qty.rand.pois]

5.6.18.4.1 Class template poisson_distribution [qty.rand.pois.poisson]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct poisson_distribution                                         // hosted
    : public std::poisson_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::poisson_distribution<rep>;

  poisson_distribution();
  explicit poisson_distribution(double p);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.4.2 Class template exponential_distribution [qty.rand.pois.exp]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct exponential_distribution                                     // hosted
    : public std::exponential_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::exponential_distribution<rep>;

  exponential_distribution();
  explicit exponential_distribution(const rep& lambda);
  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.4.3 Class template gamma_distribution [qty.rand.pois.gamma]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct gamma_distribution                                              // hosted
    : public std::gamma_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::gamma_distribution<rep>;

  gamma_distribution();
  gamma_distribution(const rep& alpha, const rep& beta);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.4.4 Class template weibull_distribution [qty.rand.pois.weibull]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct weibull_distribution                                            // hosted
    : public std::weibull_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::weibull_distribution<rep>;

  weibull_distribution();
  weibull_distribution(const rep& a, const rep& b);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.4.5 Class template extreme_value_distribution [qty.rand.pois.extreme]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct extreme_value_distribution                                      // hosted
    : public std::extreme_value_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::extreme_value_distribution<rep>;

  extreme_value_distribution();
  extreme_value_distribution(const Q& a, const rep& b);

  template<typename Generator>
  Q operator()(Generator& g);
```

```
    Q a() const;

    Q min() const;
    Q max() const;
  };
  }
```

```
template<typename Generator>
Q operator()(Generator& g);
```

1 *Effects*: Equivalent to: `return Q(base::operator()(g));`

5.6.18.5 Normal distributions [qty.rand.norm]

5.6.18.5.1 Class template `normal_distribution` [qty.rand.norm.normal]

```
  namespace mp_units {
  template<Quantity Q>
    requires std::floating_point<typename Q::rep>
  struct normal_distribution                                              // hosted
      : public std::normal_distribution<typename Q::rep> {
    using rep = Q::rep;
    using base = std::normal_distribution<rep>;

    normal_distribution();
    normal_distribution(const Q& mean, const Q& stddev);

    template<typename Generator>
    Q operator()(Generator& g);

    Q mean() const;
    Q stddev() const;

    Q min() const;
    Q max() const;
  };
  }
```

```
template<typename Generator>
Q operator()(Generator& g);
```

1 *Effects*: Equivalent to: `return Q(base::operator()(g));`

5.6.18.5.2 Class template `lognormal_distribution` [qty.rand.norm.lognormal]

```
  namespace mp_units {
  template<Quantity Q>
    requires std::floating_point<typename Q::rep>
  struct lognormal_distribution                                           // hosted
      : public std::lognormal_distribution<typename Q::rep> {
    using rep = Q::rep;
    using base = std::lognormal_distribution<rep>;

    lognormal_distribution();
    lognormal_distribution(const Q& m, const Q& s);

    template<typename Generator>
    Q operator()(Generator& g);

    Q m() const;
    Q s() const;

    Q min() const;
    Q max() const;
  };
  }
```

5.6.18.5.3 Class template chi_squared_distribution [qty.rand.norm.chisq]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct chi_squared_distribution                                                    // hosted
    : public std::chi_squared_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::chi_squared_distribution<rep>;

  chi_squared_distribution();
  explicit chi_squared_distribution(const rep& n);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.5.4 Class template cauchy_distribution [qty.rand.norm.cauchy]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct cauchy_distribution                                                         // hosted
    : public std::cauchy_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::cauchy_distribution<rep>;

  cauchy_distribution();
  cauchy_distribution(const Q& a, const Q& b);

  template<typename Generator>
  Q operator()(Generator& g);

  Q a() const;
  Q b() const;

  Q min() const;
  Q max() const;
};
}
```

5.6.18.5.5 Class template fisher_f_distribution [qty.rand.norm.f]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct fisher_f_distribution                                                       // hosted
    : public std::fisher_f_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::fisher_f_distribution<rep>;

  fisher_f_distribution();
  fisher_f_distribution(const rep& m, const rep& n);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.5.6 Class template `student_t_distribution` [qty.rand.norm.t]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
struct student_t_distribution                                                    // hosted
    : public std::student_t_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::student_t_distribution<rep>;

  student_t_distribution();
  explicit student_t_distribution(const rep& n);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.6 Sampling distributions [qty.rand.samp]

5.6.18.6.1 Class template `discrete_distribution` [qty.rand.samp.discrite]

```
namespace mp_units {
template<Quantity Q>
  requires std::integral<typename Q::rep>
struct discrete_distribution                                                     // hosted
    : public std::discrete_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::discrete_distribution<rep>;

  discrete_distribution();

  template<typename InputIt>
  discrete_distribution(InputIt first, InputIt last);

  discrete_distribution(std::initializer_list<double> weights);

  template<typename UnaryOperation>
  discrete_distribution(std::size_t count, double xmin, double xmax, UnaryOperation unary_op);

  template<typename Generator>
  Q operator()(Generator& g);

  Q min() const;
  Q max() const;
};
}
```

5.6.18.6.2 Utilities [qty.rand.samp.utils]

1 The following exposition-only function templates are used in the following subclauses.

```
template<Quantity Q, typename InputIt>
std::vector<typename Q::rep> i-qty-to-rep(InputIt first, InputIt last);
```

2 *Effects*: Equivalent to:

```
std::vector<typename Q::rep> intervals_rep;
intervals_rep.reserve(static_cast<std::size_t>(std::distance(first, last)));
for (InputIt itr = first; itr != last; ++itr) {
  intervals_rep.push_back(itr->numerical_value_ref_in(Q::unit));
}
return intervals_rep;
```

```
template<Quantity Q>
std::vector<typename Q::rep> bl-qty-to-rep(std::initializer_list<Q>& bl);
```

3 *Effects*: Equivalent to:

```
std::vector<typename Q::rep> bl_rep;
bl_rep.reserve(bl.size());
for (const Q& qty : bl) {
  bl_rep.push_back(qty.numerical_value_ref_in(Q::unit));
}
return bl_rep;
```

```
template<Quantity Q, typename UnaryOperation>
std::vector<typename Q::rep> fw-bl-pwc(std::initializer_list<Q>& bl, UnaryOperation fw);
```

