Ref:	*unspecified*
Date:	2024-10-18
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 formatting.

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 "[qties]" 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. P2900R10: *Contracts for C++*. Edited by Joshua Berne. Available from: https://wg21.link/P2900R10

(1.6) — The C++ Standards Committee. P2996R7: *Reflection for C++26*. Edited by Barry Revzin. Available from: https://wg21.link/P2996R7

(1.7) — 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 library [qties]

5.1 Summary [qties.summary]

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

Table 3 — Quantities library summary [tab:qties.summary]

	Subclause	Module
5.5	Helpers	`mp_units.core`
5.6	Representation	
5.7	Expression template	
5.8	Dimension	
5.9	Quantity specification	
5.10	Magnitude	
5.11	Unit	
5.12	Reference	
5.13	Quantity types	
5.14	Systems	`mp_units.systems`
5.15	`std::chrono` compatibility	

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

```
export module mp_units;

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

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

```
export module mp_units.core;

import std;

export namespace mp_units {

enum class text_encoding : std::int8_t { utf8, portable, default_encoding = utf8 };

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

// 5.5, helpers

// 5.5.4, class template symbol_text

// 5.6, representation

// 5.6.1, traits

// 5.6.1.1, floating-point

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

// 5.6.1.2, set

template<typename Rep>
constexpr bool is_scalar = see below;
```

```
template<typename Rep>
constexpr bool is_complex = false;

template<typename Rep>
constexpr bool is_vector = false;

template<typename Rep>
constexpr bool is_tensor = false;

// 5.6.2, concepts

template<typename T>
concept Representation = see below;

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

// 5.7, expression template

// 5.7.2, types

template<typename... Ts>
struct type_list;

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

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

// 5.7.3, algorithms

// 5.8, dimension

// 5.8.2, concepts

template<typename T>
concept Dimension = see below;

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

// 5.8.3, types

template<symbol_text Symbol>
struct base_dimension;

template<see below>
struct derived_dimension;

struct dimension_one;
inline constexpr dimension_one dimension_one = see below;

consteval Dimension auto inverse(Dimension auto d) { return dimension_one / 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) { return pow<1, 2>(d); }
consteval Dimension auto cbrt(Dimension auto d) { return pow<1, 3>(d); }
```

```
struct dimension_symbol_formatting {
  text_encoding encoding = text_encoding::default_encoding;
};

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>
constexpr auto dimension_symbol(D);

// 5.9, quantity specification

// 5.9.2, concepts

template<typename T>
concept QuantitySpec = see below;

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

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

// 5.9.3, types

struct is_kind;
inline constexpr is_kind is_kind = see below;

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

template<BaseDimension auto Dim, auto... Args>
  requires see below
struct quantity_spec<Dim, Args...>;

template<DerivedQuantitySpec auto Eq, auto... Args>
  requires see below
struct quantity_spec<Eq, Args...>;

template<NamedQuantitySpec auto QS, auto... Args>
  requires see below
struct quantity_spec<QS, Args...>;

template<NamedQuantitySpec auto QS, DerivedQuantitySpec auto Eq, auto... Args>
  requires see below
struct quantity_spec<QS, Eq, Args...>;

template<DerivedQuantitySpecExpr... Expr>
struct derived_quantity_spec;

struct dimensionless;
inline constexpr dimensionless dimensionless = see below;

template<typename Q>
struct kind_of_;  // not defined
template<typename Q>
  requires see below
struct kind_of_<Q>;
template<auto Q>
  requires requires { typename kind_of_<decltype(Q)>; }
inline constexpr kind_of_<Q> kind_of{};

consteval QuantitySpec auto inverse(QuantitySpec auto q) { return dimensionless / 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) { return pow<1, 2>(q); }
consteval QuantitySpec auto cbrt(QuantitySpec auto q) { return pow<1, 3>(q); }

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);

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

consteval QuantitySpec auto get_common_quantity_spec(QuantitySpec auto q);
template<QuantitySpec Q1, QuantitySpec Q2>
  requires see below
consteval QuantitySpec auto get_common_quantity_spec(Q1 q1, Q2 q2);
consteval QuantitySpec auto get_common_quantity_spec(QuantitySpec auto q1, QuantitySpec auto q2,
                                                     QuantitySpec auto q3,
                                                     QuantitySpec auto... rest)
  requires see below;

// 5.10, magnitude

// 5.11, unit

// 5.12, reference

// 5.13, quantity types

// 5.13.1, traits

// 5.13.1.1, values

template<typename Rep>
struct quantity_values;

// 5.13.1.2, compatibility

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

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

template<typename T>
concept quantity_like = see below;

template<typename T>
concept quantity_point_like = see below;

// 5.13.3, quantity concepts

// 5.13.4, class template quantity
template<Reference auto R, RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity;

// 5.13.5, quantity point concepts

// 5.13.6, class template quantity_point
template<unspecified>
class quantity_point;
```

```
}
```

5.4 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 {

// 5.15, std::chrono compatibility

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>>>;

}
```

5.5 Helpers [qty.helpers]

5.5.1 Non-types [qty.helpers.non.types]

```
template<typename T>
concept tag-type = type_is_empty(^T) && type_is_final(^T);  // exposition only
```

```
consteval bool is-specialization-of(  // exposition only
  std::meta::info type, std::meta::info template_name);
```

1 *Preconditions*: `is_type(type) && is_class_template(template_name)` is `true`.

