C++ONLINE

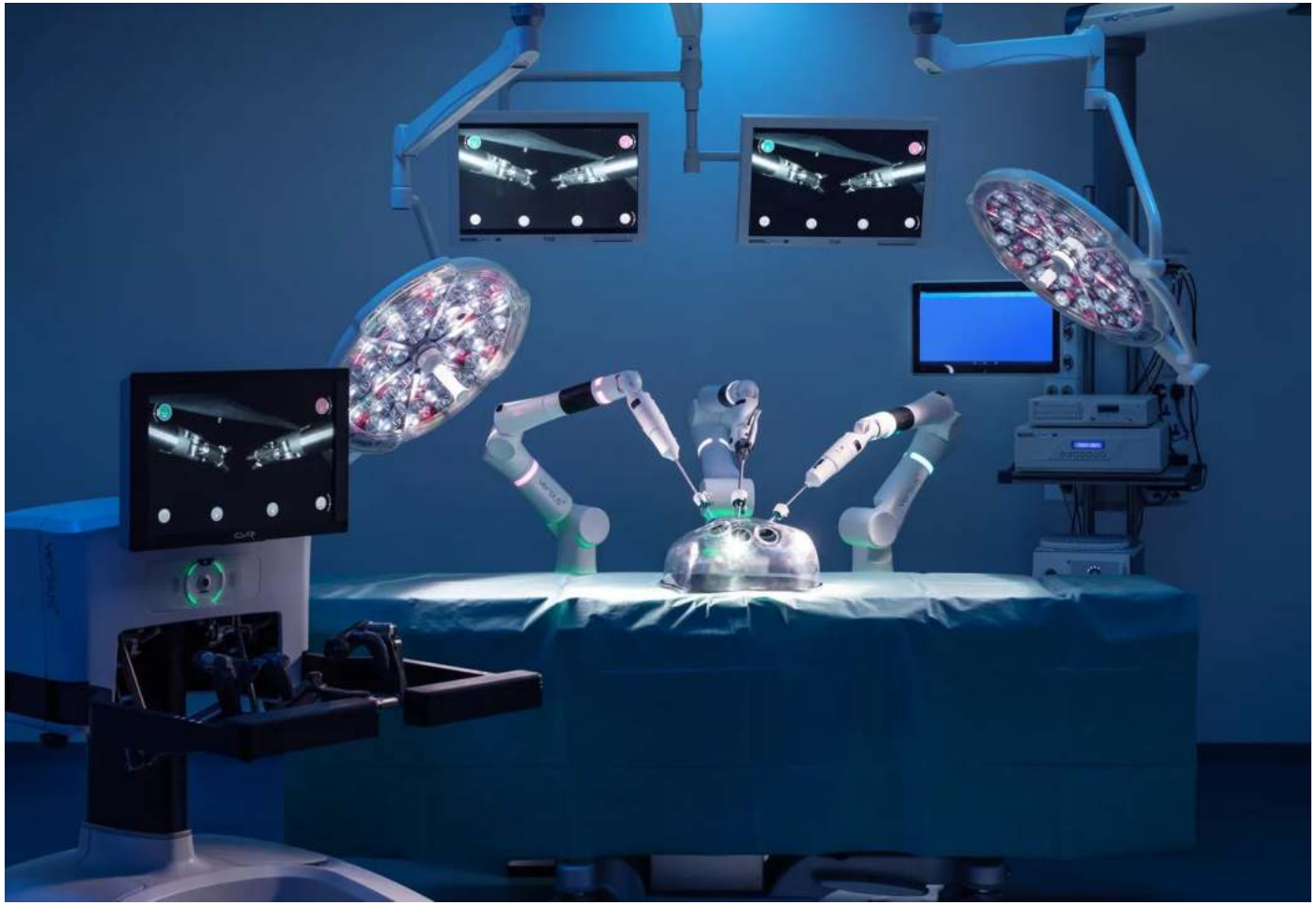
MATEUSZ PUSZ

IMPROVING SAFETY WITH QUANTITIES AND UNITS LIBRARY

2024

The future is here

The future is here



C++ safety

- A major concern in the C++ Community in the recent years
- Potential improvements are being discussed
  - handling of low-level fundamental types
  - updating the core language rules
  - providing safer high-level abstractions in the library

C++ developers needs help

- Many C++ engineers are expected to write life-critical software today
- Experience in this domain is hard to come by
- Training alone will not solve the issue of mistakes caused by confusing units or quantities
- Code handling the physical computation is often written by domain experts such as physicists that are not necessarily fluent in C++

Affected industries

- Aerospace
- Autonomous cars
- Embedded industries

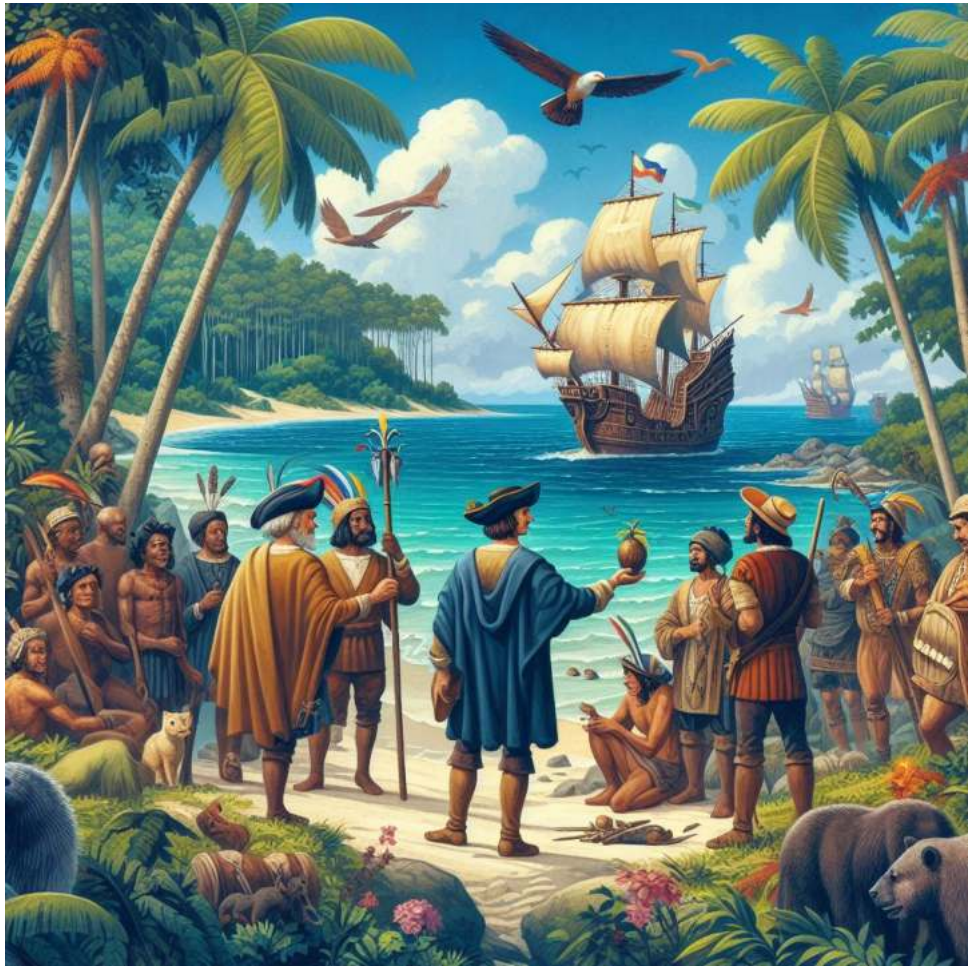
Affected industries

- Aerospace
- Autonomous cars
- Embedded industries
- Manufacturing
- Maritime industry
- Freight transport
- Military
- Astronomy
- 3D design
- Robotics
- Audio
- Medical devices
- National laboratories
- Scientific institutions and universities
- All kinds of navigation and charting
- GUI frameworks
- Finance (including HFT)
- ...

MISMEASURE FOR MEASURE

Mismeasure for measure

- Christopher Columbus



Mismeasure for measure

- Christopher Columbus
- Vasa

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider

Mismeasure for measure

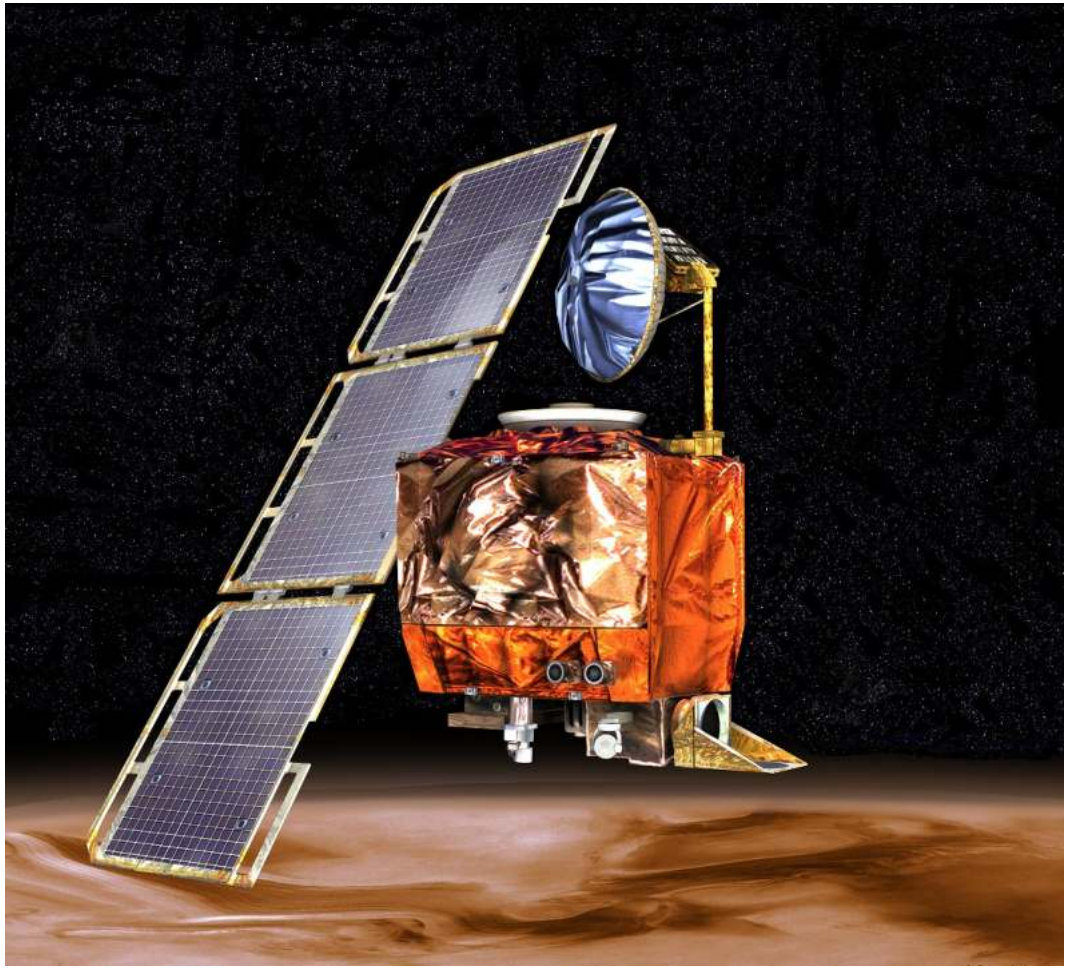
- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter



Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise
- Tokyo Disneyland’s Space Mountain

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise
- Tokyo Disneyland’s Space Mountain
- Hochrheinbrücke bridge

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise
- Tokyo Disneyland’s Space Mountain
- Hochrheinbrücke bridge
- Wild rice

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise
- Tokyo Disneyland’s Space Mountain
- Hochrheinbrücke bridge
- Wild rice
- The Guardian

Mismeasure for measure

- Christopher Columbus
- Vasa
- Gimli Glider
- Black Sabbath
- Korean Air Cargo Flight 6316
- Mars Climate Orbiter
- Clarence the Tortoise
- Tokyo Disneyland’s Space Mountain
- Hochrheinbrücke bridge
- Wild rice
- The Guardian
- Medication dose errors

TYPICAL PRODUCTION ISSUES

The proliferation of **double**

```
double GlidePolar::MacCreadyAltitude(double MCREADY,
                                     double Distance,
                                     const double Bearing,
                                     const double WindSpeed,
                                     const double WindBearing,
                                     double *BestCruiseTrack,
                                     double *VMacCready,
                                     const bool isFinalGlide,
                                     double *TimeToGo,
                                     const double AltitudeAboveTarget=1.0e6,
                                     const double cruise_efficiency=1.0,
                                     const double TaskAltDiff=-1.0e6);
```

The proliferation of magic numbers

```
double AirDensity(double hr, double temp, double abs_press)
{
  return (1/(287.06*(temp+273.15)))*
         (abs_press - 230.617 * hr * exp((17.5043*temp)/(241.2+temp)));
}
```

The proliferation of conversion macros

```
#ifndef PI
static const double PI = (4*atan(1));
#endif
#define EARTH_DIAMETER    12733426.0    // Diameter of earth in meters
#define SQUARED_EARTH_DIAMETER  162140137697476.0 // Diameter of earth in meters (EARTH_DIAMETER*EARTH_DIAMETER)
#ifndef DEG_TO_RAD
#define DEG_TO_RAD  (PI / 180)
#define RAD_TO_DEG  (180 / PI)
#endif

#define NAUTICALMILESTOMETRES (double)1851.96
#define KNOTSTOMETRESSECONDS (double)0.5144

#define TOKNOTS (double)1.944
#define TOFEETPERMINUTE (double)196.9
#define TOMPH   (double)2.237
#define TOKPH   (double)3.6

// meters to.. conversion
#define TONAUTICALMILES (1.0 / 1852.0)
#define TOMILES         (1.0 / 1609.344)
#define TOKILOMETER     (0.001)
#define TOFEET          (1.0 / 0.3048)
#define TOMETER         (1.0)
```

The proliferation of conversion macros

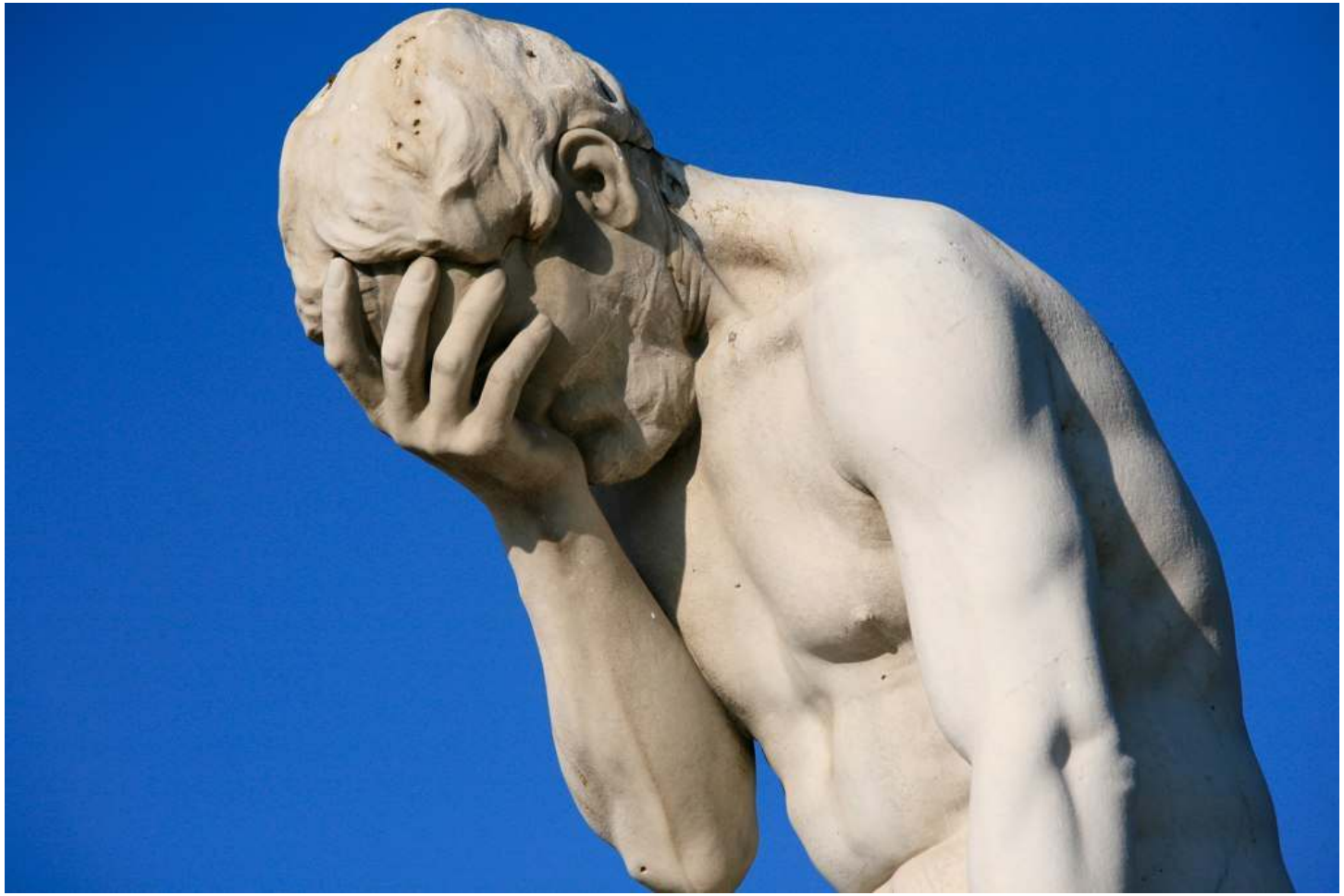
- Often more than once in the repository

```
#define RAD_TO_DEG (180 / PI)
#define RAD_TO_DEG 57.2957795131
#define RAD_TO_DEG ( radians ) ((radians ) * 180.0 / M_PI)
#define RAD_TO_DEG 57.2957805f
```

Lack of consistency

```
void DistanceBearing(double lat1, double lon1,
                     double lat2, double lon2,
                     double *Distance, double *Bearing);

double DoubleDistance(double lat1, double lon1,
                      double lat2, double lon2,
                      double lat3, double lon3);

void FindLatitudeLongitude(double Lat, double Lon,
                           double Bearing, double Distance,
                           double *lat_out, double *lon_out);

double CrossTrackError(double lon1, double lat1,
                       double lon2, double lat2,
                       double lon3, double lat3,
                       double *lon4, double *lat4);

double ProjectedDistance(double lon1, double lat1,
                         double lon2, double lat2,
                         double lon3, double lat3,
                         double *xtd, double *crs);
```