4 *Effects*: Equivalent to:

```
using rep = Q::rep;
std::vector<rep> w_bl;
w_bl.reserve(bl.size());
for (const Q& qty : bl) {
  w_bl.push_back(fw(qty));
}
std::vector<rep> weights;
weights.reserve(bl.size());
for (std::size_t i = 0; i < bl.size() - 1; ++i) {
  weights.push_back(w_bl[i] + w_bl[i + 1]);
}
weights.push_back(0);
return weights;
```

```
template<Quantity Q, typename UnaryOperation>
std::vector<typename Q::rep> fw-bl-pwl(std::initializer_list<Q>& bl, UnaryOperation fw);
```

5 *Effects*: Equivalent to:

```
std::vector<typename Q::rep> weights;
weights.reserve(bl.size());
for (const Q& qty : bl) {
  weights.push_back(fw(qty));
}
return weights;
```

5.6.18.6.3 Class template piecewise_constant_distribution [qty.rand.samp.pconst]

```
namespace mp_units {
template<Quantity Q>
  requires std::floating_point<typename Q::rep>
class piecewise_constant_distribution                                          // hosted
    : public std::piecewise_constant_distribution<typename Q::rep> {
  using rep = Q::rep;
  using base = std::piecewise_constant_distribution<rep>;

  template<typename InputIt>
  piecewise_constant_distribution(const std::vector<rep>& i, InputIt first_w);

  piecewise_constant_distribution(const std::vector<rep>& bl, const std::vector<rep>& weights);

public:
  piecewise_constant_distribution();

  template<typename InputIt1, typename InputIt2>
  piecewise_constant_distribution(InputIt1 first_i, InputIt1 last_i, InputIt2 first_w);

  template<typename UnaryOperation>
  piecewise_constant_distribution(std::initializer_list<Q> bl, UnaryOperation fw);
```

```
    template<typename UnaryOperation>
    piecewise_constant_distribution(std::size_t nw, const Q& xmin, const Q& xmax,
                                    UnaryOperation fw);

    template<typename Generator>
    Q operator()(Generator& g);

    std::vector<Q> intervals() const;

    Q min() const;
    Q max() const;
  };
  }
```

```
template<typename InputIt>
piecewise_constant_distribution(const std::vector<rep>& i, InputIt first_w);
```

1 *Effects*: Initializes the base subobject with

```
base(i.cbegin(), i.cend(), first_w)
```

```
piecewise_constant_distribution(const std::vector<rep>& bl, const std::vector<rep>& weights);
```

2 *Effects*: Initializes the base subobject with

```
base(bl.cbegin(), bl.cend(), weights.cbegin())
```

```
template<typename InputIt1, typename InputIt2>
piecewise_constant_distribution(InputIt1 first_i, InputIt1 last_i, InputIt2 first_w);
```

3 *Effects*: Equivalent to:

```
piecewise_constant_distribution(i-qty-to-rep<Q>(first_i, last_i), first_w)
```

```
template<typename UnaryOperation>
piecewise_constant_distribution(std::initializer_list<Q> bl, UnaryOperation fw);
```

4 *Effects*: Equivalent to:

```
piecewise_constant_distribution(bl-qty-to-rep(bl), fw-bl-pwc(bl, fw))
```

```
template<typename UnaryOperation>
piecewise_constant_distribution(std::size_t nw, const Q& xmin, const Q& xmax, UnaryOperation fw);
```

5 *Effects*: Initializes the base subobject with

```
base(nw, xmin.numerical_value_ref_in(Q::unit), xmax.numerical_value_ref_in(Q::unit),
     [fw](rep val) { return fw(val * Q::reference); })
```

```
std::vector<Q> intervals() const;
```

6 *Effects*: Equivalent to:

```
const std::vector<rep> intervals_rep = base::intervals();
std::vector<Q> intervals_qty;
intervals_qty.reserve(intervals_rep.size());
for (const rep& val : intervals_rep) {
  intervals_qty.push_back(val * Q::reference);
}
return intervals_qty;
```

5.6.18.6.4 Class template piecewise_linear_distribution [qty.rand.samp.plinear]

```
  namespace mp_units {
  template<Quantity Q>
    requires std::floating_point<typename Q::rep>
  class piecewise_linear_distribution                                    // hosted
      : public std::piecewise_linear_distribution<typename Q::rep> {
    using rep = Q::rep;
    using base = std::piecewise_linear_distribution<rep>;

    template<typename InputIt>
    piecewise_linear_distribution(const std::vector<rep>& i, InputIt first_w);
    piecewise_linear_distribution(const std::vector<rep>& bl, const std::vector<rep>& weights);

  public:
    piecewise_linear_distribution() : base() {}

    template<typename InputIt1, typename InputIt2>
    piecewise_linear_distribution(InputIt1 first_i, InputIt1 last_i, InputIt2 first_w);

    template<typename UnaryOperation>
    piecewise_linear_distribution(std::initializer_list<Q> bl, UnaryOperation fw);

    template<typename UnaryOperation>
    piecewise_linear_distribution(std::size_t nw, const Q& xmin, const Q& xmax, UnaryOperation fw);

    template<typename Generator>
    Q operator()(Generator& g);

    std::vector<Q> intervals() const;

    Q min() const;
    Q max() const;
  };
  }
```

```
template<typename InputIt>
piecewise_linear_distribution(const std::vector<rep>& i, InputIt first_w);
```

1 *Effects*: Initializes the base subobject with

```
base(i.cbegin(), i.cend(), first_w)
```

```
piecewise_linear_distribution(const std::vector<rep>& bl, const std::vector<rep>& weights);
```

2 *Effects*: Initializes the base subobject with

```
base(bl.cbegin(), bl.cend(), weights.cbegin())
```

```
template<typename InputIt1, typename InputIt2>
piecewise_linear_distribution(InputIt1 first_i, InputIt1 last_i, InputIt2 first_w);
```

3 *Effects*: Equivalent to:

```
piecewise_linear_distribution(i-qty-to-rep<Q>(first_i, last_i), first_w)
```

```
template<typename UnaryOperation>
piecewise_linear_distribution(std::initializer_list<Q> bl, UnaryOperation fw);
```

4 *Effects*: Equivalent to:

```
piecewise_linear_distribution(bl-qty-to-rep(bl), fw-bl-pwl(bl, fw))
```

```
template<typename UnaryOperation>
piecewise_linear_distribution(std::size_t nw, const Q& xmin, const Q& xmax, UnaryOperation fw);
```