2 *Returns*: `has_template_arguments(type) && template_of(type) == template_name`.

```
consteval bool is-convertible-to-base-subobject-of(  // exposition only
  std::meta::info type, std::meta::info template_name);
```

3 *Preconditions*: `is_type(type) && is_class_template(template_name)` is `true`.

4 *Returns*: `true` if `[:type:]` has an unambiguous and accessible base that is a specialization of `[:template_name:]`, and `false` otherwise.

5.5.2 Struct *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 ratio{-r.num, r.den}; }

  friend consteval ratio operator+(ratio lhs, ratio rhs)
  {
    return ratio{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)
  {
    if (r1.num == r2.num && r1.den == r2.den) return r1;

    // gcd(a/b, c/d) = gcd(ad, cb)/bd
    contract_assert(std::numeric_limits<std::intmax_t>::max() / r1.num > r2.den);
    contract_assert(std::numeric_limits<std::intmax_t>::max() / r2.num > r1.den);
    contract_assert(std::numeric_limits<std::intmax_t>::max() / r1.den > r2.den);

    const std::intmax_t num = std::gcd(r1.num * r2.den, r2.num * r1.den);
    const std::intmax_t den = r1.den * r2.den;
    const std::intmax_t gcd = std::gcd(num, den);
    return ratio{num / gcd, den / gcd};
  }

  }

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

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

2 *Effects*: Equivalent to: `R`.

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

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

4 Let `R(r)` be `std::ratio<N, D>`, where `N` and `D` are the values of `r.num` and `r.den`. Let `Res` be `std::ratio_multiply<R(lhs), R(rhs)>`.

5 *Effects*: Equivalent to: `return ratio{Res::num, Res::den}`.

5.5.3 Class template *basic-fixed-string* [qty.fixed.string]

5.5.4 Class template symbol_text [qty.symbol.text]

5.6 Representation [qty.rep]

5.6.1 Traits [qty.rep.traits]

5.6.1.1 Floating-point [qty.fp.traits]

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

template<typename T>
  requires(!type_is_pointer(^T) && !type_is_array(^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<Rep> ||
  std::chrono::treat_as_floating_point_v<actual-value-type-t<Rep>>;
```

1 The quantity types (5.13.2) 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`.

5.6.1.2 Set [qty.set.traits]

```
template<typename Rep>
constexpr bool is_scalar =
  type_is_floating_point(^Rep) || (type_is_integral(^Rep) && ^Rep != ^bool);

template<typename Rep>
constexpr bool is_complex = false;

template<typename Rep>
constexpr bool is_vector = false;

template<typename Rep>
constexpr bool is_tensor = 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). A quantity type (5.13.2) `Q` uses these traits to determine the set `Q::rep` should represent.

2 *Remarks*: Pursuant to N4971, [namespace.std] (4.1), users may specialize `is_scalar`, `is_complex`, `is_vector`, and `is_tensor` to `true` for cv-unqualified program-defined types which respectively represent a scalar (IEC 60050, 102-02-18), a complex number (IEC 60050, 102-02-09), a vector (IEC 60050, 102-03-04), and a tensor, and `false` for types which respectively do not. Such specializations shall be usable in constant expressions (N4971, [expr.const]) and have type `const bool`.

