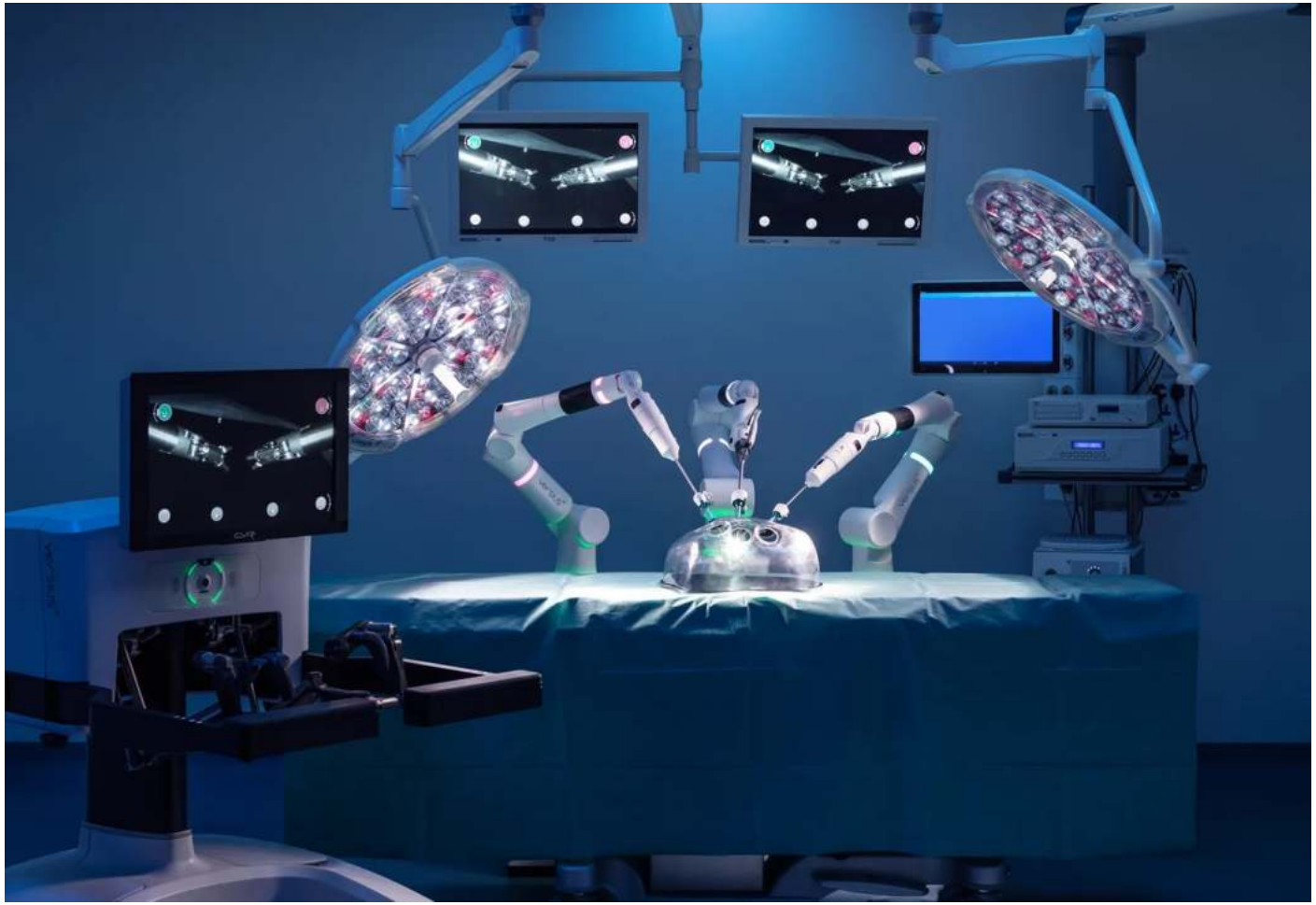
Improving Our Safety With a Quantities and Units Library

MATEUSZ PUSZ

The future is here

The future is here



Me 10 years ago

Tactical Flight Computer

Tactical Flight Computer

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)
- ...