5 *Effects*: Initializes the base subobject with

```
base(nw, xmin.numerical_value_ref_in(Q::unit), xmax.numerical_value_ref_in(Q::unit),
     [fw](rep val) { return fw(val * Q::reference); })
```

```
std::vector<Q> intervals() const;
```

6 *Effects*: Equivalent to:

```
const std::vector<rep> intervals_rep = base::intervals();
std::vector<Q> intervals_qty;
intervals_qty.reserve(intervals_rep.size());
for (const rep& val : intervals_rep) {
  intervals_qty.push_back(val * Q::reference);
}
return intervals_qty;
```

5.7 Quantity point [qty.pt]

5.7.1 General [qty.pt.general]

1 Subclause 5.7 describes the class template `quantity_point` that represents the value of a quantity (IEC 60050, 112-01-28) that is an element of an affine space (IEC 60050, 102-03-02, IEC 60050, 102-04-01).

5.7.2 Point origin [qty.pt.orig]

5.7.2.1 General [qty.pt.orig.general]

1 This subclause specifies the components for defining the origin of an affine space. An *origin* is a point from which measurements (IEC 60050, 112-04-01) take place.

5.7.2.2 Concepts [qty.pt.orig.concepts]

```
template<typename T>
concept PointOrigin = SymbolicConstant<T> && std::derived_from<T, point-origin-interface>;

template<typename T, auto QS>
concept PointOriginFor = PointOrigin<T> && QuantitySpecOf<decltype(QS), T::quantity-spec>;

template<typename T, auto V>
concept SameAbsolutePointOriginAs =   // exposition only
  PointOrigin<T> && PointOrigin<decltype(V)> && same-absolute-point-origins(T{}, V);
```

5.7.2.3 Types [qty.pt.orig.types]

5.7.2.3.1 Absolute [qty.abs.pt.orig]

```
namespace mp_units {

template<QuantitySpec auto QS>
struct absolute_point_origin : point-origin-interface {
  static constexpr QuantitySpec auto quantity-spec = QS;   // exposition only
};

}
```

1 An *absolute origin* is an origin chosen by convention and not defined in terms of another origin. A specialization of `absolute_point_origin` is used as a base type when defining an absolute origin. `QS` is the quantity the origin represents.

5.7.2.3.2 Relative [qty.rel.pt.orig]

```
namespace mp_units {

template<QuantityPoint auto QP>
struct relative_point_origin : point-origin-interface {
  static constexpr QuantityPoint auto quantity-point = QP;        // exposition only
  static constexpr QuantitySpec auto quantity-spec = see below;   // exposition only
  static constexpr PointOrigin auto absolute-point-origin =       // exposition only
    QP.absolute_point_origin;
};

}
```

1 A *relative origin* is an origin of a subspace (IEC 60050, 102-03-03). A specialization of `relative_point_origin` is used as a base type when defining a relative origin O. O is offset from `QP.absolute_point_origin` by `QP.quantity_from_zero()`.

2 The member *`quantity-spec`* is equal to `QP.point_origin.`*`quantity-spec`* if

```
QuantityKindSpec<decltype(auto(QP.quantity-spec))>
```

is satisfied, and to `QP.`*`quantity-spec`* otherwise.

5.7.2.3.3 Zeroth [qty.zeroth.pt.orig]

```
namespace mp_units {
```

```
template<QuantitySpec auto QS>
struct zeroth_point_origin_ final : absolute_point_origin<QS> {};

}
```

1 `zeroth_point_origin_<QS>` represents an origin chosen by convention as the value 0 of the quantity `QS`.

5.7.2.4 Operations [qty.pt.orig.ops]

```
namespace mp_units {

struct point-origin-interface {
  template<PointOrigin PO, typename FwdQ,
           QuantityOf<PO::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
  friend constexpr quantity_point<Q::reference, PO{}, typename Q::rep> operator+(PO, FwdQ&& q);
  template<Quantity FwdQ, PointOrigin PO,
           QuantityOf<PO::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
  friend constexpr quantity_point<Q::reference, PO{}, typename Q::rep> operator+(FwdQ&& q, PO);

  template<PointOrigin PO, Quantity Q>
    requires ReferenceOf<decltype(auto(Q::reference)), PO::quantity-spec>
  friend constexpr QuantityPoint auto operator-(PO po, const Q& q)
    requires requires { -q; };

  template<PointOrigin PO1, SameAbsolutePointOriginAs<PO1{}> PO2>
    requires see below
  friend constexpr Quantity auto operator-(PO1 po1, PO2 po2);

  template<PointOrigin PO1, PointOrigin PO2>
  friend consteval bool operator==(PO1 po1, PO2 po2);
};

}
```

```
template<PointOrigin PO, typename FwdQ,
         QuantityOf<PO::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
friend constexpr quantity_point<Q::reference, PO{}, typename Q::rep> operator+(PO, FwdQ&& q);
template<Quantity FwdQ, PointOrigin PO,
         QuantityOf<PO::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
friend constexpr quantity_point<Q::reference, PO{}, typename Q::rep> operator+(FwdQ&& q, PO);
```

1 *Effects*: Equivalent to: `return quantity_point{std::forward<FwdQ>(q), PO{}};`

```
template<PointOrigin PO, Quantity Q>
  requires ReferenceOf<decltype(auto(Q::reference)), PO::quantity-spec>
friend constexpr QuantityPoint auto operator-(PO po, const Q& q)
  requires requires { -q; };
```

2 *Effects*: Equivalent to: `return po + -q;`

```
template<PointOrigin PO1, SameAbsolutePointOriginAs<PO1{}> PO2>
  requires see below
friend constexpr Quantity auto operator-(PO1 po1, PO2 po2);
```

3 *Effects*: Equivalent to:

```
if constexpr (is-derived-from-specialization-of<PO1, absolute_point_origin>()) {
  return po1 - po2.quantity-point;
} else if constexpr (is-derived-from-specialization-of<PO2, absolute_point_origin>()) {
  return po1.quantity-point - po2;
} else {
  return po1.quantity-point - po2.quantity-point;
}
```

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
QuantitySpecOf<decltype(auto(PO1::quantity-spec)), PO2::quantity-spec> &&
  (is-derived-from-specialization-of<PO1, relative_point_origin>() ||
   is-derived-from-specialization-of<PO2, relative_point_origin>())
```