5.6.2 Concepts [qty.rep.concepts]

```
template<typename T, typename U>
concept CommonTypeWith = std::same_as<std::common_type_t<T, U>, std::common_type_t<U, T>> &&
                         std::constructible_from<std::common_type_t<T, U>, T> &&
                         std::constructible_from<std::common_type_t<T, U>, U>;

template<typename T, typename U = T>
concept ScalableNumber = std::regular_invocable<std::multiplies<>, T, U> &&
                         std::regular_invocable<std::divides<>, T, U>;

template<typename T>
concept CastableNumber =
  CommonTypeWith<T, std::intmax_t> && ScalableNumber<std::common_type_t<T, std::intmax_t>>;

template<typename T>
concept Scalable =
  CastableNumber<T> ||
  (CastableNumber<actual-value-type-t<T>> &&
   ScalableNumber<T, std::common_type_t<actual-value-type-t<T>, std::intmax_t>>);

template<typename T>
concept WeaklyRegular = std::copyable<T> && std::equality_comparable<T>;

template<typename T>
concept Representation = (is_scalar<T> || is_complex<T> || is_vector<T> || is_tensor<T>) &&
                         WeaklyRegular<T> && Scalable<T>;

template<typename T, quantity_character Ch>
concept RepresentationOf =
  Representation<T> && ((Ch == quantity_character::scalar && is_scalar<T>) ||
                        (Ch == quantity_character::complex && is_complex<T>) ||
                        (Ch == quantity_character::vector && is_vector<T>) ||
                        (Ch == quantity_character::tensor && is_tensor<T>));
```

5.7 Expression template [qty.expr.temp]

5.7.1 General [qty.expr.temp.general]

1 Subclause 5.7 specifies the components used to maintain an ordered, simplified, and readable argument lists in the names of specializations.

[*Example 1*: The framework ensures the following assertion holds.

```
using namespace si::unit_symbols;
static_assert(^decltype(km / square(h)) ==
              ^derived_unit<si::kilo_<si::metre>, per<power<non_si::hour, 2>>>);
```

— *end example*]

5.7.2 Types [qty.expr.temp.types]

```
namespace mp_units {

template<typename... Ts>
struct type_list {};

}
```

1 `type_list` encapsulates a list of types.

```
namespace mp_units {

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

}
```

2 `per` stores the arguments with negative exponents.

```
namespace mp_units {

template<typename F, int Num, int... Den>
  requires(sizeof...(Den) <= 1 && valid-ratio(Num, Den...) && positive-ratio(Num, Den...) &&
           !ratio-one(Num, Den...))
struct power {
  using factor = F;
  static constexpr ratio exponent{Num, Den...};
};

}
```

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

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

5.7.3 Algorithms [qty.expr.temp.algo]

1 Unless otherwise specified, in the following descriptions, let

(1.1) — `To` be the template of the resulting type list,

(1.2) — `OneType` be the neutral element of the operation, and

(1.3) — `Pred` be *`type-less`*.

```
consteval std::meta::info expr-type(std::meta::info t)
{
  return is-specialization-of(t, ^power) ? template_arguments_of(t)[0] : t;
}

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

```
  template<typename OneType, typename... Ts>
  struct expr-fractions;
```

2 *expr-fractions* divides an expression template to numerator and denominator parts.

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

3 *expr-multiply* multiplies two sorted expression templates.

4 *Effects*: TBD.

5 *Returns*: TBD.

```
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);
```

6 *expr-divide* divides two sorted expression templates.

7 *Effects*: TBD.

8 *Returns*: TBD.

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

9 *expr-invert* inverts the expression template `T`.

10 *Effects*: TBD.

11 *Returns*: TBD.

```
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);
```

12 *expr-pow* computes the power (IEC 60050, 102-02-08) $\texttt{T}^{\texttt{Num}/\texttt{Den}}$.

13 *Effects*: TBD.

14 *Returns*: TBD.

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

15 *expr-map* maps contents of one expression template to another resulting in a different type list.

16 Let `Proj` be projection used for mapping, and let `T` be the expression template to map from.

17 *Effects*: TBD.

18 *Returns*: TBD.

5.8 Dimension [qty.dim]

5.8.1 General [qty.dim.general]

1 Subclause 5.8 specifies the components for defining the dimension of a quantity (IEC 60050, 112-01-11).

5.8.2 Concepts [qty.dim.concepts]

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

template<typename T>
concept BaseDimension = Dimension<T> && std::derived_from<T, base_dimension>;

consteval bool is-dimension-one(std::meta::info type_alias) {
  return dealias(type_alias) == ^dimension_one;
}
```

```
template<typename T>
concept IsPowerOfDim =
  (is-specialization-of(^T, ^power) &&
   (BaseDimension<typename T::factor> || is-dimension-one(^typename T::factor)));

template<typename T>
constexpr bool is-per-of-dims = false;

template<typename... Ts>
constexpr bool is-per-of-dims<per<Ts...>> =
  (... && (BaseDimension<Ts> || is-dimension-one(^Ts) || IsPowerOfDim<Ts>));

template<typename T>
concept DerivedDimensionExpr =
  BaseDimension<T> || is-dimension-one(^T) || IsPowerOfDim<T> || is-per-of-dims<T>;

template<auto D1, auto D2>
concept SameDimension = Dimension<decltype(D1)> && Dimension<decltype(D2)> && D1 == D2;

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

5.8.3 Types [qty.dim.types]