Typical production issues



MP-UNITS & STANDARDIZATION

mp-units: C++20/23 quantities and units library

- **Compile-time safety**
  - correct handling of physical quantities, units, and numerical values

mp-units: C++20/23 quantities and units library

- **Compile-time safety**
  - correct handling of physical quantities, units, and numerical values
- **Performance**
  - as fast or even faster than working with fundamental types
  - no runtime overhead
  - no space size overhead

mp-units: C++20/23 quantities and units library

- **Compile-time safety**
  - correct handling of physical quantities, units, and numerical values
- **Performance**
  - as fast or even faster than working with fundamental types
  - no runtime overhead
  - no space size overhead
- **Great user experience**
  - optimized for readable compilation errors and great debugging experience
  - easy to use and flexible

mp-units: C++20/23 quantities and units library

- **Compile-time safety**
  - correct handling of physical quantities, units, and numerical values
- **Performance**
  - as fast or even faster than working with fundamental types
  - no runtime overhead
  - no space size overhead
- **Great user experience**
  - optimized for readable compilation errors and great debugging experience
  - easy to use and flexible
- **Scope**
  - any unit's magnitude (huge, small, floating-point)
  - systems of quantities
  - systems of units
  - the affine space
  - highly adjustable text-output formatting
  - scalar, vector, and tensor quantities
  - natural units systems

mp-units: C++20/23 quantities and units library

- **Compile-time safety**
  - correct handling of physical quantities, units, and numerical values
- **Performance**
  - as fast or even faster than working with fundamental types
  - no runtime overhead
  - no space size overhead
- **Great user experience**
  - optimized for readable compilation errors and great debugging experience
  - easy to use and flexible
- **Scope**
  - any unit's magnitude (huge, small, floating-point)
  - systems of quantities
  - systems of units
  - the affine space
  - highly adjustable text-output formatting
  - scalar, vector, and tensor quantities
  - natural units systems
- **Easy to extend**

mp-units: C++20/23 quantities and units library

C++ FEATURE	C++ VERSION	GCC	CLANG	APPLE-CLANG	MSVC
Minimum support	20	12	16	15	None
`std::format`	20	13	17	None	None
C++ modules	20	14	17	None	None
Static `constexpr` variables in `constexpr` functions	23	13	17	None	None
Explicit `this` parameter	23	14	18	None	None

mp-units: C++20/23 quantities and units library

C++ FEATURE	C++ VERSION	GCC	CLANG	APPLE-CLANG	MSVC
Minimum support	20	12	16	15	None
`std::format`	20	13	17	None	None
C++ modules	20	14	17	None	None
Static `constexpr` variables in `constexpr` functions	23	13	17	None	None
Explicit `this` parameter	23	14	18	None	None

- Available on
  - GitHub
  - Conan
  - Compiler Explorer

ISO C++ papers

- P1935: A C++ Approach to Physical Units

ISO C++ papers

- P1935: A C++ Approach to Physical Units
- P2980: A motivation, scope, and plan for a physical quantities and units library

ISO C++ papers

- P1935: A C++ Approach to Physical Units
- P2980: A motivation, scope, and plan for a physical quantities and units library
- P3045: Quantities and units library

ISO C++ papers

- P1935: A C++ Approach to Physical Units
- P2980: A motivation, scope, and plan for a physical quantities and units library
- P3045: Quantities and units library

ASSOCIATED PROPOSALS

- P3003: The design of a library of number concepts
- P3094: `std::basic_fixed_string`
- P3133: Compile-time prime numbers

ISO C++ papers

- P1935: A C++ Approach to Physical Units
- P2980: A motivation, scope, and plan for a physical quantities and units library
- P3045: Quantities and units library

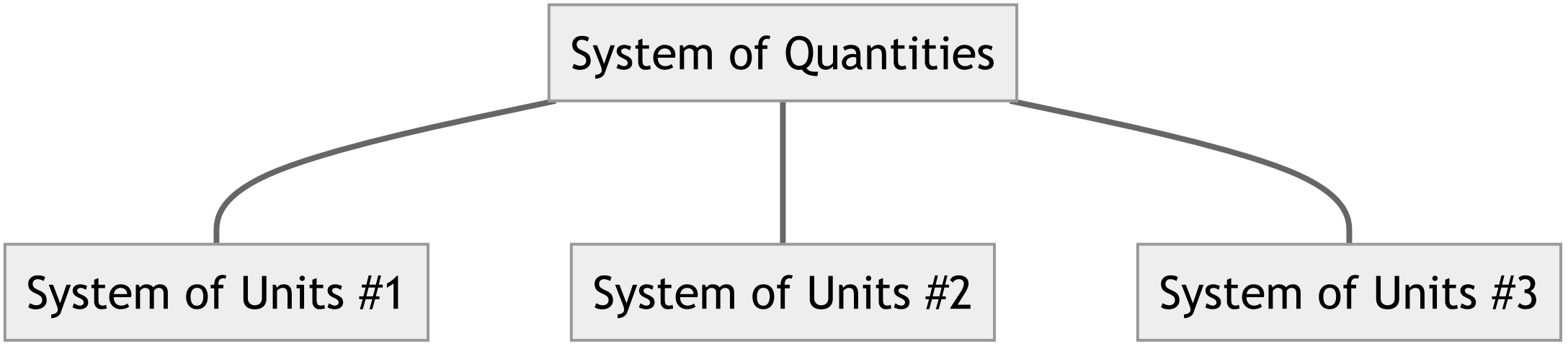
ASSOCIATED PROPOSALS

- P3003: The design of a library of number concepts
- P3094: `std::basic_fixed_string`
- P3133: Compile-time prime numbers

There is a high interest in standardizing a quantities and units library.

QUICK DOMAIN INTRODUCTION

Quick domain introduction



Quick domain introduction

SYSTEM OF QUANTITIES

- A set of quantities together with a set of noncontradictory equations relating those quantities

Quick domain introduction

SYSTEM OF QUANTITIES

- A set of quantities together with a set of noncontradictory equations relating those quantities

INTERNATIONAL SYSTEM OF QUANTITIES (ISQ)

- Provided by ISO
- System of quantities based on the seven base quantities

Quick domain introduction

SYSTEM OF QUANTITIES

- A set of quantities together with a set of noncontradictory equations relating those quantities

SYSTEM OF UNITS

- A set of base units and derived units, together with their multiples and submultiples, defined in accordance with given rules, for a given system of quantities

INTERNATIONAL SYSTEM OF QUANTITIES (ISQ)

- Provided by ISO
- System of quantities based on the seven base quantities

Quick domain introduction

SYSTEM OF QUANTITIES

- A set of quantities together with a set of noncontradictory equations relating those quantities

SYSTEM OF UNITS

- A set of base units and derived units, together with their multiples and submultiples, defined in accordance with given rules, for a given system of quantities

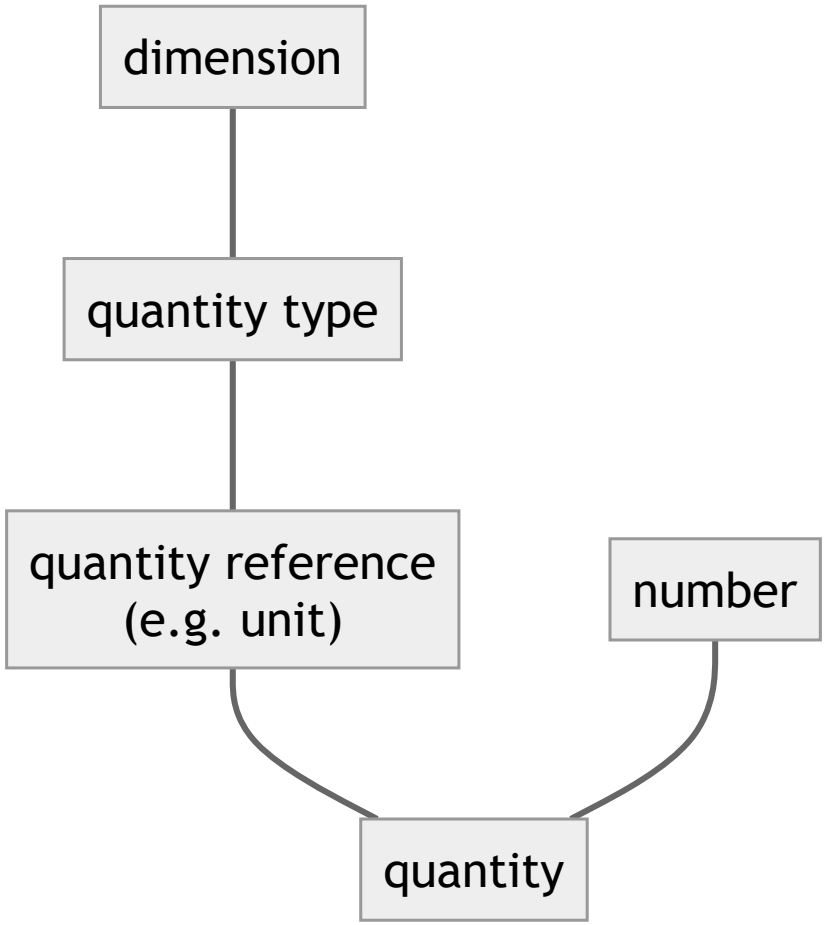
INTERNATIONAL SYSTEM OF QUANTITIES (ISQ)

- Provided by ISO
- System of quantities based on the seven base quantities

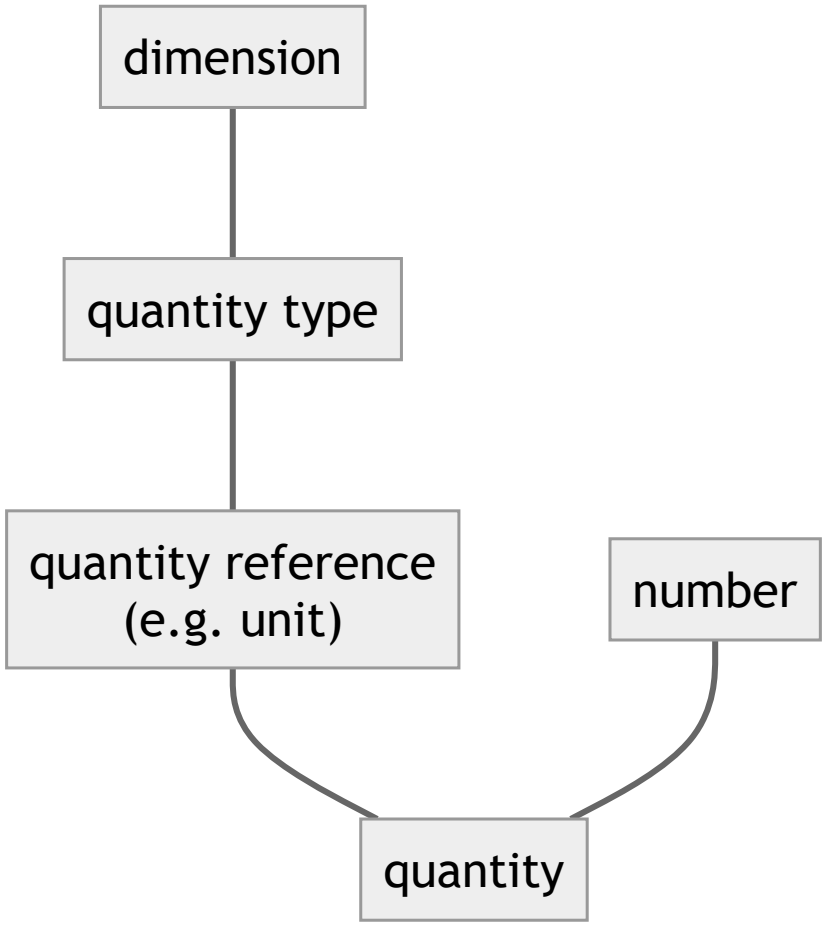
INTERNATIONAL SYSTEM OF UNITS (SI)

- Provided by the General Conference on Weights and Measures (CGPM)
- Based on the ISQ

Quick domain introduction



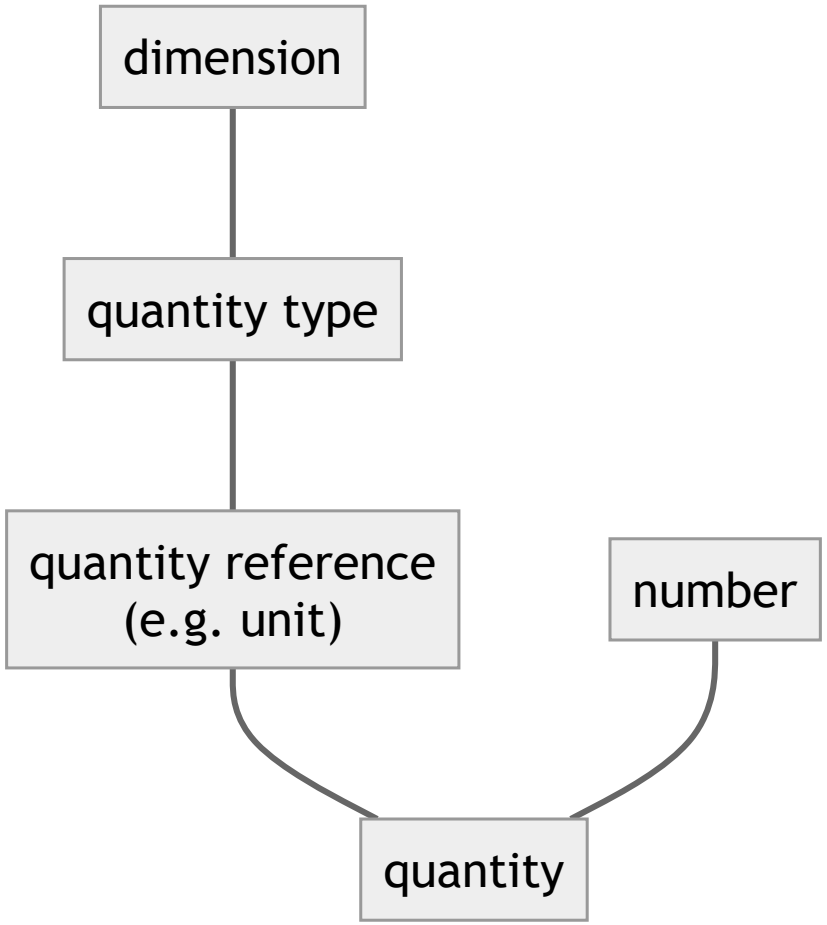
Quick domain introduction



DIMENSIONS

- Dimensions of base quantities in the ISQ
  - *length* (L),
  - *mass* (M),
  - *time* (T),
  - *electric current* (I),
  - *thermodynamic temperature* (Θ),
  - *amount of substance* (N),
  - *luminous intensity* (J)

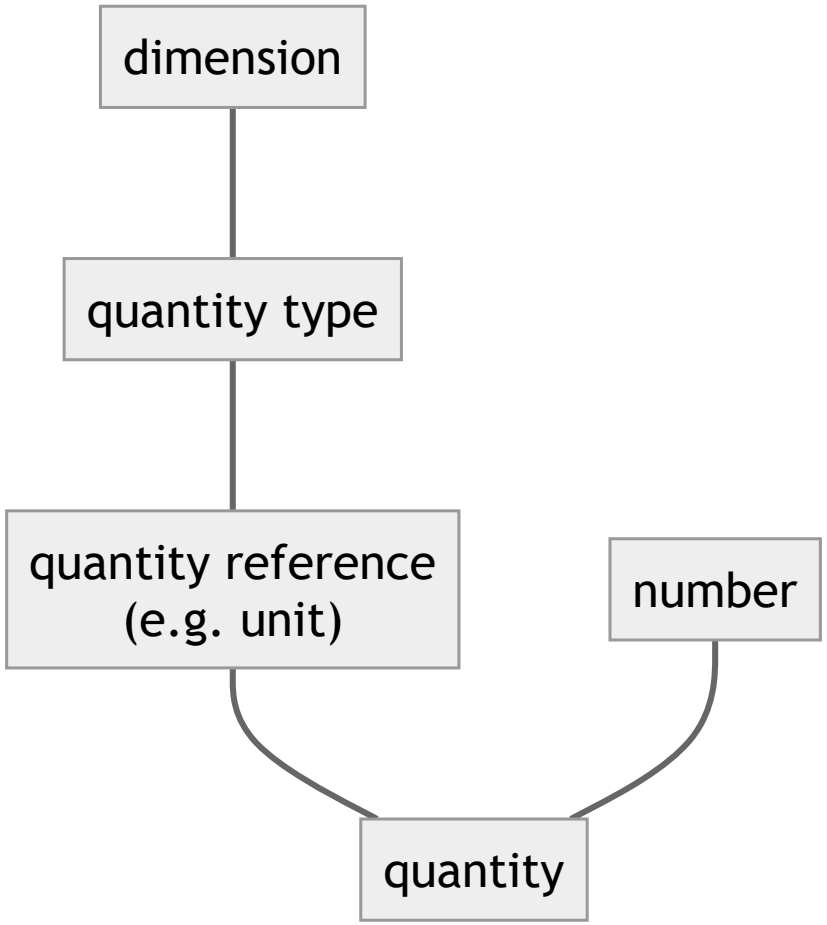
Quick domain introduction



DIMENSIONS

- Dimensions of base quantities in the ISQ
  - *length* (L),
  - *mass* (M),
  - *time* (T),
  - *electric current* (I),
  - *thermodynamic temperature* (Θ),
  - *amount of substance* (N),
  - *luminous intensity* (J)
- Dimensions of derived quantities
  - *force* is denoted by $dim\ F = LMT^{-2}$