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

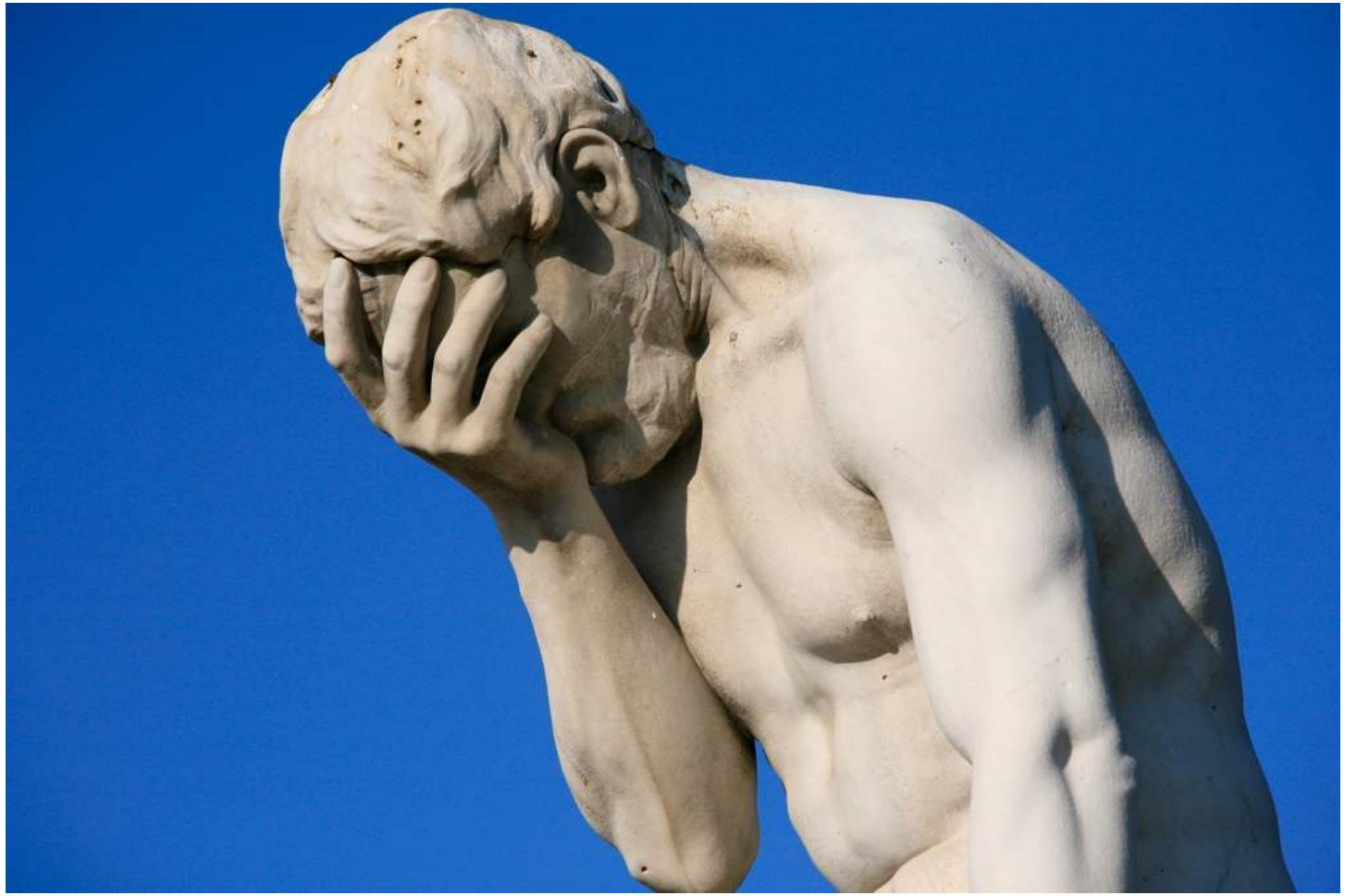
- Often more than once in the repository

```
#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	194
`std::format`	20	13	17	None	194
C++ modules	20	None	17	None	None
`import std;`	23	None	18	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	194
`std::format`	20	13	17	None	194
C++ modules	20	None	17	None	None
`import std;`	23	None	18	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

Help us improve MSVC C++20 conformance

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: Fast first-factor finding function
- P2509: A proposal for a type trait to detect value-preserving conversions

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: Fast first-factor finding function
- P2509: A proposal for a type trait to detect value-preserving conversions

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

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]}", time_to_goal_s);
```

```
TTG: 2.68432 s
```