```
namespace mp_units {

struct dimension-interface {
  friend consteval Dimension auto operator*(Dimension auto lhs, Dimension auto rhs)
  {
    return expr-multiply<derived_dimension, struct dimension_one>(lhs, rhs);
  }

  friend consteval Dimension auto operator/(Dimension auto lhs, Dimension auto rhs)
  {
    return expr-divide<derived_dimension, struct dimension_one>(lhs, rhs);
  }

  friend consteval bool operator==(Dimension auto lhs, Dimension auto rhs)
  {
    return ^decltype(lhs) == ^decltype(rhs);
  }
};

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

}
```

1 `base_dimension` is used by the program to define the dimension of a base quantity (IEC 60050, 112-01-08).

[*Example 1*:

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

— *end example*]

2 If `Symbol` is not unique in the base dimensions when simplifying a list of dimensions (5.7.3), the program is ill-formed, no diagnostic required.

```
namespace mp_units {

template<DerivedDimensionExpr... Expr>
struct derived-dimension-impl : expr-fractions<dimension_one, Expr...> {};

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

```
}
```

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

[*Example 2*:

```
using acceleration = decltype(si::dim_speed / si::dim_time);
static_assert(dealias(^acceleration) ==
              ^derived_dimension<si::dim_length, per<power<si::dim_time, 2>>>);
```

— *end example*]

4 *Mandates*: `Expr...` is simplified (5.7.3).

[*Note 1*: The library handles the representation of a derived dimension. The name is not *exposition only* for the portability and readability of the specializations. — *end note*]

```
namespace mp_units {

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

}
```

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

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

6 Computes $\texttt{d}^{\texttt{Num/Den}}$.

7 *Effects*: Equivalent to:

```
if constexpr (BaseDimension<D>) {
  if constexpr (Den == 1)
    return derived_dimension<power<D, Num>>{};
  else
    return derived_dimension<power<D, Num, Den>>{};
} else
  return expr-pow<Num, Den, derived_dimension, struct dimension_one>(d);
```

5.9 Quantity specification [qty.spec]

5.9.1 General [qty.spec.general]

1 Subclause 5.9 specifies the components for defining a quantity (IEC 60050, 112-01-01).

5.9.2 Concepts [qty.spec.concepts]

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

template<typename T>
concept QuantityKindSpec = is-specialization-of(^T, ^kind_of_);

template<typename T>
concept NamedQuantitySpec =
  QuantitySpec<T> && std::derived_from<T, quantity_spec> && !QuantityKindSpec<T>;

consteval bool is-dimensionless(std::meta::info type_alias) {
  return dealias(type_alias) == ^dimensionless;
}

template<typename T>
concept IsPowerOfQuantitySpec =
  (is-specialization-of(^T, ^power) &&
   (NamedQuantitySpec<typename T::factor> || is-dimensionless(^typename T::factor)));
```

```
template<typename T>
constexpr bool is-per-of-quantity-specs = false;

template<typename... Ts>
constexpr bool is-per-of-quantity-specs<per<Ts...>> =
  (... && (NamedQuantitySpec<Ts> || is-dimensionless(^Ts) || IsPowerOfQuantitySpec<Ts>));

template<typename T>
concept DerivedQuantitySpecExpr = NamedQuantitySpec<T> || is-dimensionless(^T) ||
                                  IsPowerOfQuantitySpec<T> || is-per-of-quantity-specs<T>;

template<typename T>
concept DerivedQuantitySpec =
  QuantitySpec<T> &&
  (is-specialization-of(^T, ^derived_quantity_spec) ||
   (QuantityKindSpec<T> && is-specialization-of(type_remove_const(type_of(^T::quantity-spec)),
                                                ^derived_quantity_spec)));

template<auto QS1, auto QS2>
concept SameQuantitySpec =
  QuantitySpec<decltype(QS1)> && QuantitySpec<decltype(QS2)> && (QS1 == QS2);

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

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

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

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

template<auto From, auto To>
concept QuantitySpecCastableTo =
  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{}>);
```

5.9.3 Types [qty.spec.types]

```
namespace mp_units {

template<QuantitySpec QS, Unit U>
  requires(!AssociatedUnit<U>) || UnitOf<U, QS{}>
consteval Reference auto make-reference(QS, U u)
{
  if constexpr (QuantityKindSpec<QS>)
    return u;
  else
    return reference<QS, U>{};
}
```