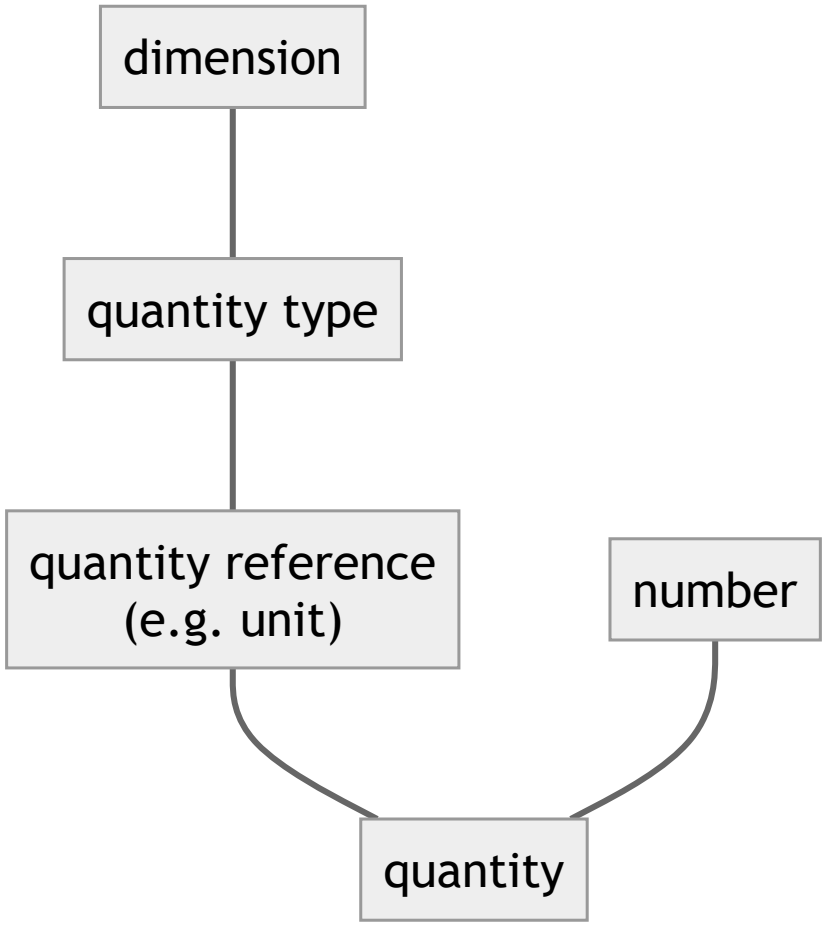
Quick domain introduction



QUANTITY TYPE

- Dimension is not enough to describe a quantity

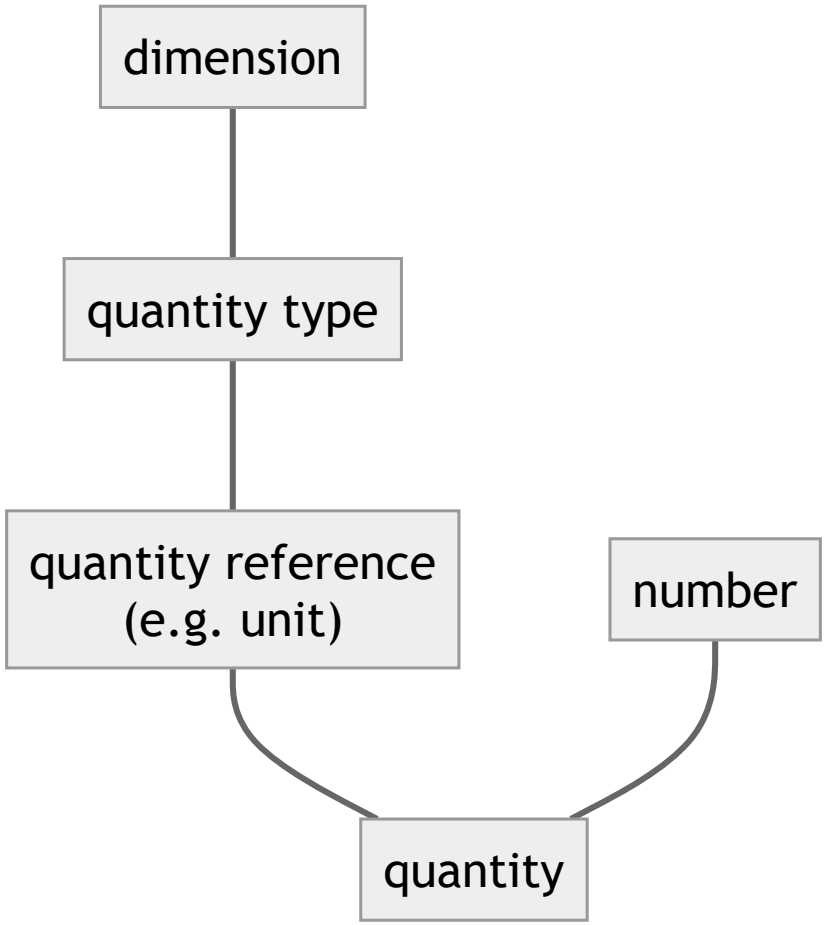
Quick domain introduction



QUANTITY TYPE

- Dimension is not enough to describe a quantity
- Different dimensions
  - *length*, *time*, *speed*, *power*

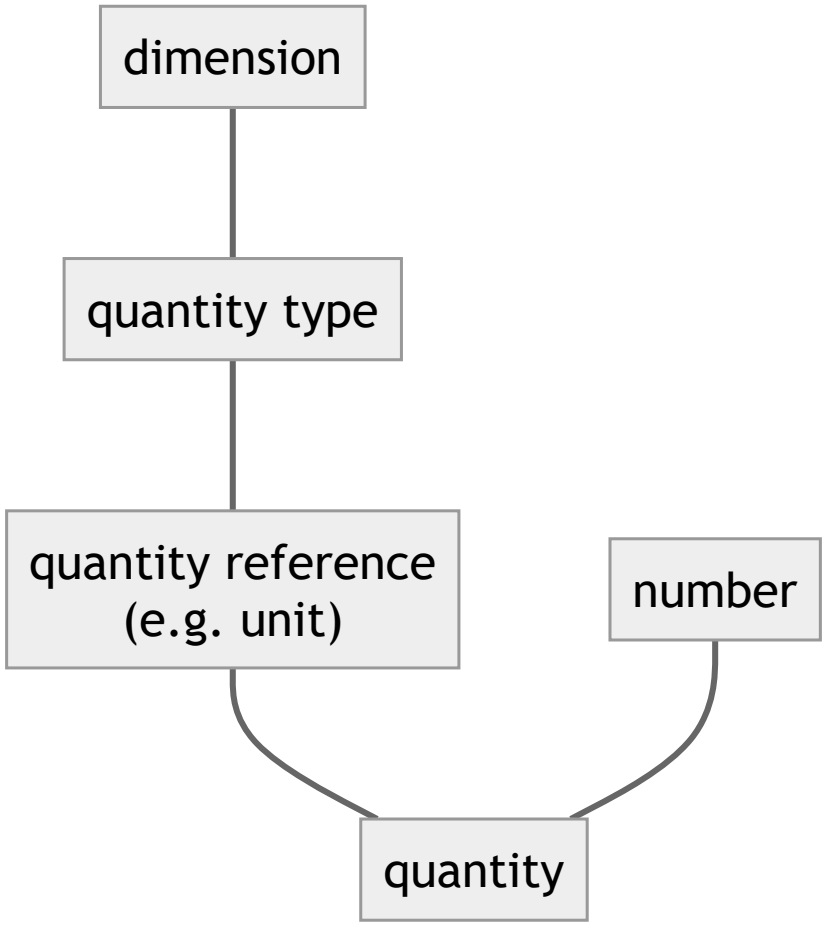
Quick domain introduction



QUANTITY TYPE

- Dimension is not enough to describe a quantity
- Different dimensions
  - *length*, *time*, *speed*, *power*
- The same dimension but a different kind
  - *work* vs. *torque*
  - *frequency* vs. *activity*
  - *area* vs. *fuel consumption*

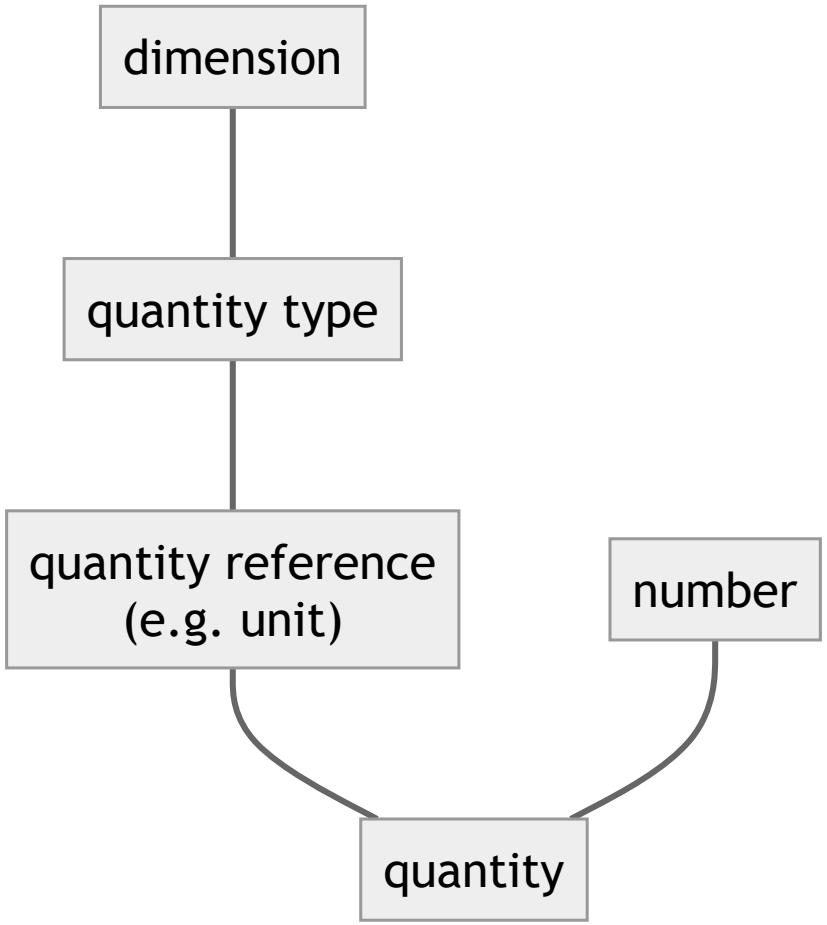
Quick domain introduction



QUANTITY TYPE

- Dimension is not enough to describe a quantity
- Different dimensions
  - *length*, *time*, *speed*, *power*
- The same dimension but a different kind
  - *work* vs. *torque*
  - *frequency* vs. *activity*
  - *area* vs. *fuel consumption*
- The same dimension and kind but still distinct
  - *radius* vs. *width* vs. *height*
  - *potential energy* vs *kinetic energy* vs *thermodynamic energy*

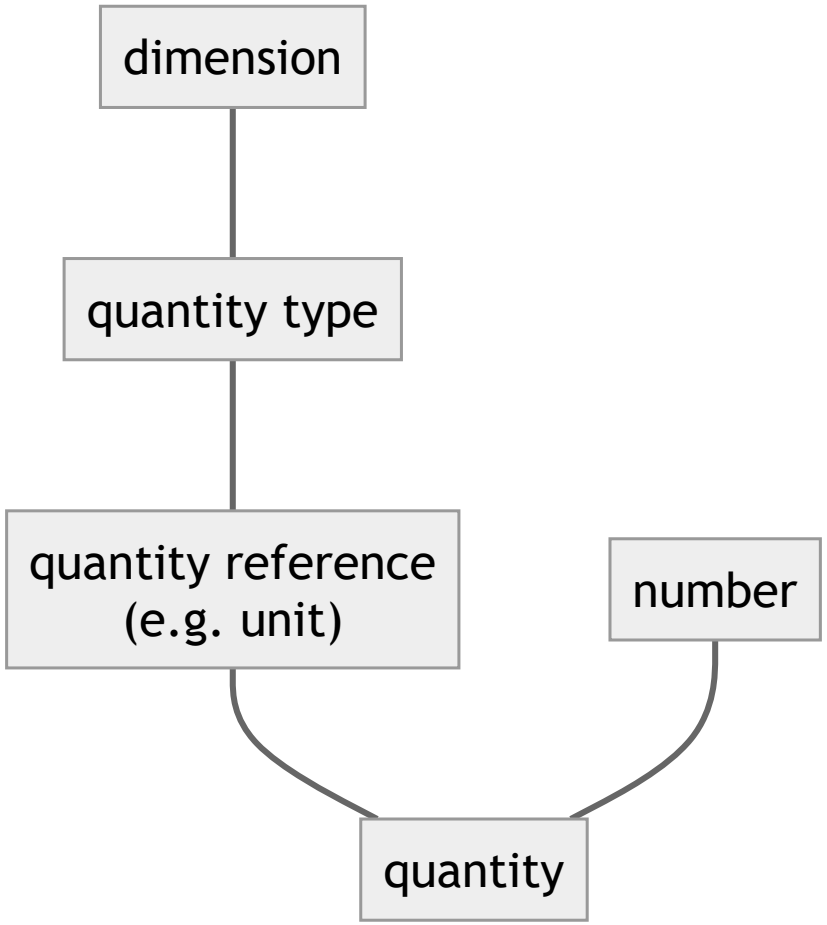
Quick domain introduction



QUANTITY REFERENCE

- Measurement unit
  - designated by conventionally assigned *name and symbol*
  - base SI units for all ISQ dimensions

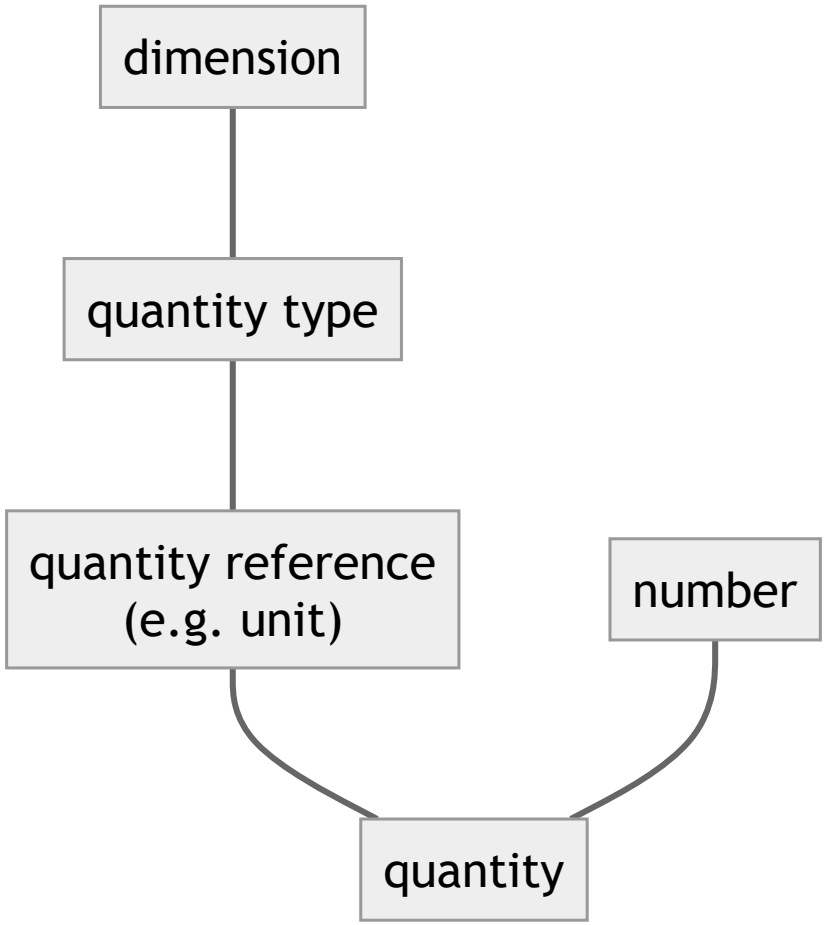
Quick domain introduction



QUANTITY REFERENCE

- Measurement unit
  - designated by conventionally assigned *name and symbol*
  - base SI units for all ISQ dimensions
  - J/K is a unit of *heat capacity* and *entropy*
  - $1/s$ is called hertz (Hz) when used for *frequencies* and becquerel (Bq) when used for *activities of radionuclides*

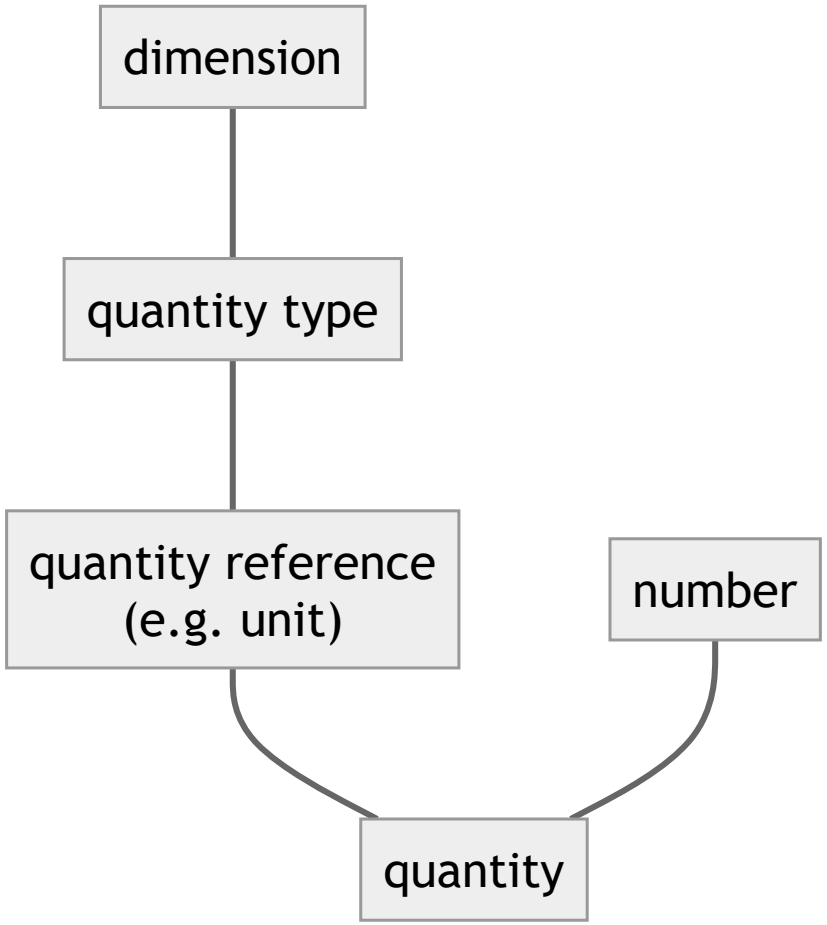
Quick domain introduction



QUANTITY REFERENCE

- Measurement unit
  - designated by conventionally assigned *name and symbol*
  - base SI units for all ISQ dimensions
  - J/K is a unit of *heat capacity* and *entropy*
  - $1/s$ is called hertz (Hz) when used for *frequencies* and becquerel (Bq) when used for *activities of radionuclides*
- Measurement procedure
- Reference material