```
template<PointOrigin PO1, PointOrigin PO2>
friend consteval bool operator==(PO1 po1, PO2 po2);
```

5 *Effects*: Equivalent to:

```
if constexpr (is-derived-from-specialization-of<PO1, absolute_point_origin>() &&
              is-derived-from-specialization-of<PO2, absolute_point_origin>())
  return std::is_same_v<PO1, PO2> ||
         (is-specialization-of<PO1, zeroth_point_origin>() &&
          is-specialization-of<PO2, zeroth_point_origin>() &&
          interconvertible(po1.quantity-spec, po2.quantity-spec));
else if constexpr (is-derived-from-specialization-of<PO1, relative_point_origin>() &&
                   is-derived-from-specialization-of<PO2, relative_point_origin>())
  return PO1::quantity-point == PO2::quantity-point;
else if constexpr (is-derived-from-specialization-of<PO1, relative_point_origin>())
  return same-absolute-point-origins(po1, po2) &&
         is_eq_zero(PO1::quantity-point.quantity_from_zero());
else if constexpr (is-derived-from-specialization-of<PO2, relative_point_origin>())
  return same-absolute-point-origins(po1, po2) &&
         is_eq_zero(PO2::quantity-point.quantity_from_zero());
```

5.7.2.5 Utilities [qty.pt.orig.utils]

5.7.2.5.1 Same absolute [qty.same.abs.pt.origs]

```
template<PointOrigin PO1, PointOrigin PO2>
consteval bool same-absolute-point-origins(PO1 po1, PO2 po2);   // exposition only
```

1 *Effects*: Equivalent to:

```
if constexpr (is-derived-from-specialization-of<PO1, absolute_point_origin>() &&
              is-derived-from-specialization-of<PO2, absolute_point_origin>())
  return po1 == po2;
else if constexpr (is-derived-from-specialization-of<PO1, relative_point_origin>() &&
                   is-derived-from-specialization-of<PO2, relative_point_origin>())
  return po1.absolute-point-origin == po2.absolute-point-origin;
else if constexpr (is-derived-from-specialization-of<PO1, relative_point_origin>())
  return po1.absolute-point-origin == po2;
else if constexpr (is-derived-from-specialization-of<PO2, relative_point_origin>())
  return po1 == po2.absolute-point-origin;
else
  return false;
```

5.7.2.5.2 Default [qty.def.pt.orig]

```
template<Reference R>
consteval PointOriginFor<get_quantity_spec(R{})> auto default_point_origin(R);
```

1 *Effects*: Equivalent to:

```
if constexpr (requires { get_unit(R{}).point-origin; })
  return get_unit(R{}).point-origin;
else
  return zeroth_point_origin<get_quantity_spec(R{})>;
```

5.7.3 Interoperability [qty.pt.like]

```
template<typename T>
concept QuantityPointLike =
  !QuantityPoint<T> &&
  qty-like-impl<T, quantity_point_like_traits> &&   // see 5.6.2
  requires {
    typename quantity_point<quantity_point_like_traits<T>::reference,
                            quantity_point_like_traits<T>::point_origin,
                            typename quantity_point_like_traits<T>::rep>;
  };
```

5.7.4 Class template quantity_point [qty.pt.syn]

```
  namespace mp_units {
```

```
template<typename T>
concept QuantityPoint = (is-derived-from-specialization-of<T, quantity_point>());

template<typename QP, auto V>
concept QuantityPointOf =
  QuantityPoint<QP> && (QuantitySpecOf<decltype(auto(QP::quantity_spec)), V> ||
                        SameAbsolutePointOriginAs<decltype(auto(QP::absolute_point_origin)), V>);

template<Reference auto R,
         PointOriginFor<get_quantity_spec(R)> auto PO = default_point_origin(R),
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity_point {
public:
  // member types and values
  static constexpr Reference auto reference = R;
  static constexpr QuantitySpec auto quantity_spec = get_quantity_spec(reference);
  static constexpr Dimension auto dimension = quantity_spec.dimension;
  static constexpr Unit auto unit = get_unit(reference);
  static constexpr PointOrigin auto absolute_point_origin = see below;
  static constexpr PointOrigin auto point_origin = PO;
  using rep = Rep;
  using quantity_type = quantity<reference, Rep>;

  quantity_type quantity-from-origin;   // exposition only

  // 5.7.5, static member functions
  static constexpr quantity_point min() noexcept
    requires see below;
  static constexpr quantity_point max() noexcept
    requires see below;

  // 5.7.6, constructors and assignment

  quantity_point() = default;
  quantity_point(const quantity_point&) = default;
  quantity_point(quantity_point&&) = default;
  ~quantity_point() = default;

  template<typename FwdQ, QuantityOf<quantity_spec> Q = std::remove_cvref_t<FwdQ>>
    requires std::constructible_from<quantity_type, FwdQ> &&
             (point_origin == default_point_origin(R)) &&
             (implicitly_convertible(Q::quantity_spec, quantity_spec))
  constexpr explicit quantity_point(FwdQ&& q);

  template<typename FwdQ, QuantityOf<quantity_spec> Q = std::remove_cvref_t<FwdQ>>
    requires std::constructible_from<quantity_type, FwdQ>
  constexpr quantity_point(FwdQ&& q, decltype(PO));

  template<typename FwdQ, PointOrigin PO2,
           QuantityOf<PO2::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
    requires std::constructible_from<quantity_type, FwdQ> && SameAbsolutePointOriginAs<PO2, PO>
  constexpr quantity_point(FwdQ&& q, PO2);

  template<QuantityPointOf<absolute_point_origin> QP>
    requires std::constructible_from<quantity_type, typename QP::quantity_type>
  constexpr explicit(!std::convertible_to<typename QP::quantity_type, quantity_type>)
    quantity_point(const QP& qp);

  template<QuantityPointLike QP>
    requires see below
  constexpr explicit(see below) quantity_point(const QP& qp);

  quantity_point& operator=(const quantity_point&) = default;
  quantity_point& operator=(quantity_point&&) = default;
```

```
// 5.7.7, conversions

template<SameAbsolutePointOriginAs<absolute_point_origin> NewPO>
constexpr QuantityPointOf<(NewPO{})> auto point_for(NewPO new_origin) const;

template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in(ToU) const;

template<RepresentationOf<quantity_spec> ToRep>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in(ToU) const;

template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in(ToU) const;

template<RepresentationOf<quantity_spec> ToRep>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in(ToU) const;