```
consteval quantity_character common-quantity-character(
  std::initializer_list<quantity_character> args)
{
  return max(args);
}

template<typename... Qs1, typename... Qs2>
consteval quantity_character derived-quantity-character(type_list<Qs1...>, type_list<Qs2...>)
{
  constexpr quantity_character num =
    common-quantity-character({quantity_character::scalar, [:expr-type(^Qs1):]::character...});
  constexpr quantity_character den =
    common-quantity-character({quantity_character::scalar, [:expr-type(^Qs2):]::character...});
  if constexpr (num == den)
    return quantity_character::scalar;
  else
    return common-quantity-character({num, den});
}

consteval quantity_character quantity-character-init(quantity_character ch,
                                                     std::initializer_list<std::meta::info> args)
{
  auto it =
    std::ranges::find_if(args, [](auto arg) { return type_of(arg) == ^quantity_character; });
  if (it != args.end()) return extract<quantity_character>(*it);
  return ch;
}

template<NamedQuantitySpec Q>
  requires requires { Q::dimension; }
using to-dimension = [:type_remove_const(type_of(^Q::dimension)):];

struct quantity-spec-interface-base {
  friend consteval QuantitySpec auto operator*(QuantitySpec auto lhs, QuantitySpec auto rhs)
  {
    return clone-kind-of<lhs, rhs>(expr-multiply<derived_quantity_spec, struct dimensionless>(
      remove-kind(lhs), remove-kind(rhs)));
  }

  friend consteval QuantitySpec auto operator/(QuantitySpec auto lhs, QuantitySpec auto rhs)
  {
    return clone-kind-of<lhs, rhs>(expr-divide<derived_quantity_spec, struct dimensionless>(
      remove-kind(lhs), remove-kind(rhs)));
  }

  friend consteval bool operator==(QuantitySpec auto lhs, QuantitySpec auto rhs)
  {
    return ^decltype(lhs) == ^decltype(rhs);
  }
};

struct quantity-spec-interface : quantity-spec-interface-base {
  template<typename Self, UnitOf<Self{}> U>
  consteval Reference auto operator[](this Self self, U u)
  {
    return make-reference(self, u);
  }

  template<typename Self, typename FwdQ, Quantity Q = [:type_remove_cvref(^FwdQ):]>
    requires QuantitySpecExplicitlyConvertibleTo<Q::quantity_spec, Self{}>
  constexpr Quantity auto operator()(this Self self, FwdQ&& q)
  {
    return quantity{std::forward<FwdQ>(q).numerical-value, make-reference(self, Q::unit)};
```

```
  }
};

inline constexpr struct is_kind {
} is_kind;

template<auto Arg>
concept quantity-spec-1-arg =
  (type_of(^Arg) == ^quantity_character) &&
  std::ranges::contains(enumerators_of(^quantity_character), value_of(^Arg));

template<auto Lhs, auto Rhs>
concept quantity-spec-2-args = (quantity-spec-1-arg<Lhs> && value_of(^Rhs) == is_kind) ||
                               (quantity-spec-1-arg<Rhs> && value_of(^Lhs) == is_kind);

template<int MaxArgs, auto... Args>
concept quantity-spec-args =
  (... && !QuantitySpec<decltype(Args)>) &&
  ((sizeof...(Args) == 0) ||  //
   (sizeof...(Args) == 1 && quantity-spec-1-arg<Args...>) ||
   (sizeof...(Args) == 2 && quantity-spec-2-args<Args...> && MaxArgs == 2));

template<BaseDimension auto Dim, auto... Args>
  requires quantity-spec-args<1, 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 =
    quantity-character-init(quantity_character::scalar, {^Args...});
};

template<DerivedQuantitySpec auto Eq, auto... Args>
  requires quantity-spec-args<1, 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 =
    quantity-character-init(Eq.character, {^Args...});
};

template<NamedQuantitySpec auto QS, auto... Args>
  requires quantity-spec-args<2, Args...>
struct quantity_spec<QS, Args...> : quantity-spec-interface {
  using base-type = quantity_spec;
  static constexpr auto parent = QS;  // clang-format off
  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;  // clang-format on
  static constexpr quantity_character character =
    quantity-character-init(QS.character, {^Args...});
};

template<NamedQuantitySpec auto QS, DerivedQuantitySpec auto Eq, auto... Args>
  requires quantity-spec-args<2, Args...> && 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 =
    quantity-character-init(Eq.character, {^Args...});
};
```