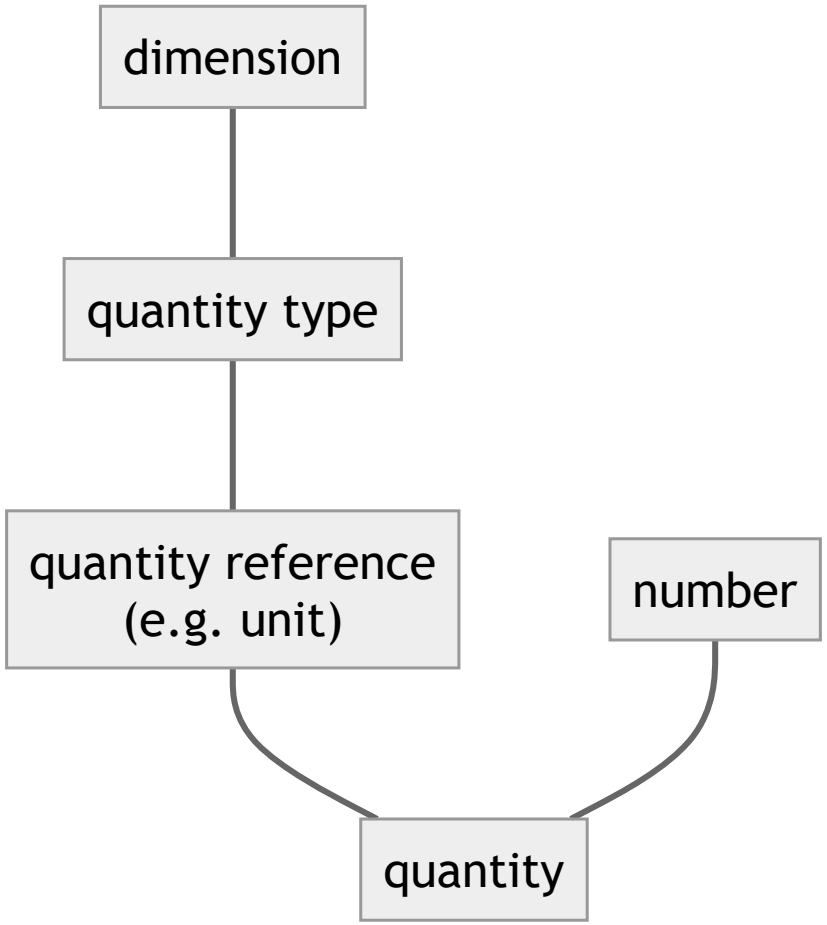
Quick domain introduction



QUANTITY

- Number and reference

Quick domain introduction



QUANTITY

- Number and reference
- Scalars (*length* of $123\ m$)
- Vectors (*force* of $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\ N$)
- Tensors (*stress* of $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\ Pa$)

Basic operations

```
// simple numeric operations
static_assert(10 * km / 2 == 5 * km);

// conversions to common units
static_assert(1 * h == 3600 * s);
static_assert(1 * km + 1 * m == 1001 * m);

// derived quantities
static_assert(1 * km / (1 * s) == 1000 * m / s);
static_assert(2 * km / h * (2 * h) == 4 * km);
static_assert(2 * km / (2 * km / h) == 1 * h);

static_assert(2 * m * (3 * m) == 6 * m2);

static_assert(10 * km / (5 * km) == 2 * one);

static_assert(1000 / (1 * s) == 1 * kHz);
```

A TASTE OF QUANTITIES AND UNITS LIBRARY

Proliferation of `double`

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * 0.44704;
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

Proliferation of double

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * 0.44704;
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

Proliferation of `double`

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * 0.44704;
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

Proliferation of double

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = MPH_TO_MPS(speed_mph);
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

```
#define KM_TO_M(v) ((v) * 1000.)
#define MI_TO_CM(v) ((v) * 2.54 * 12. * 5280)
#define MI_TO_M(v) (MI_TO_CM(v) / 100.)
#define H_TO_S(v) ((v) * 3600.)
#define KMPH_TO_MPS(v) (KM_TO_M(v) / H_TO_S(1.))
#define MPH_TO_MPS(v) (MI_TO_M(v) / H_TO_S(1.))
```

Proliferation of double

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * MPS_PER_MPH;
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

```
constexpr auto M_PER_KM = 1000.;
constexpr auto CM_PER_MI = 2.54 * 12. * 5280;
constexpr auto M_PER_MI = CM_PER_MI / 100.;
constexpr auto S_PER_H = 3600.;
constexpr auto MPS_PER_KMPH = M_PER_KM / S_PER_H;
constexpr auto MPS_PER_MPH = M_PER_MI / S_PER_H;
```

Safe unit conversions with mp-units

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * MPS_PER_MPH;
const double time_to_goal_s = distance_m / speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 2.68432 s
```

```
constexpr auto M_PER_KM = 1000.;
constexpr auto CM_PER_MI = 2.54 * 12. * 5280;
constexpr auto M_PER_MI = CM_PER_MI / 100.;
constexpr auto S_PER_H = 3600.;
constexpr auto MPS_PER_KMPH = M_PER_KM / S_PER_H;
constexpr auto MPS_PER_MPH = M_PER_MI / S_PER_H;
```

MP-UNITS

```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;

const quantity time_to_goal = (distance / speed).in(s);

std::println("TTG: {:{%N:.6} %U}", time_to_goal_s);
```

```
TTG: 2.68432 s
```

mp-units main goal is to generate errors

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * MPS_PER_MPH;
const double time_to_goal_s = distance_m * speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

MP-UNITS

```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;

const quantity time_to_goal = (distance * speed).in(s);

std::println("TTG: {:{%N:.6} %U}", time_to_goal_s);
```

mp-units main goal is to generate errors

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * MPS_PER_MPH;
const double time_to_goal_s = distance_m * speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 335.28 s
```

MP-UNITS

```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;

const quantity time_to_goal = (distance * speed).in(s);

std::println("TTG: {:{%N:.6} %U}", time_to_goal_s);
```

mp-units main goal is to generate errors

NO UNITS LIBRARY

```
const double distance_m = 30.;
const double speed_mph = 25.;
const double speed_mps = speed_mph * MPS_PER_MPH;
const double time_to_goal_s = distance_m * speed_mps;

std::println("TTG: {:.6} s", time_to_goal_s);
```

```
TTG: 335.28 s
```

MP-UNITS

```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;

const quantity time_to_goal = (distance * speed).in(s);

std::println("TTG: {:{%N:.6} %U}", time_to_goal_s);
```

```
error: no matching member function for call to 'in'
   36 | const quantity time_to_goal = (distance * speed).in(s);
      |                               ~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: constraints not satisfied [with U = struct second]
  173 |   [[nodiscard]] constexpr Quantity auto in(U) const
      |                                          ^
note: because 'UnitCompatibleWith<si::second, unit, quantity_spec>' evaluated to false
  171 |   template<UnitCompatibleWith<unit, quantity_spec> U>
      |            ^
note: because '!AssociatedUnit<si::second>' evaluated to false
  211 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::have_same_canonical_reference_unit(U{}, U2);
      |    ^
note: and 'UnitOf<si::second, kind_of_<derived_quantity_spec<power<length, 2>, per<time>>>{{{}}}>'
      evaluated to false
  211 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::have_same_canonical_reference_unit(U{}, U2);
      |                          ^
note: because 'implicitly_convertible(get_quantity_spec(si::second{}),
                kind_of_<derived_quantity_spec<power<length, 2>, per<time>>>{{{}}})'
      evaluated to false
  192 |   implicitly_convertible(get_quantity_spec(U{}), QS) &&
      |   ^
1 error generated.
Compiler returned: 1
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

```
TTG: 400 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

```
TTG: 400 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

```
TTG: 400 s
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_km, speed_kmph));
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_km, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_km, speed_kmph));
```

```
TTG: 0.4 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_km, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_km, speed_kmph));
```

```
TTG: 0.4 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_km, speed));
```

```
TTG: 400 s
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(speed_kmph, distance_m));
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(speed, distance_m));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(speed_kmph, distance_m));
```

```
TTG: 0.0324 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(speed, distance_m));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(speed_kmph, distance_m));
```

```
TTG: 0.0324 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(speed, distance_m));
```

```
error: could not convert 'speed' from 'quantity<derived_unit<si::kilo_<si::metre>, per<non_si::hour>>()>'
       to 'quantity<si::metre()>'
   50 |   std::println("TTG: {:{%N:.6} %U}",
      |               time_to_goal(speed, distance_m));
      |                                   ^~~~~
      |                                   |
      |                                   quantity<derived_unit<si::kilo_<si::metre>,
      |                                                         per<non_si::hour>>()>
Compiler returned: 1
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m * (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance * speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m * (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

```
TTG: 250000 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance * speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

Strong interfaces with mp-units

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m * (speed_kmph * MPS_PER_KMPH);
}
```

```
const double distance_km = 10.;
const double distance_m = distance_km * M_PER_KM;
const double speed_kmph = 90.;
std::println("TTG: {:.6} s",
             time_to_goal_s(distance_m, speed_kmph));
```

```
TTG: 250000 s
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance * speed;
}
```

```
const quantity distance_km = 10. * km;
const quantity distance_m = distance_km.in(m);
const quantity speed = 90. * km / h;
std::println("TTG: {:{%N:.6} %U}",
             time_to_goal(distance_m, speed));
```

```
error: could not convert 'operator*<si::metre(), double, derived_unit<si::kilo_<si::metre>,
                                    per<non_si::hour>>(), double>(distance, speed)'
      from 'quantity<derived_unit<si::kilo_<si::metre>, si::metre, per<non_si::hour>>(),[...]>'
      to 'quantity<si::second(),[...]>'
   27 |   return distance * speed;
      |          ~~~~~~~~~^~~~~~~
      |                   |
      |                   quantity<derived_unit<si::kilo_<si::metre>, si::metre,
      |                                         per<non_si::hour>>(),[...]>
Compiler returned: 1
```

As fast as double

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

As fast as double

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
time_to_goal_s(double, double):
        mulsd   xmm1, QWORD PTR .LC0[rip]
        divsd   xmm0, xmm1
        ret
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

As fast as double

NO UNITS LIBRARY

```
double time_to_goal_s(double distance_m,
                      double speed_kmph)
{
  return distance_m / (speed_kmph * MPS_PER_KMPH);
}
```

```
time_to_goal_s(double, double):
        mulsd   xmm1, QWORD PTR .LC0[rip]
        divsd   xmm0, xmm1
        ret
```

MP-UNITS

```
quantity<s> time_to_goal(quantity<m> distance,
                         quantity<km / h> speed)
{
  return distance / speed;
}
```

```
time_to_goal(quantity<si::metre{}, double>,
             quantity<derived_unit<si::kilo_<si::metre>,
                                   per<non_si::hour>>{}, double>):
        divsd   xmm0, xmm1
        mulsd   xmm0, QWORD PTR .LC0[rip]
        ret
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity distance_to_turn = 400. * ft;
const quantity car_speed = 40. * mi / h;
const quantity ttg = time_to_goal(distance_to_turn, car_speed);
std::println("Turn right after {:{%N:.1} %U}", ttg.in(s));
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity distance_to_turn = 400. * ft;
const quantity car_speed = 40. * mi / h;
const quantity ttg = time_to_goal(distance_to_turn, car_speed);
std::println("Turn right after {:{%N:.1} %U}", ttg.in(s));
```

```
Turn right after 6.8 s
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity distance_to_garda_lake = 1512. * km;
const quantity avg_speed = 95. * km / h;
const quantity ttg = time_to_goal(distance_to_garda_lake, avg_speed);
std::println("Travel time to Garda lake {:{%N:.1} %U}", ttg.in(h));
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity distance_to_garda_lake = 1512. * km;
const quantity avg_speed = 95. * km / h;
const quantity ttg = time_to_goal(distance_to_garda_lake, avg_speed);
std::println("Travel time to Garda lake {:{%N:.1} %U}", ttg.in(h));
```

```
Travel time to Garda lake 15.9 h
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity half_marathon_distance = 21.0975 * km;
const quantity pace = (4. * min + 40. * s) / km;
const quantity ttg = time_to_goal(half_marathon_distance, pace);
std::chrono::seconds seconds = value_cast<si::second, std::int64_t>(ttg);
std::println("Expected race time {:%T}", seconds);
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity half_marathon_distance = 21.0975 * km;
const quantity pace = (4. * min + 40. * s) / km;
const quantity ttg = time_to_goal(half_marathon_distance, pace);
std::chrono::seconds seconds = value_cast<si::second, std::int64_t>(ttg);
std::println("Expected race time {:%T}", seconds);
```

```
error: no matching function for call to 'time_to_goal'
   22 | const quantity ttg = time_to_goal(half_marathon_distance, pace);
      |                      ^~~~~~~~~~~~
note: candidate template ignored: constraints not satisfied [with distance:auto = quantity<kilo_<metre>{}, double>,
                                                               speed:auto = quantity<derived_unit<second, per<kilo_<metre>>>{}, double>]
   12 | QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
      |                           ^
note: because 'QuantityOf<quantity<derived_unit<si::second, per<si::kilo_<si::metre>>>{{{}}}>, isq::speed>' evaluated to false
   13 |                                         QuantityOf<isq::speed> auto speed)
      |                                         ^
note: because 'QuantitySpecOf<std::remove_const_t<decltype(get_quantity_spec(derived_unit<second, per<kilo_<metre>>>{}))>, struct speed{{{}}}>' evaluated to false
   77 |                     QuantitySpecOf<std::remove_const_t<decltype(get_quantity_spec(T{}))>, V>);
      |                     ^
note: because 'implicitly_convertible(kind_of_<derived_quantity_spec<isq::time, per<isq::length>>>{}, struct speed{{{}}})' evaluated to false
  147 |   QuantitySpec<T> && QuantitySpec<std::remove_const_t<decltype(QS)>> && implicitly_convertible(T{}, QS) &&
      |                                                                         ^
1 error generated.
Compiler returned: 1
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity half_marathon_distance = 21.0975 * km;
const quantity pace = (4. * min + 40. * s) / km;
const quantity ttg = time_to_goal(half_marathon_distance, 1 / pace);
std::chrono::seconds seconds = value_cast<si::second, std::int64_t>(ttg);
std::println("Expected race time {:%T}", seconds);
```

```
Expected race time 01:38:27
```

Power of generic programming

```
QuantityOf<isq::time> auto time_to_goal(QuantityOf<isq::length> auto distance,
                                        QuantityOf<isq::speed> auto speed)
{
  return distance / speed;
}
```

```
const quantity_point cloud_base(1850. * m);
const quantity_point glider_alt(1200. * m);
const quantity thermal_strength = 4.5 * m / s;
const quantity ttg = time_to_goal(cloud_base - glider_alt, thermal_strength);
std::println("Exit thermal in {}", value_cast<int>(ttg).in(s));
```

```
Exit thermal in 144 s
```

SAFETY FEATURES

Safe unit conversions

```
auto q1 = 5 * km;
std::cout << q1.in(m) << '\n';
quantity<si::metre, int> q2 = q1;
```

Safe unit conversions

```
auto q1 = 5 * km;
std::cout << q1.in(m) << '\n';
quantity<si::metre, int> q2 = q1;
```

- Magnitudes of the source and destination units are known at compile time
- The library uses that information to calculate and apply a conversion factor automatically for the user

Unit definitions

```
namespace si {

template<PrefixableUnit U> struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};
template<PrefixableUnit auto U> inline constexpr kilo_<std::remove_const_t<decltype(U)>> kilo;









}
```

Unit definitions

```
namespace si {

template<PrefixableUnit U> struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};
template<PrefixableUnit auto U> inline constexpr kilo_<std::remove_const_t<decltype(U)>> kilo;

inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second : named_unit<"s", kind_of<isq::time>> {} second;









}
```

Unit definitions

```
namespace si {

template<PrefixableUnit U> struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};
template<PrefixableUnit auto U> inline constexpr kilo_<std::remove_const_t<decltype(U)>> kilo;

inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second : named_unit<"s", kind_of<isq::time>> {} second;

namespace unit_symbols {

inline constexpr auto m = metre;
inline constexpr auto km = kilo<metre>;

}
}
```

Unit definitions

```
namespace si {

template<PrefixableUnit U> struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};
template<PrefixableUnit auto U> inline constexpr kilo_<std::remove_const_t<decltype(U)>> kilo;

inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second : named_unit<"s", kind_of<isq::time>> {} second;

namespace unit_symbols {

inline constexpr auto m = metre;
inline constexpr auto km = kilo<metre>;

}
}
```

`unit_symbols` are opt-in as they may easily collide with user's identifiers.

Unit definitions

```
namespace non_si {

inline constexpr struct minute : named_unit<"min", mag<60> * si::second> {} minute;
inline constexpr struct hour : named_unit<"h", mag<60> * minute> {} hour;

namespace unit_symbols {

inline constexpr auto min = minute;
inline constexpr auto h = hour;

}
}
```

Unit definitions

```
namespace non_si {

inline constexpr struct minute : named_unit<"min", mag<60> * si::second> {} minute;
inline constexpr struct hour : named_unit<"h", mag<60> * minute> {} hour;

namespace unit_symbols {

inline constexpr auto min = minute;
inline constexpr auto h = hour;

}
}
```

```
namespace international {

inline constexpr struct yard : named_unit<"yd", mag<ratio{9'144, 10'000}> * si::metre> {} yard;
inline constexpr struct mile : named_unit<"mi", mag<1760> * yard> {} mile;

namespace unit_symbols {

inline constexpr auto yd = yard;
inline constexpr auto mi = mile;

}
}
```

Limitations of `std::ratio`

In **`std::chrono::duration`**, the magnitude of a unit is always expressed with **`std::ratio`**. This is not enough for a general-purpose physical units library.

Limitations of `std::ratio`

In **`std::chrono::duration`**, the magnitude of a unit is always expressed with **`std::ratio`**. This is not enough for a general-purpose physical units library.

- **`std::ratio`** is implemented in terms of **`std::intmax_t`**
- Impossible to define *units with huge or tiny magnitudes*
  - electronvolt ($1\ eV = 1.602176634 \times 10^{-19}\ J$)
  - Dalton ($1\ Da = 1.660539040(20) \times 10^{-27}\ kg$)
- Some conversions require a conversion factor based on an *irrational number* like pi
  - radian <=> degree

Preventing truncation of data

Preventing truncation of data

```
quantity q1 = 5 * m;
std::cout << q1.in(km) << '\n';                // Compile-time error
quantity<si::kilo<si::metre>, int> q2 = q1;  // Compile-time error
```

Preventing truncation of data

```
quantity q1 = 5 * m;
std::cout << q1.in(km) << '\n';                // Compile-time error
quantity<si::kilo<si::metre>, int> q2 = q1;  // Compile-time error
```

Conversion of a quantity with the integral representation type to one with a unit of a lower resolution is truncating.