// 5.7.8, quantity value observers

template<PointOrigin PO2>
  requires(PO2{} == point_origin)
constexpr quantity_type& quantity_ref_from(PO2) & noexcept;
template<PointOrigin PO2>
  requires(PO2{} == point_origin)
constexpr const quantity_type& quantity_ref_from(PO2) const & noexcept;
template<PointOrigin PO2>
  requires(PO2{} == point_origin)
void quantity_ref_from(PO2) const && = delete;

template<PointOrigin PO2>
  requires requires(const quantity_point qp) { qp - PO2{}; }
constexpr Quantity auto quantity_from(PO2) const;

template<QuantityPointOf<absolute_point_origin> QP>
constexpr Quantity auto quantity_from(const QP&) const;

constexpr Quantity auto quantity_from_zero() const;

// 5.7.9, conversion operations
template<typename QP_, QuantityPointLike QP = std::remove_cvref_t<QP_>>
  requires see below
constexpr explicit(see below) operator QP_() const & noexcept(see below);
template<typename QP_, QuantityPointLike QP = std::remove_cvref_t<QP_>>
  requires see below
constexpr explicit(see below) operator QP_() && noexcept(see below);

// 5.7.10, unary operations
```

```
  template<Mutable<quantity_point> QP>
  friend constexpr decltype(auto) operator++(QP&& qp)
    requires see below;
  template<Mutable<quantity_point> QP>
  friend constexpr decltype(auto) operator--(QP&& qp)
    requires see below;

  constexpr quantity_point operator++(int)
    requires see below;
  constexpr quantity_point operator--(int)
    requires see below;

  // 5.7.11, compound assignment operations
  template<Mutable<quantity_point> QP, auto R2, typename Rep2>
    requires see below
  friend constexpr decltype(auto) operator+=(QP&& qp, const quantity<R2, Rep2>& q);
  template<Mutable<quantity_point> QP, auto R2, typename Rep2>
    requires see below
  friend constexpr decltype(auto) operator-=(QP&& qp, const quantity<R2, Rep2>& q);

  // 5.7.12, arithmetic operations

  template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
  friend constexpr QuantityPoint auto operator+(const QP& qp, const quantity<R2, Rep2>& q)
    requires see below;
  template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
  friend constexpr QuantityPoint auto operator+(const quantity<R2, Rep2>& q, const QP& qp)
    requires see below;
  template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
  friend constexpr QuantityPoint auto operator-(const QP& qp, const quantity<R2, Rep2>& q)
    requires see below;

  template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
  friend constexpr Quantity auto operator-(const QP& lhs, const QP2& rhs)
    requires see below;

  template<std::derived_from<quantity_point> QP, PointOrigin PO2>
    requires see below
  friend constexpr Quantity auto operator-(const QP& qp, PO2 po);
  template<std::derived_from<quantity_point> QP, PointOrigin PO2>
    requires see below
  friend constexpr Quantity auto operator-(PO2 po, const QP& qp);

  // 5.7.13, comparison
  template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
    requires see below
  friend constexpr bool operator==(const QP& lhs, const QP2& rhs);
  template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
    requires see below
  friend constexpr auto operator<=>(const QP& lhs, const QP2& rhs);
};

template<Quantity Q>
explicit quantity_point(Q q)
  -> quantity_point<Q::reference, default_point_origin(Q::reference), typename Q::rep>;

template<Quantity Q, PointOriginFor<Q::quantity_spec> PO>
quantity_point(Q, PO) -> quantity_point<Q::reference, PO{}, typename Q::rep>;

template<QuantityPointLike QP, typename Traits = quantity_point_like_traits<QP>>
explicit(quantity_point_like_traits<QP>::explicit_import) quantity_point(QP)
  -> quantity_point<Traits::reference, Traits::point_origin, typename Traits::rep>;
```

```
}
```

1 `quantity_point<R, PO, Rep>` is a structural type (N4971, [temp.param]) if `Rep` is a structural type.

2 The member `absolute_point_origin` is equal to `PO` if

```
is-derived-from-specialization-of<decltype(PO), absolute_point_origin>()
```

is `true`, and to `PO.`*`quantity-point`*`.absolute_point_origin` otherwise.

5.7.5 Static member functions [qty.pt.static]

```
static constexpr quantity_point min() noexcept
  requires see below;
static constexpr quantity_point max() noexcept
  requires see below;
```

1 Let *F* be one of `min` and `max`.

2 *Returns*: `{quantity_type::`*`F`*`(), PO}`.

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { quantity_type::F(); }
```

5.7.6 Constructors [qty.pt.cons]

```
template<typename FwdQ, QuantityOf<quantity_spec> Q = std::remove_cvref_t<FwdQ>>
  requires std::constructible_from<quantity_type, FwdQ> &&
           (point_origin == default_point_origin(R)) &&
           (implicitly_convertible(Q::quantity_spec, quantity_spec))
constexpr explicit quantity_point(FwdQ&& q);

template<typename FwdQ, QuantityOf<quantity_spec> Q = std::remove_cvref_t<FwdQ>>
  requires std::constructible_from<quantity_type, FwdQ>
constexpr quantity_point(FwdQ&& q, decltype(PO));
```

1 *Effects*: Initializes *`quantity-from-origin`* with `std::forward<FwdQ>(q)`.

```
template<typename FwdQ, PointOrigin PO2,
         QuantityOf<PO2::quantity-spec> Q = std::remove_cvref_t<FwdQ>>
  requires std::constructible_from<quantity_type, FwdQ> && SameAbsolutePointOriginAs<PO2, PO>
constexpr quantity_point(FwdQ&& q, PO2);
```

2 *Effects*: Equivalent to:

```
quantity_point(quantity_point<Q::reference, PO2{}, typename Q::rep>{std::forward<FwdQ>(q),
                                                                     PO2{}})
```

```
template<QuantityPointOf<absolute_point_origin> QP>
  requires std::constructible_from<quantity_type, typename QP::quantity_type>
constexpr explicit(!std::convertible_to<typename QP::quantity_type, quantity_type>)
  quantity_point(const QP& qp);
```