```
}
```

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 used an alluded signature. The identifier of `Q` represents its quantity name (IEC 60050, 112-01-02).

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

(3.1) — (A base dimension, $\texttt{set}_{opt}$),

(3.2) — (A quantity calculus, $\texttt{set}_{opt}$),

(3.3) — (A named quantity, $\texttt{set}_{opt}$, $\texttt{is\_kind}_{opt}$), and

(3.4) — (A named quantity, a quantity calculus, $\texttt{set}_{opt}$, $\texttt{is\_kind}_{opt}$).

4 If the first argument is a base dimension, then `Q` is its base quantity (IEC 60050, 112-01-08). If an argument is a quantity calculus (IEC 60050, 112-01-30) C, then $\texttt{Q} = C$ (a quantity equation (IEC 60050, 112-01-31)). If the first argument is a named quantity, then `Q` is of its kind (IEC 60050, 112-01-04).

5 `set` is the set of the numerical value of `Q`. If not specified, then it defaults to `scalar` for the first signature, and that of the last quantity argument otherwise. `is_kind` specifies `Q` to start a new hierarchy tree of a kind.

6 [*Example 1*:

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

// The second signature establishes an equation.
inline constexpr struct area final : quantity_spec<pow<2>(length)> {
} area;  // A = l².

// The third signature establishes a kind of quantity.
inline constexpr struct width final : quantity_spec<length> {
} width;

// 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 s/r, not just any quantity of dimension one.
```

— *end example*]

```
namespace mp_units {

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

  static constexpr Dimension auto dimension =
    expr-map<to-dimension, derived_dimension, struct dimension_one>(base{});
  static constexpr quantity_character character =
    derived-quantity-character(typename base::num{}, typename base::den{});
};

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

}
```

7 A specialization of `derived_quantity_spec` is returned from a quantity calculus not equal to a named quantity.

[*Example 2*:

```
auto area = pow<2>(length);
```

```
static_assert(decltype(area) == ^derived_quantity_spec<power<length, 2>>);
```

— *end example*]

```
namespace mp_units {

inline constexpr struct dimensionless final : derived_quantity_spec<> {
} dimensionless;

}
```

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

```
namespace mp_units {

template<typename T>
concept QuantitySpecWithNoSpecifiers = NamedQuantitySpec<T> || DerivedQuantitySpec<T>;

template<typename Q>
  requires QuantitySpecWithNoSpecifiers<Q> && SameQuantitySpec<get-kind-tree-root(Q{}), Q{}>
struct kind_of_<Q> final : Q::base-type {
  using base-type = kind_of_;
  static constexpr auto quantity-spec = Q{};
};

}
```

9 `kind_of<Q>` specifies `Q` to be treated as a quantity kind.

```
namespace mp_units {

template<QuantitySpec auto... From, QuantitySpec Q>
consteval QuantitySpec auto clone-kind-of(Q q)
{
  if constexpr ((... && QuantityKindSpec<decltype(From)>))
    return kind_of<Q{}>;
  else
    return q;
}

template<QuantitySpec Q>
consteval auto remove-kind(Q q)
{
  if constexpr (QuantityKindSpec<Q>)
    return Q::quantity-spec;
  else
    return q;
}

}
```

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

10 Computes $q^{Num/Den}$.

11 *Effects*: Equivalent to:

```
if constexpr (Num == 0 || q == dimensionless)
  return dimensionless;
else if constexpr (ratio{Num, Den} == 1)
  return q;
else if constexpr (DerivedQuantitySpec<Q>)
  return clone-kind-of<q>(
    expr-pow<Num, Den, derived_quantity_spec, struct dimensionless>(remove-kind(q)));
else if constexpr (Den == 1)
  return clone-kind-of<q>(derived_quantity_spec<power<decltype(remove-kind(q)), Num>>{});
```

```
      else
        return clone-kind-of<q>(
          derived_quantity_spec<power<decltype(remove-kind(q)), Num, Den>>{});
```

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

12 *Returns*: TBD.

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

13 *Returns*: TBD.

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

14 *Returns*: TBD.

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

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

```
namespace mp_units {

template<QuantitySpec Q>
  requires requires(Q q) { get-kind-tree-root(q); }
using to-kind = decltype(get-kind-tree-root(Q{}));

template<QuantitySpec Q>
consteval QuantitySpec auto get-kind-tree-root(Q q)
{
  auto defined_as_kind = []<auto... Args>(quantity_spec<Args...>) {
    return (... || type_of(^Args) == ^struct is_kind);
  };

  if constexpr (QuantityKindSpec<Q>) {
    return remove-kind(q);
  } else if constexpr (defined_as_kind(q)) {
    return q;
  } else if constexpr (requires { Q::parent; }) {
    return get-kind-tree-root(Q::parent);
  } else if constexpr (DerivedQuantitySpec<Q>) {
    return expr-map<to-kind, derived_quantity_spec, struct dimensionless>(q);
  } else {
    return q;
  }
}

}
```

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

16 *Returns*: `kind_of<`*`get-kind-tree-root`*`(q)>`.