Preventing truncation of data

```
quantity q1 = 5 * m;
std::cout << q1.in(km) << '\n';                // Compile-time error
quantity<si::kilo<si::metre>, int> q2 = q1;    // Compile-time error
```

- Floating-point representation type is considered value-preserving

```
quantity q1 = 5. * m;     // source quantity uses double as a representation type
std::cout << q1.in(km) << '\n';
quantity<si::kilo<si::metre>> q2 = q1;
```

Preventing truncation of data

```
quantity q1 = 5 * m;
std::cout << q1.in(km) << '\n';                // Compile-time error
quantity<si::kilo<si::metre>, int> q2 = q1;    // Compile-time error
```

- Floating-point representation type is considered value-preserving

```
quantity q1 = 5. * m;     // source quantity uses double as a representation type
std::cout << q1.in(km) << '\n';
quantity<si::kilo<si::metre>> q2 = q1;
```

```
quantity q1 = 5 * m;      // source quantity uses int as a representation type
std::cout << value_cast<double>(q1).in(km) << '\n';
quantity<si::kilo<si::metre>> q2 = q1;  // double by default
```

Preventing truncation of data

```
quantity q1 = 5 * m;
std::cout << q1.in(km) << '\n';                // Compile-time error
quantity<si::kilo<si::metre>, int> q2 = q1;  // Compile-time error
```

- Truncating conversion can be **explicitly forced** from the code

```
quantity q1 = 5 * m;     // source quantity uses int as a representation type
std::cout << q1.force_in(km) << '\n';
quantity<si::kilo<si::metre>, int> q2 = value_cast<km>(q1);
```

Preventing truncation of data

```
quantity q1 = 2.5 * m;
quantity<si::metre, int> q2 = q1;  // Compile-time error
```

Preventing truncation of data

```
quantity q1 = 2.5 * m;
quantity<si::metre, int> q2 = q1;  // Compile-time error
```

Assigning a quantity with a floating-point representation type to the one using an integral representation type is considered truncating.

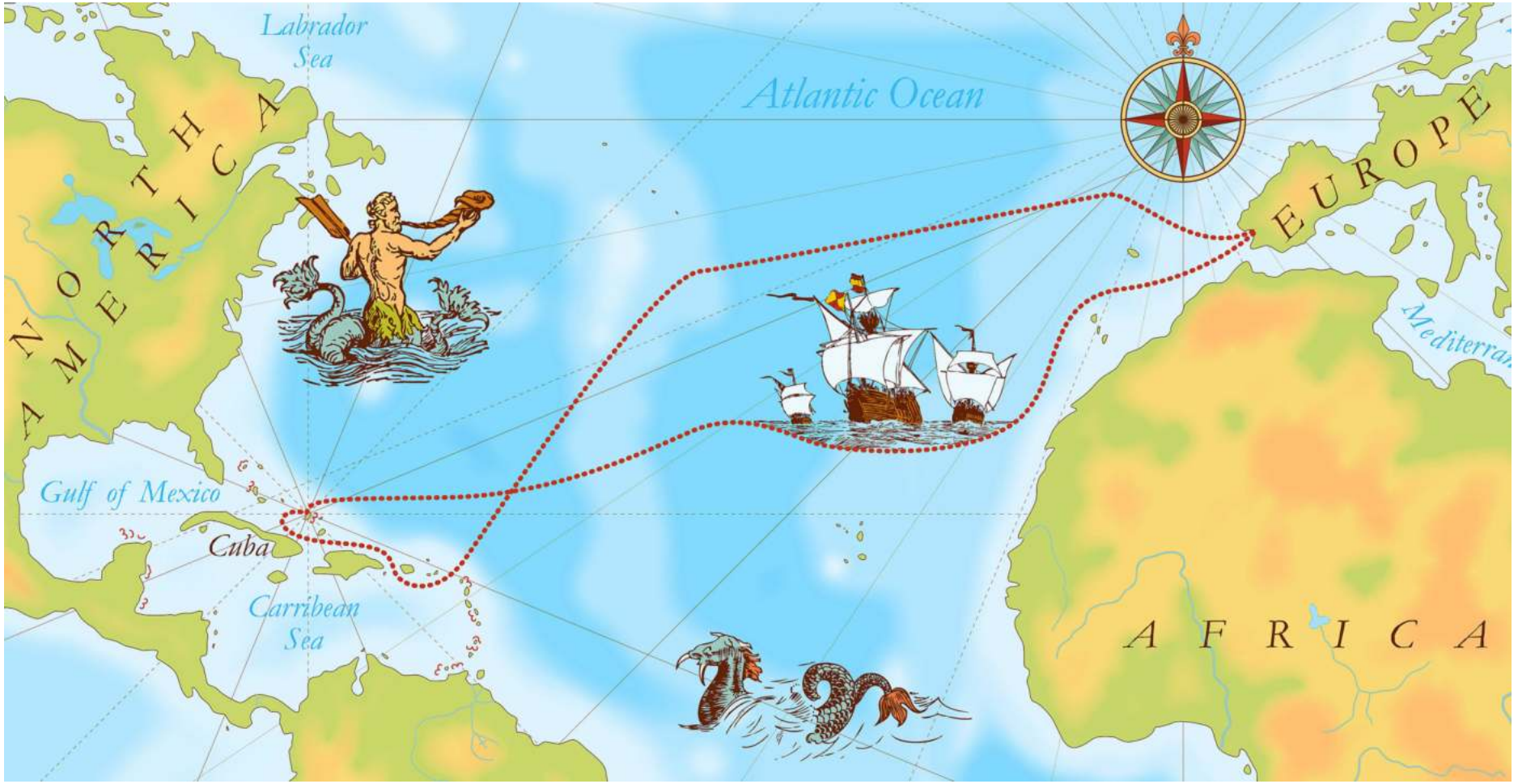
Preventing truncation of data

```
quantity q1 = 2.5 * m;
quantity<si::metre, int> q2 = q1;  // Compile-time error
```

- Truncating conversion can be **explicitly forced** from the code

```
quantity q1 = 2.5 * m;
quantity<si::metre, int> q2 = value_cast<int>(q1);
```

Why Columbus thought that he reached India?



Why Columbus thought that he reached India?

- Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus) was a *medieval Persian geographer*
- Columbus learned that Alfraganus estimated **degree of latitude to span 56.67 miles**
  - degreee of longitude at the equator should be roughly equivalent

Why Columbus thought that he reached India?

ROMAN FOOT (PES)

- The length of the foot on the statue of

  Cossutius

Why Columbus thought that he reached India?

ROMAN FOOT (PES)

- The length of the foot on the statue of Cossutius

ROMAN PACE

- The length of every other step of a Roman legionary
- 5 Roman feet

Why Columbus thought that he reached India?

ROMAN FOOT (PES)

- The length of the foot on the statue of Cossutius

ROMAN PACE

- The length of every other step of a Roman legionary
- 5 Roman feet

ROMAN MILE

- Total distance of the left foot of Roman legionaries hitting the ground 1,000 times

Why Columbus thought that he reached India?

```
// length of degree of latitude estimation by medieval Persian geographer
// Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus)
// (degreee of longitude at the equator should be roughly equivalent)
template<UnitOf<isq::length> auto Mile>
struct estimated_degree : named_unit<"deg", mag<ratio{5667, 100}> * Mile> {};
```

Why Columbus thought that he reached India?

```
// length of degree of latitude estimation by medieval Persian geographer
// Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus)
// (degreee of longitude at the equator should be roughly equivalent)
template<UnitOf<isq::length> auto Mile>
struct estimated_degree : named_unit<"deg", mag<ratio{5667, 100}> * Mile> {};
```

```
inline constexpr struct roman_foot : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile : named_unit<"mi_r", mag<1000> * roman_pace> {} roman_mile;
```

Why Columbus thought that he reached India?

```
// length of degree of latitude estimation by medieval Persian geographer
// Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus)
// (degreee of longitude at the equator should be roughly equivalent)
template<UnitOf<isq::length> auto Mile>
struct estimated_degree : named_unit<"deg", mag<ratio{5667, 100}> * Mile> {};
```

```
inline constexpr struct roman_foot : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile : named_unit<"mi_r", mag<1000> * roman_pace> {} roman_mile;
```

```
// used in Persia
// extended the Roman mile to fit an astronomical approximation of 1 minute of an arc of latitude
inline constexpr struct arabic_mile : named_unit<"mi_a", mag<2163> * si::metre> {} arabic_mile;
```

Why Columbus thought that he reached India?

```
// length of degree of latitude estimation by medieval Persian geographer
// Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus)
// (degreee of longitude at the equator should be roughly equivalent)
template<UnitOf<isq::length> auto Mile>
struct estimated_degree : named_unit<"deg", mag<ratio{5667, 100}> * Mile> {};
```

```
inline constexpr struct roman_foot : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile : named_unit<"mi_r", mag<1000> * roman_pace> {} roman_mile;
```

```
// used in Persia
// extended the Roman mile to fit an astronomical approximation of 1 minute of an arc of latitude
inline constexpr struct arabic_mile : named_unit<"mi_a", mag<2163> * si::metre> {} arabic_mile;
```

```
// 1 minute of arc along the Earth's equator
inline constexpr struct geographical_mile : named_unit<"mi_g", mag<ratio{18553, 10}> * si::metre> {} geographical_mile;
```

Why Columbus thought that he reached India?

```
// length of degree of latitude estimation by medieval Persian geographer
// Abu al Abbas Ahmad ibn Muhammad ibn Kathir al-Farghani (a.k.a. Alfraganus)
// (degreee of longitude at the equator should be roughly equivalent)
template<UnitOf<isq::length> auto Mile>
struct estimated_degree : named_unit<"deg", mag<ratio{5667, 100}> * Mile> {};
```

```
inline constexpr struct roman_foot : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile : named_unit<"mi_r", mag<1000> * roman_pace> {} roman_mile;
```

```
// used in Persia
// extended the Roman mile to fit an astronomical approximation of 1 minute of an arc of latitude
inline constexpr struct arabic_mile : named_unit<"mi_a", mag<2163> * si::metre> {} arabic_mile;
```

```
// 1 minute of arc along the Earth's equator
inline constexpr struct geographical_mile : named_unit<"mi_g", mag<ratio{18553, 10}> * si::metre> {} geographical_mile;
```

```
inline constexpr struct degree : named_unit<"deg", mag<60> * geographical_mile> {} degree;
inline constexpr auto Alfraganus_degree = estimated_degree<arabic_mile>{};
inline constexpr auto Columbus_degree = estimated_degree<roman_mile>{};
```

Why Columbus thought that he reached India?

```
template<Quantity Q1, Quantity Q2>
  requires std::invocable<std::minus<>, Q1, Q2>
quantity<percent> error(const Q1& approximate, const Q2& exact)
{
  return abs(approximate - exact) / exact;
}
```

Why Columbus thought that he reached India?

```
template<Quantity Q1, Quantity Q2>
  requires std::invocable<std::minus<>, Q1, Q2>
quantity<percent> error(const Q1& approximate, const Q2& exact)
{
  return abs(approximate - exact) / exact;
}
```

```
std::cout << "Roman mile: " << (1. * roman_mile).in(si::metre) << "\n";
std::cout << "Arabic mile: " << (1. * arabic_mile).in(si::metre) << "\n";
std::cout << "Mile error: " << error(1. * roman_mile, 1. * arabic_mile) << "\n";
```

Why Columbus thought that he reached India?

```
template<Quantity Q1, Quantity Q2>
  requires std::invocable<std::minus<>, Q1, Q2>
quantity<percent> error(const Q1& approximate, const Q2& exact)
{
  return abs(approximate - exact) / exact;
}
```

```
std::cout << "Roman mile: " << (1. * roman_mile).in(si::metre) << "\n";
std::cout << "Arabic mile: " << (1. * arabic_mile).in(si::metre) << "\n";
std::cout << "Mile error: " << error(1. * roman_mile, 1. * arabic_mile) << "\n";
```

```
Roman mile: 1480 m
Arabic mile: 2163 m
Mile error: 31.5765%
```

Why Columbus thought that he reached India?

```
const quantity Columbus_equator_length = 360. * Columbus_degree;
const quantity Alfraganus_equator_length = 360. * Alfraganus_degree;
const quantity equator_length = 360. * degree;

std::cout << "Columbus equator length: " << Columbus_equator_length.in(nmi) << "\n";
std::cout << "Alfraganus equator length: " << Alfraganus_equator_length.in(nmi) << "\n";
std::cout << "Equator length: " << equator_length.in(nmi) << "\n";
std::cout << "Equator error: " << error(Columbus_equator_length, equator_length) << "\n";
```

Why Columbus thought that he reached India?

```
const quantity Columbus_equator_length = 360. * Columbus_degree;
const quantity Alfraganus_equator_length = 360. * Alfraganus_degree;
const quantity equator_length = 360. * degree;

std::cout << "Columbus equator length: " << Columbus_equator_length.in(nmi) << "\n";
std::cout << "Alfraganus equator length: " << Alfraganus_equator_length.in(nmi) << "\n";
std::cout << "Equator length: " << equator_length.in(nmi) << "\n";
std::cout << "Equator error: " << error(Columbus_equator_length, equator_length) << "\n";
```

```
Columbus equator length: 16303.3 nmi
Alfraganus equator length: 23827.1 nmi
Equator length: 21638.8 nmi
Equator error: 24.6569%
```

Why Columbus thought that he reached India?

Why Columbus thought that he reached India?

"six parts are habitable and the seventh is covered with water."

-- 2 Esdras

Why Columbus thought that he reached India?

```
const quantity Columbus_distance = 68. * Columbus_degree;
```

Why Columbus thought that he reached India?

```
const quantity Columbus_distance = 68. * Columbus_degree;
```

```
const quantity Tenerife_Bahamas_distance = 5'982. * km;
const quantity Tenerife_Japan_distance = 10'600. * nmi;

std::cout << "Columbus distance: " << Columbus_distance.in(nmi) << "\n";
std::cout << "Tenerife-Bahamas distance: " << Tenerife_Bahamas_distance.in(nmi) << "\n";
std::cout << "Tenerife-Japan distance: " << Tenerife_Japan_distance.in(nmi) << "\n";
std::cout << "Distance error: " << error(Columbus_distance, Tenerife_Japan_distance) << "\n";
```

Why Columbus thought that he reached India?

```
const quantity Columbus_distance = 68. * Columbus_degree;
```

```
const quantity Tenerife_Bahamas_distance = 5'982. * km;
const quantity Tenerife_Japan_distance = 10'600. * nmi;

std::cout << "Columbus distance: " << Columbus_distance.in(nmi) << "\n";
std::cout << "Tenerife-Bahamas distance: " << Tenerife_Bahamas_distance.in(nmi) << "\n";
std::cout << "Tenerife-Japan distance: " << Tenerife_Japan_distance.in(nmi) << "\n";
std::cout << "Distance error: " << error(Columbus_distance, Tenerife_Japan_distance) << "\n";
```

```
Columbus distance: 3079.52 nmi
Tenerife-Bahamas distance: 3230.02 nmi
Tenerife-Japan distance: 10600 nmi
Distance error: 70.9479%
```

Why Columbus thought that he reached India?

Code correct by construction

Thanks to the usage of quantities and units library a developer has to focus only on a program logic and does not have to carefully verify every unit conversion and quantity arithmetics.

Code correct by construction

Thanks to the usage of quantities and units library a developer has to focus only on a program logic and does not have to carefully verify every unit conversion and quantity arithmetics.

Imagine that you have to implement the Columbus's story with conversion macros...

`explicit` is not explicit enough

TODAY

```
struct X {
  std::vector<std::chrono::milliseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42);
```

`explicit` is not explicit enough

TODAY

```
struct X {
  std::vector<std::chrono::milliseconds> vec;
  // ...
};
```

TOMORROW

```
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42);
```

`explicit` is not explicit enough

TODAY

```
struct X {
  std::vector<std::chrono::milliseconds> vec;
  // ...
};
```

TOMORROW

```
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42);
```

The code continues to compile fine, but all the calculations are now off by orders of magnitude.

`explicit` is not explicit enough

TODAY

```
struct X {
  std::vector<quantity<si::milli<si::second>>> vec;
  // ...
};
```

TOMORROW

```
struct X {
  std::vector<quantity<si::micro<si::second>>> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42);       // Compile-time error
x.vec.emplace_back(42 * ms);  // OK
```

`explicit` is not explicit enough

TODAY

```
struct X {
  std::vector<quantity<si::milli<si::second>>> vec;
  // ...
};
```

TOMORROW

```
struct X {
  std::vector<quantity<si::micro<si::second>>> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42);        // Compile-time error
x.vec.emplace_back(42 * ms);   // OK
```

Quantity construction in mp-units always requires both a number and a unit.