3 *Effects*: If `point_origin == QP::point_origin` is `true`, initializes *`quantity-from-origin`* with `qp.quantity_ref_from(point_origin)`. Otherwise, initializes *`quantity-from-origin`* with `qp - point_origin`.

```
template<QuantityPointLike QP>
  requires see below
constexpr explicit(see below) quantity_point(const QP& qp);
```

4 Let `Traits` be `quantity_point_like_traits<QP>`.

5 *Effects*: Initializes *`quantity-from-origin`* with

```
Traits::to_numerical_value(qp), get_unit(Traits::reference)
```

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(Traits::point_origin == point_origin) &&
  std::convertible_to<quantity<Traits::reference, typename Traits::rep>, quantity_type>
```

The expression inside `explicit` is equivalent to:

```
    Traits::explicit_import ||
      !std::convertible_to<quantity<Traits::reference, typename Traits::rep>, quantity_type>
```

5.7.7 Conversions [qty.pt.conv]

```
template<SameAbsolutePointOriginAs<absolute_point_origin> NewPO>
constexpr QuantityPointOf<(NewPO{})> auto point_for(NewPO new_origin) const;
```

1 *Effects*: Equivalent to:

```
if constexpr (std::is_same_v<NewPO, decltype(point_origin)>)
  return *this;
else
  return ::mp_units::quantity_point{*this - new_origin, new_origin};
```

```
template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in(ToU) const;

template<RepresentationOf<quantity_spec> ToRep>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto in(ToU) const;

template<UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in(ToU) const;

template<RepresentationOf<quantity_spec> ToRep>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in() const;

template<RepresentationOf<quantity_spec> ToRep,
         UnitCompatibleWith<unit, quantity_spec> ToU>
  requires see below
constexpr QuantityPointOf<quantity_spec> auto force_in(ToU) const;
```

2 Let *converted-quantity-expr* be an expression denoting the function call to the corresponding member of `quantity_ref_from(point_origin)`.

3 *Effects*: Equivalent to:

```
return ::mp_units::quantity_point{converted-quantity-expr, point_origin};
```

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { converted-quantity-expr; }
```

5.7.8 Quantity value observers [qty.pt.obs]

```
template<PointOrigin PO2>
  requires(PO2{} == point_origin)
constexpr quantity_type& quantity_ref_from(PO2) & noexcept;
template<PointOrigin PO2>
  requires(PO2{} == point_origin)
constexpr const quantity_type& quantity_ref_from(PO2) const & noexcept;
```

1 *Returns*: *quantity-from-origin*.

```
template<PointOrigin PO2>
  requires requires(const quantity_point qp) { qp - PO2{}; }
constexpr Quantity auto quantity_from(PO2 rhs) const;
```

```
template<QuantityPointOf<absolute_point_origin> QP>
constexpr Quantity auto quantity_from(const QP& rhs) const;
```

2 *Effects*: Equivalent to: `return *this - rhs;`

```
constexpr Quantity auto quantity_from_zero() const;
```

3 *Effects*: Equivalent to:

```
if constexpr (requires { unit.point-origin; }) {
  // can lose the input unit
  const auto q = quantity_from(unit.point-origin);
  if constexpr (requires { q.in(unit); })
    // restore the unit
    return q.in(unit);
  else
    return q;
} else
  return quantity_from(absolute_point_origin);
```

5.7.9 Conversion operations [qty.pt.conv.ops]

```
template<typename QP_, QuantityPointLike QP = std::remove_cvref_t<QP_>>
  requires see below
constexpr explicit(see below) operator QP_() const & noexcept(see below);
template<typename QP_, QuantityPointLike QP = std::remove_cvref_t<QP_>>
  requires see below
constexpr explicit(see below) operator QP_() && noexcept(see below);
```

1 Let `Traits` be `quantity_point_like_traits<QP>`. Let *`result-expr`* be

```
Traits::from_numerical_value(std::move(quantity-from-origin).numerical-value)
```

2 *Returns*: *`result-expr`*.

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(point_origin == Traits::point_origin) &&
  std::convertible_to<quantity_type, quantity<Traits::reference, typename Traits::rep>>
```

The expression inside `explicit` is equivalent to:

```
Traits::explicit_export ||
  !std::convertible_to<quantity_type, quantity<Traits::reference, typename Traits::rep>>
```

Let T be `std::is_nothrow_copy_constructible_v` for the first signature, and `std::is_nothrow_move_constructible_v` for the second signature. The exception specification is equivalent to:

```
noexcept(result-expr) && T<rep>
```

5.7.10 Unary operations [qty.pt.unary.ops]

1 In the following descriptions, let `@` be the *operator*.

```
template<Mutable<quantity_point> QP>
friend constexpr decltype(auto) operator++(QP&& qp)
  requires see below;
template<Mutable<quantity_point> QP>
friend constexpr decltype(auto) operator--(QP&& qp)
  requires see below;
```

2 *Effects*: Equivalent to `@qp.`*`quantity-from-origin`*.

3 *Returns*: `std::forward<QP>(qp)`.

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { @qp.quantity-from-origin; }
```

```
constexpr quantity_point operator++(int)
  requires see below;
constexpr quantity_point operator--(int)
  requires see below;
```

5 *Effects*: Equivalent to: `return {`*`quantity-from-origin`*`@, PO};`

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { quantity-from-origin@; }
```

5.7.11 Compound assignment operations [qty.pt.assign.ops]

```
template<Mutable<quantity_point> QP, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator+=(QP&& qp, const quantity<R2, Rep2>& q);
template<Mutable<quantity_point> QP, auto R2, typename Rep2>
  requires see below
friend constexpr decltype(auto) operator-=(QP&& qp, const quantity<R2, Rep2>& q);
```

1 Let `@` be the *operator*.

2 *Effects*: Equivalent to `qp.quantity-from-origin @ q`.

3 *Returns*: `std::forward<QP>(qp)`.

4 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
QuantityConvertibleTo<quantity<R2, Rep2>, quantity_type> &&
  requires { qp.quantity-from-origin @ q; }
```

5.7.12 Arithmetic operations [qty.pt.arith.ops]

1 In the following descriptions, let `@` be the *operator*.

```
template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
friend constexpr QuantityPoint auto operator+(const QP& qp, const quantity<R2, Rep2>& q)
  requires see below;
template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
friend constexpr QuantityPoint auto operator+(const quantity<R2, Rep2>& q, const QP& qp)
  requires see below;
template<std::derived_from<quantity_point> QP, auto R2, typename Rep2>
friend constexpr QuantityPoint auto operator-(const QP& qp, const quantity<R2, Rep2>& q)
  requires see below;
```