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

17 *Returns*: `q`.

```
template<QuantitySpec Q1, QuantitySpec Q2>
  requires QuantitySpecConvertibleTo<get-kind-tree-root(Q1{}), get-kind-tree-root(Q2{})> ||
           QuantitySpecConvertibleTo<get-kind-tree-root(Q2{}), get-kind-tree-root(Q1{})>
consteval QuantitySpec auto get_common_quantity_spec(Q1 q1, Q2 q2);
```

18 *Effects*: Equivalent to:

```
  using QQ1 = decltype(remove_kind(q1));
  using QQ2 = decltype(remove_kind(q2));

  if constexpr (^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);

consteval QuantitySpec auto get_common_quantity_spec(QuantitySpec auto q1, QuantitySpec auto q2,
                                                     QuantitySpec auto q3,
                                                     QuantitySpec auto... rest)
  requires requires { get_common_quantity_spec(get_common_quantity_spec(q1, q2), q3, rest...); };
```

19 *Returns*: `get_common_quantity_spec(get_common_quantity_spec(q1, q2), q3, rest...)`.

20 Let `Q` be a specialization of `quantity_spec` with a *`parent`* member.

```
namespace mp_units {

consteval std::vector<std::meta::info> quantity-spec-hierarchy(std::meta::info q)
{
  auto get_parent = [](std::meta::info q) -> std::optional<std::meta::info> {
    auto mems = members_of(q);
    auto it = std::ranges::find_if(mems, [parent_id = identifier_of(^Q::parent)](auto m) {
      return has_identifier(m) && identifier_of(m) == parent_id;
    });
    if (it != args.end()) return *it;
    return std::nullopt;
  };
  std::vector<std::meta::info> res{q};
  std::optional<std::meta::info> parent;
  while (parent = get_parent(res.back())) {
    res.push_back(*parent);
  }
  return res;
}

}

template<QuantitySpec A, QuantitySpec B>
consteval std::optional<std::meta::info> get-common-base(A a, B b);
```

21 *Effects*: Equivalent to:

```
  auto hier_a = quantity-spec-hierarchy(^a);
  auto hier_b = quantity-spec-hierarchy(^b);
  auto max_valid_depth = std::min(hier_a.size(), hier_b.size());
  auto res =
    std::ranges::mismatch(std::span{hier_a}.last(max_valid_depth),
                          std::span{hier_b}.last(max_valid_depth), std::not_equal_to);
```

```
    if (to_address(res.in1) != to_address(hier_a.end())) {
      return *res.in1;
    }
    return std::nullopt;

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

22 *Effects*: Equivalent to:

```
    if constexpr (Child{} == Parent{})
      return true;
    else {
      auto hier_ch = quantity-spec-hierarchy(^ch);
      auto hier_p = quantity-spec-hierarchy(^p);
      if (hier_p.size() > hier_ch.size())
        return false;
      else
        return std::span{hier_ch}.last(hier_p.size())[0] == ^p;
    }
```

5.10 Magnitude [qty.mag]

5.11 Unit [qty.unit]

```
template<typename T>
concept Unit = tag-type<T> && std::derived_from<T, unit-interface>;
template<typename T>
concept UnitOf = unspecified;
template<typename T>
concept AssociatedUnit = unspecified;
```

5.12 Reference [qty.ref]

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

5.13 Quantity types [qty.types]

5.13.1 Traits [qty.traits]

5.13.1.1 Values [qty.val.traits]

```
namespace mp_units {

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

}
```

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

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

2 *Effects*: Equivalent to:

```
    if constexpr (requires {
                    { std::chrono::duration_values<Rep>::one() } -> std::same_as<Rep>;
                  })
      return std::chrono::duration_values<Rep>::one();
    else
      return Rep(1);
```

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

5.13.1.2 Compatibility [qty.compat.traits]

1 The interfaces specified in this subclause are used by the quantity types (5.13.2) to specify conversions with other types representing quantities. 5.15 implements them for `std::chrono::duration` and `std::chrono::time_point`.

```
template<typename T, template<typename> typename Traits>
concept qty-like = requires(const T& qty, const Traits<T>::rep& num) {
  requires !is-specialization-of(^T, ^quantity);
  requires !is-specialization-of(^T, ^quantity_point);
  { Traits<T>::to_numerical_value(qty) } -> std::same_as<typename Traits<T>::rep>;
  { Traits<T>::from_numerical_value(num) } -> std::same_as<T>;
  { Traits<T>::explicit_import } -> std::same_as<const bool>;
  { Traits<T>::explicit_export } -> std::same_as<const bool>;
  typename std::bool_constant<Traits<T>::explicit_import>;
  typename std::bool_constant<Traits<T>::explicit_export>;
};

template<typename T>
concept quantity_like =                  //
  qty-like<T, quantity_like_traits> &&   //
  requires {
    typename quantity<quantity_like_traits<T>::reference, typename quantity_like_traits<T>::rep>;
  };

template<typename T>
concept quantity_point_like =
  qty-like<T, quantity_point_like_traits> &&  //
  requires {
    typename quantity_point<quantity_point_like_traits<T>::reference,
                            typename quantity_point_like_traits<T>::point_origin,
                            typename quantity_point_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 Tratis<Q>::rep`.