Safe quantity numerical value getters

```
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

Safe quantity numerical value getters

```
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

std::chrono::duration

```
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42s);
legacy_func(x.vec[0].count());
```

Safe quantity numerical value getters

```cpp
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

std::chrono::duration

```cpp
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```cpp
X x;
x.vec.emplace_back(42s);
legacy_func(x.vec[0].count());
```

```
42000000 s
```

Safe quantity numerical value getters

```
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

std::chrono::duration

```
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42s);
legacy_func(duration_cast<seconds>(x.vec[0]).count());
```

```
42 s
```

Safe quantity numerical value getters

```
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

std::chrono::duration

```
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42s);
legacy_func(duration_cast<seconds>(x.vec[0]).count());
```

```
42 s
```

MP-UNITS

```
struct X {
  std::vector<quantity<si::micro<si::second>>> vec;
  // ...
};
```

```
X x;
x.vec.emplace_back(42 * s);
legacy_func(x.vec[0].numerical_value_in(si::second));
```

```
42 s
```

Safe quantity numerical value getters

```cpp
void legacy_func(std::int64_t seconds)
{
  std::cout << seconds << " s\n";
}
```

std::chrono::duration

```cpp
struct X {
  std::vector<std::chrono::microseconds> vec;
  // ...
};
```

```cpp
X x;
x.vec.emplace_back(42s);
legacy_func(duration_cast<seconds>(x.vec[0]).count());
```

```
42 s
```

MP-UNITS

```cpp
struct X {
  std::vector<quantity<si::micro<si::second>>> vec;
  // ...
};
```

```cpp
X x;
x.vec.emplace_back(42 * s);
legacy_func(x.vec[0].numerical_value_in(si::second));
```

```
42 s
```

Numerical value getters in mp-units always require a unit as an argument.

Do you remember Clarence the tortoise?

How do you like such an interface?

NO UNITS LIBRARY

```
class Box {
  double length_m_;
  double width_m_;
  double height_m_;
public:
  Box(double l_m, double w_m, double h_m)
    : length_m_(l_m), width_m_(w_m), height_m_(h_m)
  {}

  double base_m2() const { return length_m_ * width_m_; }
  // ...
};
```

```
Box my_box(2, 3, 1);
```

How do you like such an interface?

NO UNITS LIBRARY

```
class Box {
  double length_m_;
  double width_m_;
  double height_m_;
public:
  Box(double l_m, double w_m, double h_m)
    : length_m_(l_m), width_m_(w_m), height_m_(h_m)
  {}

  double base_m2() const { return length_m_ * width_m_; }
  // ...
};
```

```
Box my_box(2, 3, 1);
```

USING A TYPICAL UNITS LIBRARY

```
class Box {
  length length_;
  length width_;
  length height_;
public:
  Box(length l, length w, length h)
    : length_(l), width_(w), height_(h)
  {}

  area base() const { return length_ * width_; }
  // ...
};
```

```
Box my_box(2 * m, 3 * m, 1 * m);
```

How do you like such an interface?

NO UNITS LIBRARY

```
class Box {
  double length_m_;
  double width_m_;
  double height_m_;
public:
  Box(double l_m, double w_m, double h_m)
    : length_m_(l_m), width_m_(w_m), height_m_(h_m)
  {}

  double base_m2() const { return length_m_ * width_m_; }
  // ...
};
```

```
Box my_box(2, 3, 1);
```

USING A TYPICAL UNITS LIBRARY

```
class Box {
  length length_;
  length width_;
  length height_;
public:
  Box(length l, length w, length h)
    : length_(l), width_(w), height_(h)
  {}

  area base() const { return length_ * width_; }
  // ...
};
```

```
Box my_box(2 * m, 3 * m, 1 * m);
```

Nearly none of the libraries on the market guarantee type safety for different quantities of the same kind.

Proper direction matters!

ISQ base dimensions

```
namespace mp_units::isq {

inline constexpr struct dim_length : base_dimension<"L"> {} dim_length;
inline constexpr struct dim_mass : base_dimension<"M"> {} dim_mass;
inline constexpr struct dim_time : base_dimension<"T"> {} dim_time;
inline constexpr struct dim_electric_current : base_dimension<"I"> {} dim_electric_current;
inline constexpr struct dim_thermodynamic_temperature : base_dimension<{"Θ", "O"}> {} dim_thermodynamic_temperature;
inline constexpr struct dim_amount_of_substance : base_dimension<"N"> {} dim_amount_of_substance;
inline constexpr struct dim_luminous_intensity : base_dimension<"J"> {} dim_luminous_intensity;

}
```

ISQ base dimensions

```
namespace mp_units::isq {

inline constexpr struct dim_length : base_dimension<"L"> {} dim_length;
inline constexpr struct dim_mass : base_dimension<"M"> {} dim_mass;
inline constexpr struct dim_time : base_dimension<"T"> {} dim_time;
inline constexpr struct dim_electric_current : base_dimension<"I"> {} dim_electric_current;
inline constexpr struct dim_thermodynamic_temperature : base_dimension<{"Θ", "Θ"}> {} dim_thermodynamic_temperature;
inline constexpr struct dim_amount_of_substance : base_dimension<"N"> {} dim_amount_of_substance;
inline constexpr struct dim_luminous_intensity : base_dimension<"J"> {} dim_luminous_intensity;

}
```

Derived dimensions are never explicitly defined in the library. They are transitively created with the derived quantities definitions.

Introducing `quantity_spec`

Dimension is not enough to specify all properties of a quantity.

Introducing `quantity_spec`

Dimension is not enough to specify all properties of a quantity.

- **More than one quantity** may be defined **for the same dimension**
  - quantities of *different kinds* (e.g., *frequency*, *modulation rate*, *activity*, ...)
  - quantities of the *same kind* (e.g., *length*, *width*, *altitude*, *distance*, *radius*, *wavelength*, *position vector*, ...)

Introducing `quantity_spec`

Dimension is not enough to specify all properties of a quantity.

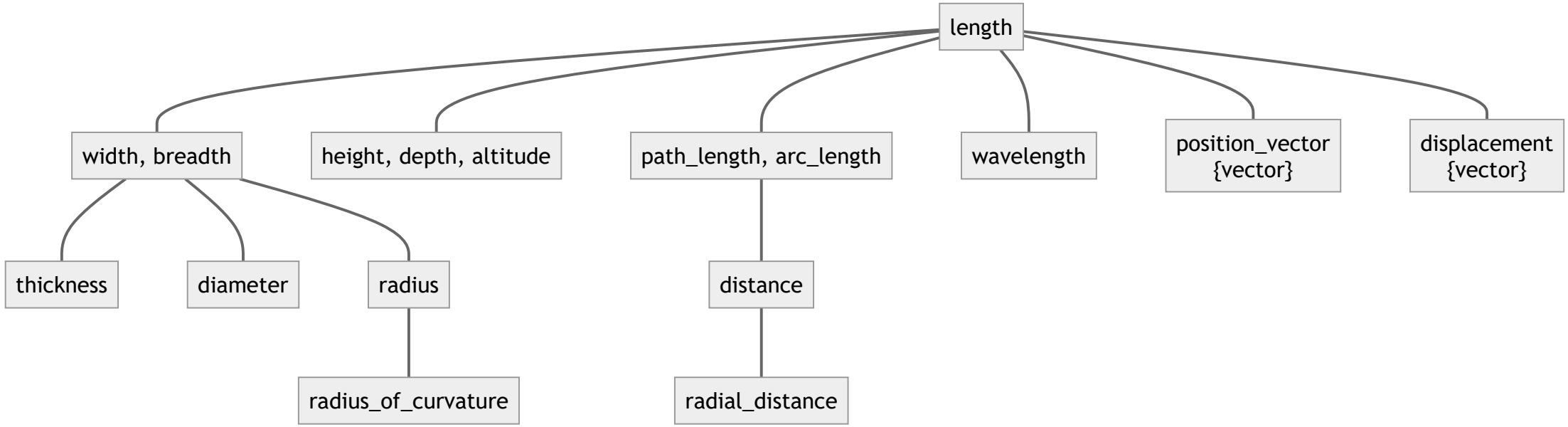
- **More than one quantity** may be defined **for the same dimension**
  - quantities of *different kinds* (e.g., *frequency*, *modulation rate*, *activity*, ...)
  - quantities of the *same kind* (e.g., *length*, *width*, *altitude*, *distance*, *radius*, *wavelength*, *position vector*, ...)
- Quantities may have **different character**
  - scalars
  - vectors
  - tensors

Introducing `quantity_spec`

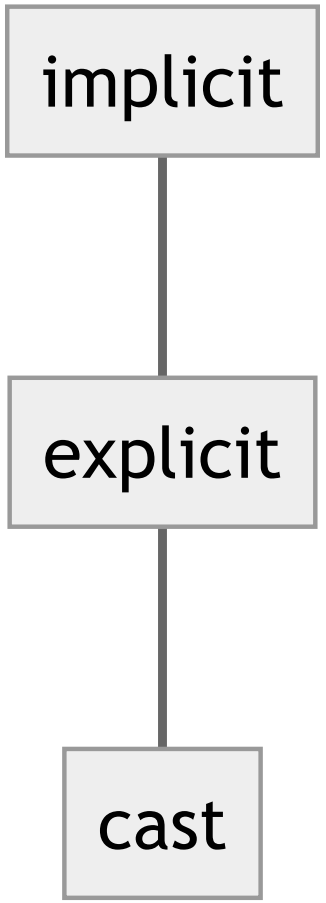
Dimension is not enough to specify all properties of a quantity.

- **More than one quantity** may be defined **for the same dimension**
  - quantities of *different kinds* (e.g., *frequency*, *modulation rate*, *activity*, ...)
  - quantities of the *same kind* (e.g., *length*, *width*, *altitude*, *distance*, *radius*, *wavelength*, *position vector*, ...)
- Quantities may have **different character**
  - scalars
  - vectors
  - tensors
- Quantities may be defined as **non-negative**

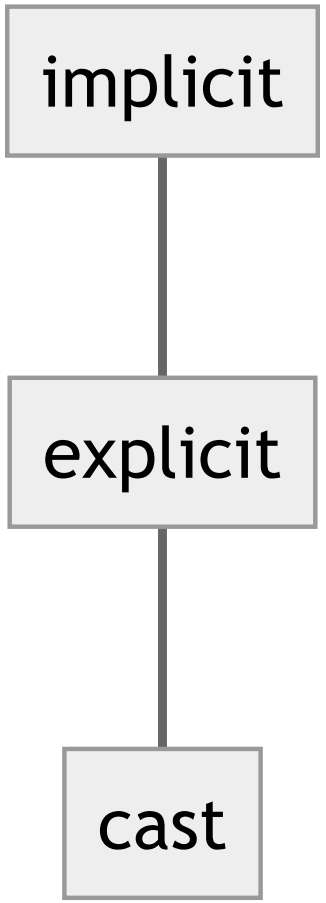
ISQ: International System of Quantities (ISO 80000)

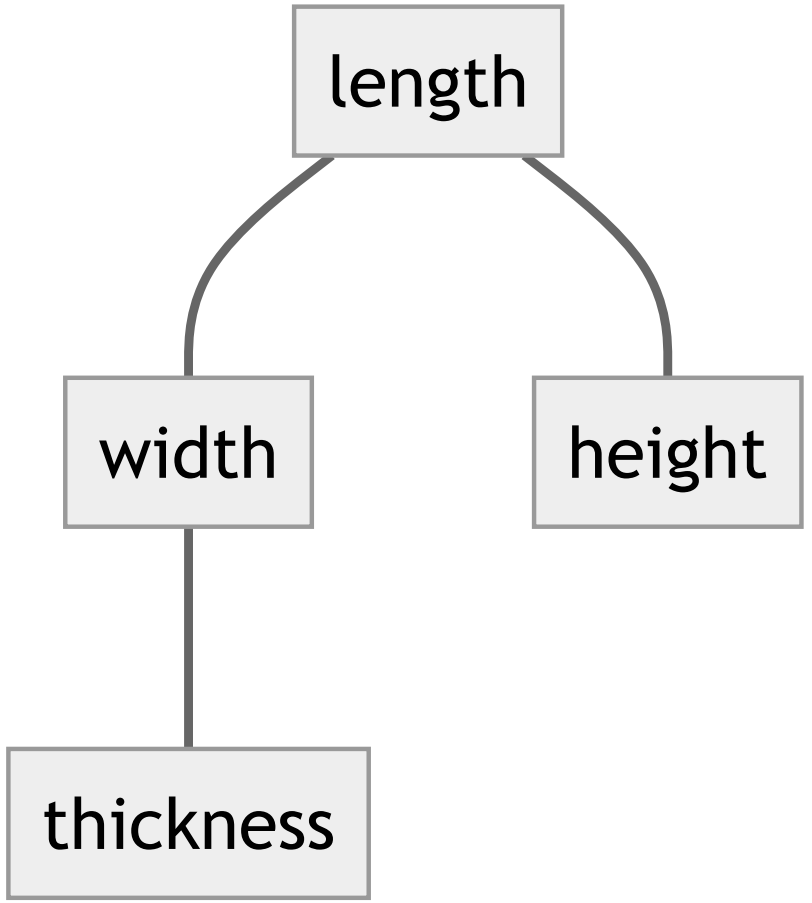


Three levels of quantity conversions



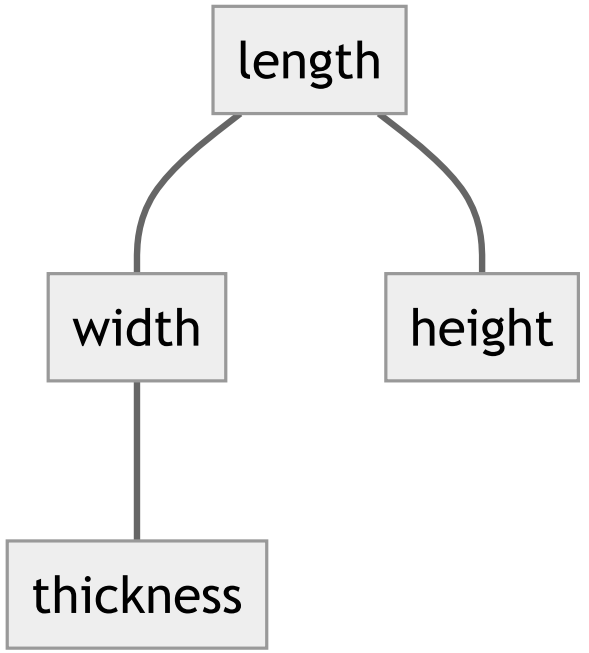
Three levels of quantity conversions





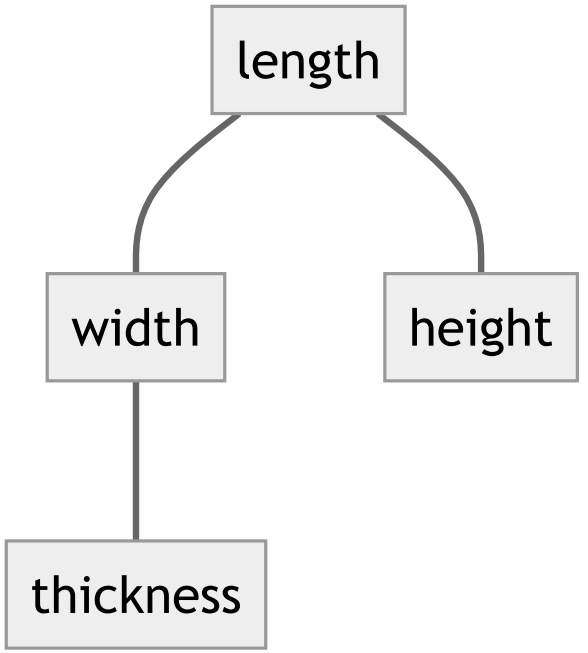
Three levels of quantity conversions

```
static_assert(isq::width != isq::length);
static_assert(isq::width != isq::height);

// width -> length
static_assert(implicitly_convertible(isq::width, isq::length));

// length -> width
static_assert(!implicitly_convertible(isq::length, isq::width));
static_assert(explicitly_convertible(isq::length, isq::width));

// height -> width
static_assert(!implicitly_convertible(isq::height, isq::width));
static_assert(!explicitly_convertible(isq::height, isq::width));
static_assert(castable(isq::height, isq::width));

// time -> length
static_assert(!implicitly_convertible(isq::time, isq::length));
static_assert(!explicitly_convertible(isq::time, isq::length));
static_assert(!castable(isq::time, isq::length));
```



Common quantity specification

```
// width + width -> width
static_assert(common_quantity_spec(isq::width, isq::width) ==
              isq::width);

// thickness + width -> width
static_assert(common_quantity_spec(isq::thickness, isq::width) ==
              isq::width);

// thickness + height -> length
static_assert(common_quantity_spec(isq::thickness, isq::height) ==
              isq::length);
```



Simple user extensions and composability

- Users can always easily **add new quantities** to existing hierarchies

```
inline constexpr struct horizontal_length : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area : quantity_spec<isq::area, horizontal_length * isq::width> {} horizontal_area;
```

Simple user extensions and composability

- Users can always easily **add new quantities** to existing hierarchies

```
inline constexpr struct horizontal_length : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area : quantity_spec<isq::area, horizontal_length * isq::width> {} horizontal_area;
```

```
static_assert(implicitly_convertible(horizontal_length, isq::length));
static_assert(!implicitly_convertible(isq::length, horizontal_length));

static_assert(implicitly_convertible(horizontal_area, isq::area));
static_assert(!implicitly_convertible(isq::area, horizontal_area));
```

Simple user extensions and composability

- Users can always easily **add new quantities** to existing hierarchies

```
inline constexpr struct horizontal_length : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area : quantity_spec<isq::area, horizontal_length * isq::width> {} horizontal_area;
```

```
static_assert(implicitly_convertible(horizontal_length, isq::length));
static_assert(!implicitly_convertible(isq::length, horizontal_length));

static_assert(implicitly_convertible(horizontal_area, isq::area));
static_assert(!implicitly_convertible(isq::area, horizontal_area));
```

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(!implicitly_convertible(isq::length * isq::length, horizontal_area));
```

Simple user extensions and composability

- Users can always easily **add new quantities** to existing hierarchies

```
inline constexpr struct horizontal_length : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area : quantity_spec<isq::area, horizontal_length * isq::width> {} horizontal_area;
```

```
static_assert(implicitly_convertible(horizontal_length, isq::length));
static_assert(!implicitly_convertible(isq::length, horizontal_length));

static_assert(implicitly_convertible(horizontal_area, isq::area));
static_assert(!implicitly_convertible(isq::area, horizontal_area));
```

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(!implicitly_convertible(isq::length * isq::length, horizontal_area));
```

```
static_assert(implicitly_convertible(horizontal_length * isq::width, isq::area));
static_assert(implicitly_convertible(horizontal_length * isq::width, horizontal_area));
```

Typed quantities

```
class Box {
  quantity<horizontal_length[m]> length_;
  quantity<isq::width[m]> width_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    : length_(l), width_(w), height_(h)
  {}

  quantity<horizontal_area[m2]> base() const { return length_ * width_; }
  // ...
};
```

Typed quantities

```
class Box {
  quantity<horizontal_length[m]> length_;
  quantity<isq::width[m]> width_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    : length_(l), width_(w), height_(h)
  {}

  quantity<horizontal_area[m2]> base() const { return length_ * width_; }
  // ...
};
```

```
Box my_box1(2 * m, 3 * m, 1 * m);
Box my_box2(2 * horizontal_length[m], 3 * isq::width[m], 1 * isq::height[m]);
Box my_box3(horizontal_length(2 * m), isq::width(3 * m), isq::height(1 * m));
```

Typed quantities

```
class Box {
  quantity<horizontal_length[m]> length_;
  quantity<isq::width[m]> width_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    : length_(l), width_(w), height_(h)
  {}

  quantity<horizontal_area[m2]> base() const { return length_ * width_; }
  // ...
};
```

```
Box my_box1(2 * m, 3 * m, 1 * m);
Box my_box2(2 * horizontal_length[m], 3 * isq::width[m], 1 * isq::height[m]);
Box my_box3(horizontal_length(2 * m), isq::width(3 * m), isq::height(1 * m));
```

It is up to the user to decide when and where to care about explicit quantity types and when to prefer simple unit-only mode.

Typed quantities

```
class Box {
  quantity<horizontal_length[m]> length_;
  quantity<isq::width[m]> width_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    : length_(l), width_(w), height_(h)
  {}

  quantity<horizontal_area[m2]> base() const { return length_ * width_; }
  // ...
};
```

```
Box my_box(isq::height(1 * m), horizontal_length(2 * m), isq::width(3 * m));
```