2 *Effects*: Equivalent to:

```
if constexpr (is-specialization-of<PO, zeroth_point_origin>())
  return ::mp_units::quantity_point{qp.quantity_ref_from(PO) @ q};
else
  return ::mp_units::quantity_point{qp.quantity_ref_from(PO) @ q, PO};
```

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
ReferenceOf<decltype(R2), PO.quantity-spec> && requires {
  qp.quantity_ref_from(PO) @ q;
}
```

```
template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
friend constexpr Quantity auto operator-(const QP& lhs, const QP2& rhs)
  requires see below;
```

4 *Effects*: Equivalent to:

```
return lhs.quantity_ref_from(point_origin) - rhs.quantity_ref_from(QP2::point_origin) +
       (lhs.point_origin - rhs.point_origin);
```

5 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
requires { lhs.quantity_ref_from(point_origin) - rhs.quantity_ref_from(QP2::point_origin); }
```

6 *Recommended practice*: The subtraction of two equal origins is not evaluated.

```
template<std::derived_from<quantity_point> QP, PointOrigin PO2>
  requires see below
friend constexpr Quantity auto operator-(const QP& qp, PO2 po);
template<std::derived_from<quantity_point> QP, PointOrigin PO2>
  requires see below
friend constexpr Quantity auto operator-(PO2 po, const QP& qp);
```

7 *Effects*: For the first signature, equivalent to:

```
  if constexpr (point_origin == po)
    return qp.quantity_ref_from(point_origin);
  else if constexpr (is-derived-from-specialization-of<PO2,
                                                 ::mp_units::absolute_point_origin>()) {
    return qp.quantity_ref_from(point_origin) + (qp.point_origin - qp.absolute_point_origin);
  } else {
    return qp.quantity_ref_from(point_origin) -
           po.quantity-point.quantity_ref_from(po.quantity-point.point_origin) +
           (qp.point_origin - po.quantity-point.point_origin);
  }
```

For the second signature, equivalent to: `return -(qp - po);`

8 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
QuantityPointOf<quantity_point, PO2{}> &&
  ReferenceOf<decltype(auto(reference)), PO2::quantity-spec>
```

9 *Recommended practice*: The subtraction of two equal origins is not evaluated.

5.7.13 Comparison [qty.pt.cmp]

```
template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
  requires see below
friend constexpr bool operator==(const QP& lhs, const QP2& rhs);
template<std::derived_from<quantity_point> QP, QuantityPointOf<absolute_point_origin> QP2>
  requires see below
friend constexpr auto operator<=>(const QP& lhs, const QP2& rhs);
```

1 Let `@` be the *operator*, and let C be `std::equality_comparable_with` if `@` is `==`, and `std::three_way_comparable_with` if `@` is `<=>`.

2 *Effects*: Equivalent to:

```
return lhs - lhs.absolute_point_origin @ rhs - rhs.absolute_point_origin;
```

3 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
C<quantity_type, typename QP2::quantity_type>
```

4 *Recommended practice*: If the origins are equal, instead evaluate

```
lhs.quantity_ref_from(point_origin) @ rhs.quantity_ref_from(QP2::point_origin)
```

5.7.14 Construction helper point [qty.point]

```
namespace mp_units {

template<Reference R>
struct point_ {
  template<typename FwdRep,
           RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
  constexpr quantity_point<R{}, default_point_origin(R{}), Rep> operator()(FwdRep&& lhs) const;
};

}
```

```
template<typename FwdRep,
         RepresentationOf<get_quantity_spec(R{})> Rep = std::remove_cvref_t<FwdRep>>
constexpr quantity_point<R{}, default_point_origin(R{}), Rep> operator()(FwdRep&& lhs) const;
```

1 *Effects*: Equivalent to: `return quantity_point{quantity{std::forward<FwdRep>(lhs), R{}}};`

5.7.15 Non-member conversions [qty.pt.non.mem.conv]

```
template<QuantityPoint ToQP, typename FwdFromQP,
         QuantityPoint FromQP = std::remove_cvref_t<FwdFromQP>>
  requires see below
constexpr QuantityPoint auto sudo-cast(FwdFromQP&& qp);
```

1 *Returns*: TBD.

2 `value_cast` is an explicit cast that allows truncation.

```
template<Unit auto ToU, typename FwdQP, QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires(convertible(QP::reference, ToU))
constexpr QuantityPoint auto value_cast(FwdQP&& qp);
```

3 *Effects*: Equivalent to:

```
return quantity_point{value_cast<ToU>(std::forward<FwdQP>(qp).quantity-from-origin),
                      QP::point_origin};
```

```
template<Representation ToRep, typename FwdQP, QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires RepresentationOf<ToRep, QP::quantity_spec> &&
           std::constructible_from<ToRep, typename QP::rep>
constexpr quantity_point<QP::reference, QP::point_origin, ToRep> value_cast(FwdQP&& qp);
```

4 *Effects*: Equivalent to:

```
return {value_cast<ToRep>(std::forward<FwdQP>(qp).quantity-from-origin), QP::point_origin};
```

```
template<Unit auto ToU, Representation ToRep, typename FwdQP,
         QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires see below
constexpr QuantityPoint auto value_cast(FwdQP&& qp);
template<Representation ToRep, Unit auto ToU, typename FwdQP,
         QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires see below
constexpr QuantityPoint auto value_cast(FwdQP&& qp);
```

5 *Effects*: Equivalent to:

```
return quantity_point{value_cast<ToU, ToRep>(std::forward<FwdQP>(qp).quantity-from-origin),
                      QP::point_origin};
```

6 *Remarks*: The expression in the *requires-clause* is equivalent to:

```
(convertible(QP::reference, ToU)) && RepresentationOf<ToRep, QP::quantity_spec> &&
std::constructible_from<ToRep, typename QP::rep>
```

```
template<Quantity ToQ, typename FwdQP, QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires(convertible(QP::reference, ToQ::unit)) && (ToQ::quantity_spec == QP::quantity_spec) &&
          std::constructible_from<typename ToQ::rep, typename QP::rep>
constexpr QuantityPoint auto value_cast(FwdQP&& qp);
```