3 `Q` models *qty-like*`<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 a quantity type.

(4.2) — `Traits<Q>::explicit_export` for the conversion from a quantity type to `Q`.

5.13.2 General [qty.types.general]

1 A *quantity type* is a type *Q* that is a specialization of `quantity` or `quantity_point`. *Q* represents the value of a quantity (IEC 60050, 112-01-28) with *Q*`::rep` as its number and *Q*`::reference` as its reference. *Q* is a structural type (N4971, [temp.param]) if *Q*`::rep` is a structural type.

2 Each class template defined in subclauses 5.13.4 and 5.13.6 have data members and special members specified below, and have no base classes or members other than those specified.

5.13.3 Quantity concepts [qty.concepts]

```
template<typename T>
concept Quantity = unspecified;
```

5.13.4 Class template quantity [qty]

```
namespace mp_units {

template<auto R1, auto R2, typename Rep1, typename Rep2>
concept same-value-as =
  equivalent(get_unit(R1), get_unit(R2)) && std::convertible_to<Rep1, Rep2>;

template<Reference auto R, RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  Rep numerical-value;

  // 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;

  // static member functions

  static constexpr quantity zero() noexcept
    requires requires { quantity_values<rep>::zero(); }
  {
    return {quantity_values<rep>::zero(), R};
  }

  static constexpr quantity one() noexcept
    requires requires { quantity_values<rep>::one(); }
  {
    return {quantity_values<rep>::one(), R};
  }

  static constexpr quantity min() noexcept
    requires requires { quantity_values<rep>::min(); }
  {
    return {quantity_values<rep>::min(), R};
  }

  static constexpr quantity max() noexcept
    requires requires { quantity_values<rep>::max(); }
  {
    return {quantity_values<rep>::max(), R};
  }

  // construction, assignment, destruction

  quantity() = default;
  quantity(const quantity&) = default;
  quantity(quantity&&) = default;
  ~quantity() = default;

  template<typename FwdValue, Reference R2>
    requires same-value-as<R2{}, R, [:type_remove_cvref(^FwdValue):], Rep>
  constexpr quantity(FwdValue&& v, R2) : numerical-value(std::forward<FwdValue>(v))
  {
  }
};

}
```

1 Let *Q* be a specialization of `quantity`.

(1.1) — If `Rep` is a scalar (5.6.1.2), *Q* represents a scalar quantity (IEC 60050, 102-02-19).

(1.2) — If `Rep` is a vector (5.6.1.2), *`Q`* represents a vector (IEC 60050, 102-03-04).

5.13.5 Quantity point concepts [qty.pt.concepts]

```
template<typename T>
concept point_origin = unspecified;
```

5.13.6 Class template `quantity_point` [qty.pt]

```
namespace mp_units {

template<unspecified>
class quantity_point { unspecified };

}
```

1 A *quantity point type* is a specialization of `quantity_point`. Let *`Q`* be a quantity point type. *`Q`*`::point_origin` represents the origin point of a position vector (IEC 60050, 102-03-15) or of a component (IEC 60050, 102-03-10) thereof.

(1.1) — If `Rep` is a scalar (5.6.1.2), *`Q`* represents the scalar quantity (IEC 60050, 102-02-19) of a position vector.

(1.2) — If `Rep` is a vector (5.6.1.2), *`Q`* represents a position vector.

5.14 Systems [qty.systems]

5.15 `std::chrono` compatibility [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>();
  using rep = Rep;

  static constexpr bool explicit_import = false;
  static constexpr rep to_numerical_value(const std::chrono::duration<Rep, Period>& q) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return q.count();
  }
```

```
  static constexpr bool explicit_export = false;
  static constexpr std::chrono::duration<Rep, Period> from_numerical_value(
    const rep& v) noexcept(std::is_nothrow_copy_constructible_v<rep>)
  {
    return std::chrono::duration<Rep, Period>(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>>> {
  using Tp = 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>;
  using rep = Rep;

  static constexpr bool explicit_import = false;
  static constexpr rep to_numerical_value(const Tp& tp) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return tp.time_since_epoch().count();
  }

  static constexpr bool explicit_export = false;
  static constexpr Tp from_numerical_value(const rep& v) noexcept(
    std::is_nothrow_copy_constructible_v<rep>)
  {
    return Tp(std::chrono::duration<Rep, Period>(v));
  }
};

}
```

Cross-references

Each clause and subclause label is listed below along with the corresponding clause or subclause number and page number, in alphabetical order by label.

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.

U

V

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.