```
error: no matching function for call to 'Box::Box(quantity<reference<isq::height, si::metre>(), int>,
                                                  quantity<reference<horizontal_length, si::metre>(), int>,
                                                  quantity<reference<isq::width, si::metre>(), int>)'
   27 | Box my_box(isq::height(1 * m), horizontal_length(2 * m), isq::width(3 * m));
      |                                                                          ^
<source>:19:3: note: candidate: 'Box::Box(quantity<reference<horizontal_length, si::metre>()>,
                                          quantity<reference<isq::width, si::metre>()>,
                                          quantity<reference<isq::height, si::metre>()>)'
   19 |   Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
      |   ^~~
<source>:19:38: note:   no known conversion for argument 1 from 'quantity<reference<isq::height, si::metre>(),int>'
                        to 'quantity<reference<horizontal_length, si::metre>(),double>'
   19 |   Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
      |       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
```

Proper direction matters!

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

- **Hz** (hertz) - unit of frequency
- **Bq** (becquerel) - unit of activity
- **Bd** (baud) - unit of modulation rate

All the above are quantities of a dimension T^{-1}.

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

2 Hz

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

```
using namespace boost::units::si;
std::cout << 1 * becquerel + 1 * hertz << '\n';
```

2 Hz

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

2 Hz

```
using namespace boost::units::si;
std::cout << 1 * becquerel + 1 * hertz << '\n';
```

2 Hz

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

2 Hz

```
using namespace boost::units::si;
std::cout << 1 * becquerel + 1 * hertz << '\n';
```

2 Hz

nholthaus/units

```
using namespace units::literals;
std::cout << 1_Hz + 1_Bq << '\n';
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Boost.Units

```
using namespace boost::units::si;
std::cout << 1 * hertz + 1 * becquerel << '\n';
```

```
2 Hz
```

```
using namespace boost::units::si;
std::cout << 1 * becquerel + 1 * hertz << '\n';
```

```
2 Hz
```

nholthaus/units

```
using namespace units::literals;
std::cout << 1_Hz + 1_Bq << '\n';
```

```
2 s^-1
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Pint

```
print(1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud)
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Pint

```
print(1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud)
```

```
3.0 Hz
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Pint

```
print(1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud)
```

```
3.0 Hz
```

JSR 385

```
System.out.println(Quantities.getQuantity(1, Units.HERTZ)
                              .add(Quantities.getQuantity(1, Units.BECQUEREL)));
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

Pint

```
print(1 * ureg.hertz + 1 * ureg.becquerel + 1 * ureg.baud)
```

```
3.0 Hz
```

JSR 385

```
System.out.println(Quantities.getQuantity(1, Units.HERTZ)
                              .add(Quantities.getQuantity(1, Units.BECQUEREL)));
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The method add(Quantity<Frequency>) in the type ComparableQuantity<Frequency> is not applicable for the arguments
        (ComparableQuantity<Radioactivity>)
```

Quantity kinds (ISO 80000)

- Quantities may be grouped together into **categories** of quantities that are **mutually comparable**
- Mutually comparable quantities are called **quantities of the same kind**

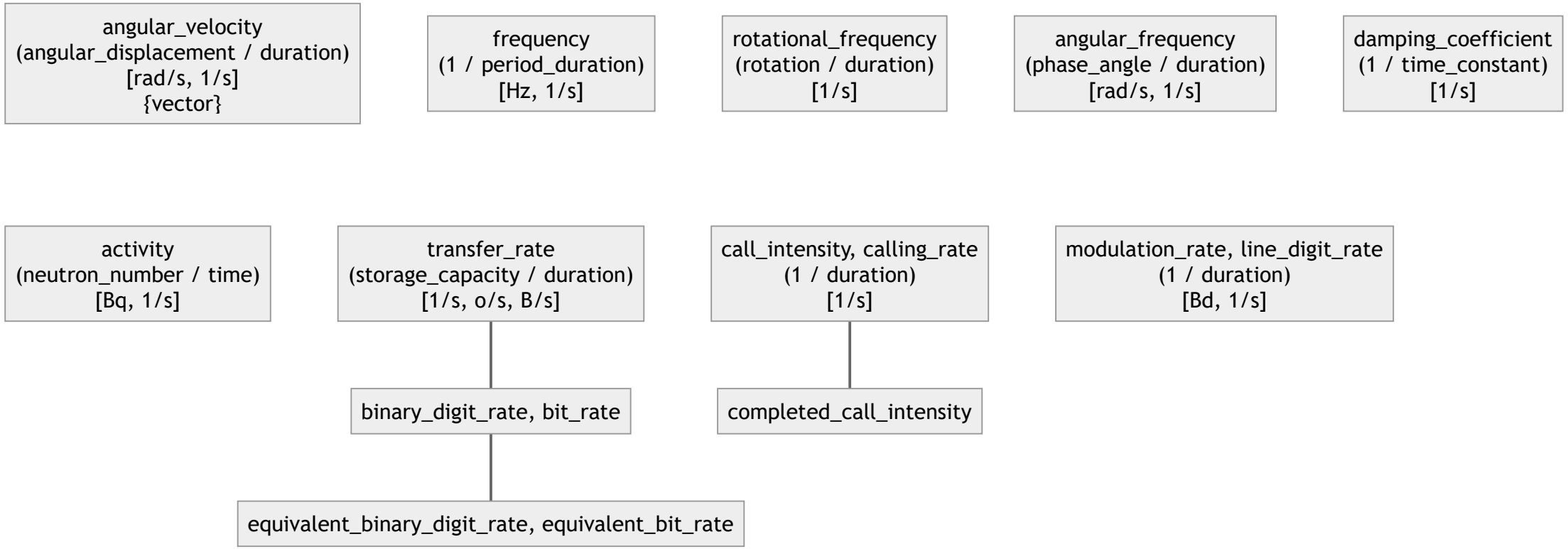
Quantity kinds (ISO 80000)

- Quantities may be grouped together into **categories** of quantities that are **mutually comparable**
- Mutually comparable quantities are called **quantities of the same kind**
- Two or more quantities cannot be **added or subtracted** unless they belong to the same category of mutually comparable quantities

Quantity kinds (ISO 80000)

- Quantities may be grouped together into **categories** of quantities that are **mutually comparable**
- Mutually comparable quantities are called **quantities of the same kind**
- Two or more quantities cannot be **added or subtracted** unless they belong to the same category of mutually comparable quantities
- Quantities of the same kind within a given system of quantities **have the same quantity dimension**
- Quantities of the same dimension are **not necessarily of the same kind**

ISQ quantities of dimension T^{-1}



`kind_of<QS>` modifier

- Denotes a **family of quantities belonging to the same kind**
  - such quantity *represents any quantity from a kind hierarchy tree*

kind_of<QS> modifier

- Denotes a **family of quantities belonging to the same kind**
  - such quantity *represents any quantity from a kind hierarchy tree*
- Can be *obtained through* ***get_kind()***

```
static_assert(get_kind(isq::width) == get_kind(isq::height));
static_assert(get_kind(isq::width) == kind_of<isq::length>);
```

`kind_of<QS>` modifier

- Denotes a **family of quantities belonging to the same kind**
  - such quantity *represents any quantity from a kind hierarchy tree*
- Can be *obtained through* ***`get_kind()`***

```
static_assert(get_kind(isq::width) == get_kind(isq::height));
static_assert(get_kind(isq::width) == kind_of<isq::length>);
```

- Quantity of **`kind_of<QS>`** is *implicitly convertible* to any quantity from its tree

```
static_assert(implicitly_convertible(kind_of<isq::length>, isq::width));
```

`kind_of<QS>` modifier

- Denotes a **family of quantities belonging to the same kind**
  - such quantity *represents any quantity from a kind hierarchy tree*
- Can be *obtained through* ***`get_kind()`***

```
static_assert(get_kind(isq::width) == get_kind(isq::height));
static_assert(get_kind(isq::width) == kind_of<isq::length>);
```

- Quantity of **`kind_of<QS>`** is *implicitly convertible* to any quantity from its tree

```
static_assert(implicitly_convertible(kind_of<isq::length>, isq::width));
```

Quantities of different kinds **can't be compared added or subtracted**.

SI units for ISQ base quantities

```
namespace mp_units::si {

inline constexpr struct second : named_unit<"s", kind_of<isq::time>> {} second;
inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct gram : named_unit<"g", kind_of<isq::mass>> {} gram;
inline constexpr struct kilogram : decltype(kilo<gram>) {} kilogram;
inline constexpr struct ampere : named_unit<"A", kind_of<isq::electric_current>> {} ampere;
inline constexpr struct kelvin : named_unit<"K", kind_of<isq::thermodynamic_temperature>> {} kelvin;
inline constexpr struct mole : named_unit<"mol", kind_of<isq::amount_of_substance>> {} mole;
inline constexpr struct candela : named_unit<"cd", kind_of<isq::luminous_intensity>> {} candela;

}
```

SI units for ISQ base quantities

```
namespace mp_units::si {

inline constexpr struct second : named_unit<"s", kind_of<isq::time>> {} second;
inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct gram : named_unit<"g", kind_of<isq::mass>> {} gram;
inline constexpr struct kilogram : decltype(kilo<gram>) {} kilogram;
inline constexpr struct ampere : named_unit<"A", kind_of<isq::electric_current>> {} ampere;
inline constexpr struct kelvin : named_unit<"K", kind_of<isq::thermodynamic_temperature>> {} kelvin;
inline constexpr struct mole : named_unit<"mol", kind_of<isq::amount_of_substance>> {} mole;
inline constexpr struct candela : named_unit<"cd", kind_of<isq::luminous_intensity>> {} candela;

}
```

All SI units store the information about associated quantity kind.

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian : named_unit<"sr", square(metre) / square(metre),
                                               kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz : named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian : named_unit<"sr", square(metre) / square(metre),
                                               kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz : named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

Derived units can also be explicitly constrained to only work with specific quantity kind.

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

```
error: invalid operands to binary expression ('quantity<hertz{}, std::remove_cvref_t<int>>' (aka 'quantity<si::hertz{}, int>') and
                                               'quantity<becquerel{}, std::remove_cvref_t<int>>' (aka 'quantity<si::becquerel{}, int>'))
   10 | auto res = 1 * Hz + 1 * Bq + 1 * Bd;
      |            ~~~~~~ ^ ~~~~~~
note: candidate template ignored: constraints not satisfied [with R1 = si::hertz{{{{{}}}}}, Rep1 = int, R2 = struct becquerel{{{{{}}}}}, Rep2 = int]
  420 | [[nodiscard]] constexpr Quantity auto operator+(const quantity<R1, Rep1>& lhs, const quantity<R2, Rep2>& rhs)
      |                                       ^
note: because 'detail::CommonlyInvocableQuantities<std::plus<>, quantity<hertz{{{{{}}}}}, int>, quantity<struct becquerel{{{{{}}}}}, int> >' evaluated to false
  419 |   requires detail::CommonlyInvocableQuantities<std::plus<>, quantity<R1, Rep1>, quantity<R2, Rep2>>
      |            ^
note: because 'common_reference(Q1::reference, Q2::reference)' would be invalid: no matching function for call to 'common_reference'
   74 |   requires { common_reference(Q1::reference, Q2::reference); } &&
      |              ^
```

What should be the result of the following equation?

```
auto res = 1 * Hz + 1 * Bq + 1 * Bd;
```

The same rules apply to:

- `1 * rad + 1 * sr + 1 * bit`
- `1 * Gy + 1 * Sv`
- …

The affine space abstractions

The affine space has two types of entities:

- **point** - a position specified with coordinate values
- **vector** - the difference between two points

The affine space abstractions

The affine space has two types of entities:

- **point** - a position specified with coordinate values
- **vector** - the difference between two points

- One can do a **limited set of operations** in affine space on points and vectors
- **Prevents quantity equations that do not have physical sense**

The affine space abstractions

The affine space has two types of entities:

- **point** - a position specified with coordinate values
- **vector** - the difference between two points

- One can do a limited set of operations in affine space on points and vectors
- Prevents quantity equations that do not have physical sense

```
quantity_point temp1(21.2 * deg_C);
quantity_point temp2(21.4 * deg_C);
auto res = temp1 + temp2;  // Compile-time error
```

Invalid affine space operations

Invalid affine space operations

- Adding two **`quantity_point`** objects

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar
- Multiplying/dividing a **`quantity_point`** with a **`quantity`**

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar
- Multiplying/dividing a **`quantity_point`** with a **`quantity`**
- Multiplying/dividing two **`quantity_point`** objects

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar
- Multiplying/dividing a **`quantity_point`** with a **`quantity`**
- Multiplying/dividing two **`quantity_point`** objects
- Mixing **`quantity_points`** of *different quantity kinds*

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar
- Multiplying/dividing a **`quantity_point`** with a **`quantity`**
- Multiplying/dividing two **`quantity_point`** objects
- Mixing **`quantity_points`** of *different quantity kinds*
- Mixing **`quantity_points`** of *inconvertible* quantities

Invalid affine space operations

- Adding two **`quantity_point`** objects
- Subtracting a **`quantity_point`** from a **`quantity`**
- Multiplying/dividing a **`quantity_point`** with a scalar
- Multiplying/dividing a **`quantity_point`** with a **`quantity`**
- Multiplying/dividing two **`quantity_point`** objects
- Mixing **`quantity_points`** of *different quantity kinds*
- Mixing **`quantity_points`** of *inconvertible* quantities
- Mixing **`quantity_points`** of convertible quantities but *with unrelated origins*

Fixing The Guardian article

```
namespace si {

inline constexpr struct absolute_zero :
    absolute_point_origin<absolute_zero, isq::thermodynamic_temperature> {} absolute_zero;
inline constexpr struct zeroth_kelvin : decltype(absolute_zero) {} zeroth_kelvin;

inline constexpr struct kelvin : named_unit<"K", kind_of<isq::thermodynamic_temperature>, zeroth_kelvin> {} kelvin;

inline constexpr struct ice_point : relative_point_origin<quantity_point{273'150 * milli<kelvin>}> {} ice_point;
inline constexpr struct zeroth_degree_Celsius : decltype(ice_point) {} zeroth_degree_Celsius;

inline constexpr struct degree_Celsius : named_unit<{"°C", "'C"}, kelvin, zeroth_degree_Celsius> {} degree_Celsius;

}
```

Fixing The Guardian article

```
namespace si {

inline constexpr struct absolute_zero :
    absolute_point_origin<absolute_zero, isq::thermodynamic_temperature> {} absolute_zero;
inline constexpr struct zeroth_kelvin : decltype(absolute_zero) {} zeroth_kelvin;

inline constexpr struct kelvin : named_unit<"K", kind_of<isq::thermodynamic_temperature>, zeroth_kelvin> {} kelvin;

inline constexpr struct ice_point : relative_point_origin<quantity_point{273'150 * milli<kelvin>}> {} ice_point;
inline constexpr struct zeroth_degree_Celsius : decltype(ice_point) {} zeroth_degree_Celsius;

inline constexpr struct degree_Celsius : named_unit<{"°C", "'C"}, kelvin, zeroth_degree_Celsius> {} degree_Celsius;

}
```

```
namespace usc {

inline constexpr struct zeroth_degree_Fahrenheit :
    relative_point_origin<si::zeroth_degree_Celsius -
                          32 * (mag<ratio{5, 9}> * si::degree_Celsius)> {} zeroth_degree_Fahrenheit;

inline constexpr struct degree_Fahrenheit :
    named_unit<{"°F", "'F"}, mag<ratio{5, 9}> * si::degree_Celsius, zeroth_degree_Fahrenheit> {} degree_Fahrenheit;

}
```

Fixing The Guardian article

```
const quantity_point max{43. * deg_C};
const quantity_point avg{25. * deg_C};
const quantity delta = max - avg;

std::println("EU: Malawi swelters with temperatures nearly {} above average",
             delta);
std::println("US: Malawi swelters with temperatures nearly {} above average",
             delta.in(deg_F));
std::println("");
std::println("By Saturday, parts of Malawi saw a maximum temperature of {} ({}),",
             max.quantity_from_zero(), max.in(deg_F).quantity_from_zero());
std::println("compared with an average of nearly {} ({}) for the time of year.",
             avg.quantity_from_zero(), avg.in(deg_F).quantity_from_zero());
```

Fixing The Guardian article

```
const quantity_point max{43. * deg_C};
const quantity_point avg{25. * deg_C};
const quantity delta = max - avg;

std::println("EU: Malawi swelters with temperatures nearly {} above average",
             delta);
std::println("US: Malawi swelters with temperatures nearly {} above average",
             delta.in(deg_F));
std::println("");
std::println("By Saturday, parts of Malawi saw a maximum temperature of {} ({}),",
             max.quantity_from_zero(), max.in(deg_F).quantity_from_zero());
std::println("compared with an average of nearly {} ({}) for the time of year.",
             avg.quantity_from_zero(), avg.in(deg_F).quantity_from_zero());
```

```
EU: Malawi swelters with temperatures nearly 18 °C above average
US: Malawi swelters with temperatures nearly 32.4 °F above average

By Saturday, parts of Malawi saw a maximum temperature of 43 °C (109.4 °F),
compared with an average of nearly 25 °C (77 °F) for the time of year.
```

Points are more common than most of us imagine

Points are everywhere around us and should become more popular in the products we implement.

Points are more common than most of us imagine

Points are everywhere around us and should become more popular in the products we implement.

- Temperature points
- Timestamps
- Daily mass readouts from the scale
- Altitudes of mountain peaks on a map
- Current speed displayed on a car's speed-o-meter
- Today's price of instruments on the market
- ...

Points are more common than most of us imagine

Points are everywhere around us and should become more popular in the products we implement.

Improving the affine space's points intuition will allow us to write better and safer software.

The Hochrheinbrücke bridge case

The Hochrheinbrücke bridge case

- The German uses the North Sea as a reference for sea level
  - reference point is known as an Amsterdam Sea Level

The Hochrheinbrücke bridge case

- The German uses the North Sea as a reference for sea level
  - reference point is known as an Amsterdam Sea Level
- The Swiss uses the Mediterranean Sea reference for sea level

The Hochrheinbrücke bridge case

- The German uses the North Sea as a reference for sea level
  - reference point is known as an Amsterdam Sea Level
- The Swiss uses the Mediterranean Sea reference for sea level
- The reference altitude used in Switzerland is 270 millimeters lower than in Germany

The Hochrheinbrücke bridge case

- The German uses the North Sea as a reference for sea level
  - reference point is known as an Amsterdam Sea Level
- The Swiss uses the Mediterranean Sea reference for sea level
- The reference altitude used in Switzerland is 270 millimeters lower than in Germany

> “The difference of **27 cm** was certainly known, and everything had been drawn up correctly on paper” explained Beat von Arx, project manager at the department of Civil Works of the Swiss canton Aargau.