7 *Effects*: Equivalent to:

```
return quantity_point{value_cast<ToQ>(std::forward<FwdQP>(qp).quantity-from-origin),
                      QP::point_origin};
```

```
template<QuantityPoint ToQP, typename FwdQP, QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires(convertible(QP::reference, ToQP::unit)) &&
          (ToQP::quantity_spec == QP::quantity_spec) &&
          (same-absolute-point-origins(ToQP::point_origin, QP::point_origin)) &&
          std::constructible_from<typename ToQP::rep, typename QP::rep>
constexpr QuantityPoint auto value_cast(FwdQP&& qp);
```

8 *Effects*: Equivalent to: `return sudo-cast<ToQP>(std::forward<FwdQP>(qp));`

9 `quantity_cast` is an explicit cast that allows converting to more specific quantities.

```
template<QuantitySpec auto ToQS, typename FwdQP, QuantityPoint QP = std::remove_cvref_t<FwdQP>>
  requires QuantitySpecCastableTo<QP::quantity_spec, ToQS>
constexpr QuantityPoint auto quantity_cast(FwdQP&& qp);
```

10 *Effects*: Equivalent to:

```
return QP{quantity_cast<ToQS>(std::forward<FwdQP>(qp).quantity_from_origin),
          QP::point_origin};
```

5.7.16 Math [qty.pt.math]

```
template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isfinite(q); }
```

```
constexpr bool isfinite(const quantity_point<R, PO, Rep>& a) noexcept;                // hosted
```

1 *Effects*: Equivalent to: `return isfinite(a.quantity_ref_from(a.point_origin));`

```
template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isinf(q); }
constexpr bool isinf(const quantity_point<R, PO, Rep>& a) noexcept;                   // hosted
```

2 *Effects*: Equivalent to: `return isinf(a.quantity_ref_from(a.point_origin));`

```
template<auto R, auto PO, typename Rep>
  requires requires(quantity<R, Rep> q) { isnan(q); }
constexpr bool isnan(const quantity_point<R, PO, Rep>& a) noexcept;                   // hosted
```

3 *Effects*: Equivalent to: `return isnan(a.quantity_ref_from(a.point_origin));`

```
template<auto R, auto S, auto T, auto Origin, typename Rep1, typename Rep2, typename Rep3>
  requires requires {
    get_common_quantity_spec(get_quantity_spec(R) * get_quantity_spec(S), get_quantity_spec(T));
  } && (equivalent(get_unit(R) * get_unit(S), get_unit(T))) &&
           requires(Rep1 v1, Rep2 v2, Rep3 v3) {
             requires requires { fma(v1, v2, v3); } || requires { std::fma(v1, v2, v3); };
           }
constexpr QuantityPointOf<get_common_quantity_spec(get_quantity_spec(R) * get_quantity_spec(S),
                                                    get_quantity_spec(T))> auto
fma(const quantity<R, Rep1>& a, const quantity<S, Rep2>& x,                           // hosted
    const quantity_point<T, Origin, Rep3>& b) noexcept;
```

4 *Effects*: Equivalent to:

```
using std::fma;
return Origin +
       quantity{fma(a.numerical_value_ref_in(a.unit), x.numerical_value_ref_in(x.unit),
                    b.quantity_ref_from(b.point_origin).numerical_value_ref_in(b.unit)),
                get_common_reference(R* S, T)};
```

5.8 Systems [qty.systems]

5.9 `std::chrono` interoperability [qty.chrono]

```
namespace mp_units {

template<typename Period>
consteval auto time-unit-from-chrono-period()
{
  using namespace si;

  if constexpr (is_same_v<Period, std::chrono::nanoseconds::period>)
    return nano<second>;
  else if constexpr (is_same_v<Period, std::chrono::microseconds::period>)
    return micro<second>;
  else if constexpr (is_same_v<Period, std::chrono::milliseconds::period>)
    return milli<second>;
  else if constexpr (is_same_v<Period, std::chrono::seconds::period>)
    return second;
  else if constexpr (is_same_v<Period, std::chrono::minutes::period>)
    return minute;
  else if constexpr (is_same_v<Period, std::chrono::hours::period>)
    return hour;
  else if constexpr (is_same_v<Period, std::chrono::days::period>)
    return day;
  else if constexpr (is_same_v<Period, std::chrono::weeks::period>)
    return mag<7> * day;
  else
    return mag_ratio<Period::num, Period::den> * second;
}
template<typename Rep, typename Period>
struct quantity_like_traits<std::chrono::duration<Rep, Period>> {
  static constexpr auto reference = time-unit-from-chrono-period<Period>();
  static constexpr bool explicit_import = false;
  static constexpr bool explicit_export = false;
  using rep = Rep;
  using T = std::chrono::duration<Rep, Period>;

  static constexpr rep to_numerical_value(const T& q) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return q.count();
  }

  static constexpr T from_numerical_value(const rep& v) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return T(v);
  }
};

template<typename Clock>
struct chrono_point_origin_ final : absolute_point_origin<isq::time> {
  using clock = Clock;
};

template<typename Clock, typename Rep, typename Period>
struct quantity_point_like_traits<
  std::chrono::time_point<Clock, std::chrono::duration<Rep, Period>>> {
  static constexpr auto reference = time-unit-from-chrono-period<Period>();
  static constexpr auto point_origin = chrono_point_origin<Clock>;
  static constexpr bool explicit_import = false;
  static constexpr bool explicit_export = false;
  using rep = Rep;
  using T = std::chrono::time_point<Clock, std::chrono::duration<Rep, Period>>;

  static constexpr rep to_numerical_value(const T& tp) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return tp.time_since_epoch().count();
  }

  static constexpr T from_numerical_value(const rep& v) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return T(std::chrono::duration<Rep, Period>(v));
  }
};

}
```

Index

Constructions whose name appears in *`monospaced italics`* are for exposition only.

Index of library modules

The bold page number for each entry refers to the page where the synopsis of the module is shown.

Index of library names

Constructions whose name appears in *italics* are for exposition only.

Index of library concepts

The bold page number for each entry is the page where the concept is defined. Other page numbers refer to pages where the concept is mentioned in the general text. Concepts whose name appears in *italics* are for exposition only.

Index of implementation-defined behavior

The entries in this index are rough descriptions; exact specifications are at the indicated page in the general text.