The Hochrheinbrücke bridge case

- Even during the **early construction phase**, it became clear that **something was wrong**
- At Christmas 2003, the bridge builders received an unpleasant gift
  - a clear *difference in height* was clearly evident on both sides and *more than **27** centimeters*

The Hochrheinbrücke bridge case

- Even during the **early construction phase**, it became clear that **something was wrong**
- At Christmas 2003, the bridge builders received an unpleasant gift
  - a clear *difference in height* was clearly evident on both sides and *more than **27** centimeters*
- **A sign error was hidden in the height calculations of the responsible engineering office on the Swiss side**

The Hochrheinbrücke bridge case

- Even during the **early construction phase**, it became clear that **something was wrong**
- At Christmas 2003, the bridge builders received an unpleasant gift
  - a clear *difference in height* was clearly evident on both sides and *more than **27** centimeters*
- **A sign error was hidden in the height calculations of the responsible engineering office on the Swiss side**

Instead of 27 cm higher the Swiss built 27 cm lower which doubled the difference to 54 cm.

The Hochrheinbrücke bridge case

```
// reference points for both systems
constexpr struct amsterdam_sea_level : absolute_point_origin<amsterdam_sea_level, isq::altitude> {} amsterdam_sea_level;
constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level - 27*cm> {} mediterranean_sea_level;
```

The Hochrheinbrücke bridge case

```
// reference points for both systems
constexpr struct amsterdam_sea_level : absolute_point_origin<amsterdam_sea_level, isq::altitude> {} amsterdam_sea_level;
constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level - 27*cm> {} mediterranean_sea_level;
```

```
// altitude quantity point types expressing both systems
using altitude_DE = quantity_point<isq::altitude[m], amsterdam_sea_level>;
using altitude_CH = quantity_point<isq::altitude[m], mediterranean_sea_level>;
```

The Hochrheinbrücke bridge case

```
// reference points for both systems
constexpr struct amsterdam_sea_level : absolute_point_origin<amsterdam_sea_level, isq::altitude> {} amsterdam_sea_level;
constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level - 27*cm> {} mediterranean_sea_level;
```

```
// altitude quantity point types expressing both systems
using altitude_DE = quantity_point<isq::altitude[m], amsterdam_sea_level>;
using altitude_CH = quantity_point<isq::altitude[m], mediterranean_sea_level>;
```

```
// expected bridge altitude in a specific reference system
quantity_point expected_bridge_alt = amsterdam_sea_level + 330 * m;
```

The Hochrheinbrücke bridge case

```
// reference points for both systems
constexpr struct amsterdam_sea_level : absolute_point_origin<amsterdam_sea_level, isq::altitude> {} amsterdam_sea_level;
constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level - 27*cm> {} mediterranean_sea_level;
```

```
// altitude quantity point types expressing both systems
using altitude_DE = quantity_point<isq::altitude[m], amsterdam_sea_level>;
using altitude_CH = quantity_point<isq::altitude[m], mediterranean_sea_level>;
```

```
// expected bridge altitude in a specific reference system
quantity_point expected_bridge_alt = amsterdam_sea_level + 330 * m;
```

```
// some nearest landmark altitudes on both sides of the river
// equal but not equal ;-)
altitude_DE landmark_alt_DE = altitude_DE::point_origin + 300 * m;
altitude_CH landmark_alt_CH = altitude_CH::point_origin + 300 * m;

// artificial deltas from landmarks of the bridge base on both sides of the river
quantity delta_DE = isq::height(3 * m);
quantity delta_CH = isq::height(-2 * m);

// artificial altitude of the bridge base on both sides of the river
quantity_point bridge_base_alt_DE = landmark_alt_DE + delta_DE;
quantity_point bridge_base_alt_CH = landmark_alt_CH + delta_CH;
```

The Hochrheinbrücke bridge case

```
// artificial height of the required bridge pilar height on both sides of the river
quantity bridge_pilar_height_DE = expected_bridge_alt - bridge_base_alt_DE;
quantity bridge_pilar_height_CH = expected_bridge_alt - bridge_base_alt_CH;
```

The Hochrheinbrücke bridge case

```
// artificial height of the required bridge pilar height on both sides of the river
quantity bridge_pilar_height_DE = expected_bridge_alt - bridge_base_alt_DE;
quantity bridge_pilar_height_CH = expected_bridge_alt - bridge_base_alt_CH;
```

```
std::println("Bridge pillars height:");
std::println("- Germany:     {}", bridge_pilar_height_DE);
std::println("- Switzerland: {}", bridge_pilar_height_CH);

// artificial bridge altitude on both sides of the river in both systems
quantity_point bridge_road_alt_DE = bridge_base_alt_DE + bridge_pilar_height_DE;
quantity_point bridge_road_alt_CH = bridge_base_alt_CH + bridge_pilar_height_CH;

std::println("Bridge road altitude:");
std::println("- Germany:     {}", bridge_road_alt_DE);
std::println("- Switzerland: {}", bridge_road_alt_CH);

std::println("Bridge road altitude relative to the Amsterdam Sea Level:");
std::println("- Germany:     {}",
             bridge_road_alt_DE.quantity_from(amsterdam_sea_level));
std::println("- Switzerland: {}",
             bridge_road_alt_CH.quantity_from(amsterdam_sea_level));
```

```
Bridge pilars height:
- Germany:     27 m
- Switzerland: 3227 cm
Bridge road altitude:
- Germany:     330 m AMSL(DE)
- Switzerland: 33027 cm AMSL(CH)
Absolute bridge road altitude:
- Germany:     330 m
- Switzerland: 33000 cm
```

SAFETY PITFALLS

What would you expect as a result?

```
quantity res = 5 * h / (120 * min);
```

Integer division

```
static_assert(120 * km / (2 * h) == 60 * km / h);
```

Integer division

```
static_assert(120 * km / (2 * h) == 60 * km / h);
```

```
static_assert(5 * km / (24 * h) == 0 * km/h);
```

Integer division

```
static_assert(120 * km / (2 * h) == 60 * km / h);
```

```
static_assert(5 * km / (24 * h) == 0 * km/h);
```

```
static_assert(5 * h / (120 * min) == 0 * one);
```

Integer division

```
static_assert(120 * km / (2 * h) == 60 * km / h);
```

```
static_assert(5 * km / (24 * h) == 0 * km/h);
```

```
static_assert(5 * h / (120 * min) == 0 * one);
```

- Modulo works OK

```
static_assert(5 * h % (120 * min) == 1 * h);
```

Lack of safe numeric types

Lack of safe numeric types

- *Integers can overflow* on arithmetics
  - this has already caused some expensive failures in engineering (Ariane flight V88)

Lack of safe numeric types

- *Integers can overflow* on arithmetics
  - this has already caused some expensive failures in engineering (Ariane flight V88)
- *Integers can be truncated during assignment* to a narrower type

Lack of safe numeric types

- *Integers can overflow* on arithmetics
  - this has already caused some expensive failures in engineering (Ariane flight V88)
- *Integers can be truncated during assignment* to a narrower type
- *Floating-point types may lose precision during assignment* to a narrower type

Lack of safe numeric types

- *Integers can overflow* on arithmetics
  - this has already caused some expensive failures in engineering (Ariane flight V88)
- *Integers can be truncated during assignment* to a narrower type
- *Floating-point types may lose precision during assignment* to a narrower type
- Conversion from ***`std::int64_t`*** *to* ***`double`*** *may also lose precision*

Lack of safe numeric types

If we had *safe numeric types* in the C++ standard library, they could easily be used as a quantity representation type in the quantities and units library, which would address these safety concerns.

Lack of safe numeric types

If we had *safe numeric types* in the C++ standard library, they could easily be used as a quantity representation type in the quantities and units library, which would address these safety concerns.

Also, having a *type trait* informing if a *conversion* from one type to another is *value-preserving* would help to address some of the issues here.

Potential surprises during units composition

```
quantity q = 60 * km / 2 * h;
```

Potential surprises during units composition

```
quantity q = 60 * km / 2 * h;
```

```
QuantityOf<isq::speed> auto q = 60 * km / 2 * h;
```

Potential surprises during units composition

```
quantity q = 60 * km / 2 * h;
```

```
QuantityOf<isq::speed> auto q = 60 * km / 2 * h;
```

```
error: deduced type 'quantity<derived_unit<hour, kilo_<metre>>{}, int>'
       (aka 'quantity<derived_unit<non_si::hour, si::kilo_<si::metre>>{}, int>') does not satisfy 'QuantityOf<isq::speed>'
   51 | QuantityOf<isq::speed> auto q = 60 * km / 2 * h;
      | ^~~~~~~~~~~~~~~~~~~~~~~~~~
note: because 'QuantitySpecOf<std::remove_const_t<decltype(quantity<derived_unit<hour, kilo_<metre> >{{{}}}, int>::quantity_spec)>,
                              struct speed{{{}}}>' evaluated to false
   61 | concept QuantityOf = Quantity<Q> && QuantitySpecOf<std::remove_const_t<decltype(Q::quantity_spec)>, QS>;
      |                                     ^
note: because 'implicitly_convertible(kind_of_<derived_quantity_spec<isq::length, isq::time>>{},
                                       struct speed{{{}}})' evaluated to false
  147 |   QuantitySpec<T> && QuantitySpec<std::remove_const_t<decltype(QS)>> && implicitly_convertible(T{}, QS) &&
      |
```

Potential surprises during units composition

```
quantity q = 60 * km / 2 * h;
```

```
QuantityOf<isq::speed> auto q = 60 * km / (2 * h);
```

Potential surprises during units composition

```
quantity q = 60 * km / 2 * h;
```

```
QuantityOf<isq::speed> auto q = 60 * km / (2 * h);
```

In order to support **`60 * km / h`** without additional parentheses, the above compiles fine with no errors.

Potential surprises during units composition

```
template<typename T>
auto make_length(T v) { return v * si::metre; }
```

Potential surprises during units composition

```
template<typename T>
auto make_length(T v) { return v * si::metre; }
```

```
auto v = 42;
quantity q = make_length(v);
```

Potential surprises during units composition

```
template<typename T>
auto make_length(T v) { return v * si::metre; }
```

```
auto v = 42;
quantity q = make_length(v);
```

```
auto v = 42 * m;
quantity q = make_length(v);
```

Potential surprises during units composition

The previous issues could be immediately changed to the compilation errors with the cost of having to type `60 * (km / h)`.

Potential surprises during units composition

The previous issues could be immediately changed to the compilation errors with the cost of having to type `60 * (km / h)`.

mp-units initially produced such errors, but users were confused why `60 * km / h` does not work.

Potential surprises during units composition

The problems with units compositions will *always surface with a compile-time error* at some point in the user's code when they *assign* the resulting quantity *to one with an explicitly provided quantity type*.

No more surprises during units composition

```
quantity<km / h, int> q1 = 60 * km / 2 * h;                 // Compile-time error
quantity<isq::speed[km / h], int> q2 = 60 * km / 2 * h;     // Compile-time error
QuantityOf<isq::speed> auto q3 = 60 * km / 2 * h;           // Compile-time error
```

No more surprises during units composition

```
template<typename T>
auto make_length(T v) { return v * si::metre; }
```

```
auto v = 42 * m;
quantity<si::metre, int> q1 = make_length(v);           // Compile-time error
quantity<isq::length[si::metre]> q2 = make_length(v);   // Compile-time error
QuantityOf<isq::length> q3 = make_length(v);            // Compile-time error
```

No more surprises during units composition

```
template<typename T>
QuantityOf<isq::length> auto make_length(T v) { return v * si::metre; } // Compile-time error
```

```
auto v = 42 * m;
quantity q = make_length(v);
```

No more surprises during units composition

```
template<Representation T>
auto make_length(T v) { return v * si::metre; }
```

```
auto v = 42 * m;
quantity q = make_length(v);  // Compile-time error
```

Limitations of systems of quantities

ALLOWING IRRATIONAL QUANTITY COMBINATIONS

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(implicitly_convertible(isq::area, isq::length * isq::length));
```

Limitations of systems of quantities

ALLOWING IRRATIONAL QUANTITY COMBINATIONS

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(implicitly_convertible(isq::area, isq::length * isq::length));
```

```
static_assert(implicitly_convertible(isq::width * isq::height, isq::area));
static_assert(!implicitly_convertible(isq::area, isq::width * isq::height));
static_assert(explicitly_convertible(isq::area, isq::width * isq::height));
```

Limitations of systems of quantities

ALLOWING IRRATIONAL QUANTITY COMBINATIONS

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(implicitly_convertible(isq::area, isq::length * isq::length));
```

```
static_assert(implicitly_convertible(isq::width * isq::height, isq::area));
static_assert(!implicitly_convertible(isq::area, isq::width * isq::height));
static_assert(explicitly_convertible(isq::area, isq::width * isq::height));
```

```
static_assert(implicitly_convertible(isq::height * isq::height, isq::area));
static_assert(!implicitly_convertible(isq::area, isq::height * isq::height));
static_assert(explicitly_convertible(isq::area, isq::height * isq::height));
```

Limitations of systems of quantities

ALLOWING IRRATIONAL QUANTITY COMBINATIONS

```
static_assert(implicitly_convertible(isq::length * isq::length, isq::area));
static_assert(implicitly_convertible(isq::area, isq::length * isq::length));
```

```
static_assert(implicitly_convertible(isq::width * isq::height, isq::area));
static_assert(!implicitly_convertible(isq::area, isq::width * isq::height));
static_assert(explicitly_convertible(isq::area, isq::width * isq::height));
```

```
static_assert(implicitly_convertible(isq::height * isq::height, isq::area));
static_assert(!implicitly_convertible(isq::area, isq::height * isq::height));
static_assert(explicitly_convertible(isq::area, isq::height * isq::height));
```

For humans, it is hard to imagine how two heights form an area, but the library's logic has no way to prevent such operations.

Limitations of systems of quantities

QUANTITIES OF DIMENSION ONE

```
quantity q1 = 1 * m / (10 * m) + 1 * us / (1 * h);
quantity q2 = isq::length(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
quantity q3 = isq::height(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
```

Limitations of systems of quantities

QUANTITIES OF DIMENSION ONE

```
quantity q1 = 1 * m / (10 * m) + 1 * us / (1 * h);
quantity q2 = isq::length(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
quantity q3 = isq::height(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
```

- If we divide two quantities of the same kind, **we end up with a quantity of dimension one**
  - we can divide two *lengths* to get a *slope of the ramp* or two *durations* to get the *clock accuracy*

Limitations of systems of quantities

QUANTITIES OF DIMENSION ONE

```
quantity q1 = 1 * m / (10 * m) + 1 * us / (1 * h);
quantity q2 = isq::length(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
quantity q3 = isq::height(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
```

- If we divide two quantities of the same kind, **we end up with a quantity of dimension one**
  - we can divide two *lengths* to get a *slope of the ramp* or two *durations* to get the *clock accuracy*

Such ratios mean something fundamentally different, but from the dimensional analysis standpoint, they are mutually comparable.

Limitations of systems of quantities

QUANTITIES OF DIMENSION ONE

```
quantity q1 = 1 * m / (10 * m) + 1 * us / (1 * h);
quantity q2 = isq::length(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
quantity q3 = isq::height(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
```

```
quantity<si::metre / si::metre> ok1 = q1;
quantity<(isq::length / isq::length)[m / m]> ok2 = q1;
quantity<(isq::height / isq::length)[m / m]> ok3 = q1;
```

Limitations of systems of quantities

QUANTITIES OF DIMENSION ONE

```
quantity q1 = 1 * m / (10 * m) + 1 * us / (1 * h);
quantity q2 = isq::length(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
quantity q3 = isq::height(1 * m) / isq::length(10 * m) + isq::time(1 * us) / isq::time(1 * h);
```

```
quantity<si::metre / si::metre> ok1 = q1;
quantity<(isq::length / isq::length)[m / m]> ok2 = q1;
quantity<(isq::height / isq::length)[m / m]> ok3 = q1;
```

```
quantity<si::metre / si::metre> ok4 = q2;
quantity<(isq::length / isq::length)[m / m]> ok5 = q2;
quantity<(isq::height / isq::length)[m / m]> bad1 = q2;  // Compile-time error

quantity<si::metre / si::metre> ok6 = q3;
quantity<(isq::length / isq::length)[m / m]> ok7 = q3;
quantity<(isq::height / isq::length)[m / m]> bad2 = q3;  // Compile-time error
```

Structural types

The **`quantity`** and **`quantity_point`** class templates are structural types to allow them to be passed as template arguments.

Structural types

The **`quantity`** and **`quantity_point`** class templates are structural types to allow them to be passed as template arguments.

```
constexpr struct amsterdam_sea_level : absolute_point_origin<isq::altitude> {
} amsterdam_sea_level;

constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level + -27 * cm> {
} mediterranean_sea_level;

using altitude_DE = quantity_point<isq::altitude[m], amsterdam_sea_level>;
using altitude_CH = quantity_point<isq::altitude[m], mediterranean_sea_level>;
```

Structural types

The **quantity** and **quantity_point** class templates are structural types to allow them to be passed as template arguments.

```
constexpr struct amsterdam_sea_level : absolute_point_origin<isq::altitude> {
} amsterdam_sea_level;

constexpr struct mediterranean_sea_level : relative_point_origin<amsterdam_sea_level + -27 * cm> {
} mediterranean_sea_level;

using altitude_DE = quantity_point<isq::altitude[m], amsterdam_sea_level>;
using altitude_CH = quantity_point<isq::altitude[m], mediterranean_sea_level>;
```

Current language rules require that all member data of a structural type are public :-(

SUMMARY

mp-units is about safety

- Automated but safe unit conversions
- Preventing truncation of data
- **`explicit`** is not explicit enough
- Safe quantity numerical value getters
- Quantities of the same dimension but different kinds
- Various quantities of the same kind
- The affine space abstractions

and more...

- **Composable units**
- Consistent *NTTP usage* improves readability and usability
- *Expression templates* improve readability of types
- One **`named_unit`** and **`quantity_spec`** to rule them all
- Much **terser** systems definitions
- *Pure dimensionless analysis*
- Robust *quantity creation* helpers
- Support for *all ISQ quantities*
- Faster than lightspeed *constants*
- Powerful *text formatting*
- Much more ...

The fun of working with AI

The fun of working with AI

Your personal and company data are protected in this chat

Here is a sketch of the Hochrheinbrücke bridge during construction. The bridge connects Germany with Switzerland and was started from both sides concurrently but the altitude of both sides does not match in the center. I hope you like it! 🎨

2 of 30 responses


CAUTION
Programming
is addictive
(and too much fun)