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]}", time_to_goal_s);
```

```
TTG: 2.68432 s
```

- No need to track units in the identifiers
- The library knows all the conversion factors
  - no magic numbers
  - intermediate results not needed

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]}", 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]}", 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]}", time_to_goal_s);
```

```
error: no matching member function for call to 'in'
   16 | const quantity time_to_goal = (distance * speed).in(s);
      |                               ~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: constraints not satisfied [with ToU = struct second]
  197 |   [[nodiscard]] constexpr QuantityOf<quantity_spec> auto in(ToU) const
      |                                                           ^
note: because 'detail::UnitCompatibleWith<si::second, unit, quantity_spec>' evaluated to false
  195 |   template<detail::UnitCompatibleWith<unit, quantity_spec> ToU>
      |            ^
note: because '!AssociatedUnit<si::second>' evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |    ^
note: and 'UnitOf<si::second, kind_of_<decltype(`
          derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>{{{}}}>' evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |                           ^
note: because 'detail::QuantitySpecConvertibleTo<get_quantity_spec(si::second{}),
      kind_of_<decltype(derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>{{{}}}>'
      evaluated to false
  186 |                 detail::QuantitySpecConvertibleTo<get_quantity_spec(U{}), QS> &&
      |                 ^
note: because 'implicitly_convertible(kind_of_<decltype(struct time{{{}}})>{{{}, {}}},
      kind_of_<decltype(derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>{{{}}})'
      evaluated to false
  146 |   implicitly_convertible(From, To);
      |   ^
```

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]}", time_to_goal_s);
```

```
error: no matching member function for call to 'in'
   16 | const quantity time_to_goal = (distance * speed).in(s);
      |                               ~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: constraints not satisfied [with ToU = struct second]
  197 |   [[nodiscard]] constexpr QuantityOf<quantity_spec> auto in(ToU) const
      |                                                           ^
note: because 'detail::UnitCompatibleWith<si::second, unit, quantity_spec>' evaluated to false
  195 |   template<detail::UnitCompatibleWith<unit, quantity_spec> ToU>
      |            ^
note: because '!AssociatedUnit<si::second>' evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |    ^
note: and 'UnitOf<si::second, kind_of_<decltype(`
          derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>)>{{{}}}>' evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |                           ^
note: because 'detail::QuantitySpecConvertibleTo<get_quantity_spec(si::second{}),
      kind_of_<decltype(derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>)>{{{}}}>'
      evaluated to false
  186 |                 detail::QuantitySpecConvertibleTo<get_quantity_spec(U{}), QS> &&
      |                 ^
note: because 'implicitly_convertible(kind_of_<decltype(struct time{{{}}})>{{{}, {}}},
      kind_of_<decltype(derived_quantity_spec<power<length, 2>, per<time> >{{{}, {{}}}}>)>{{{}}})'
      evaluated to false
  146 |   implicitly_convertible(From, To);
      |   ^
```

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]}", time_to_goal_s);
```

```
error: no matching member function for call to 'in'
   16 | const quantity time_to_goal = (distance * speed).in(s);
      |                                ~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: constraints not satisfied [with ToU = struct second]
  197 |   [[nodiscard]] constexpr QuantityOf<quantity_spec> auto in(ToU) const
      |                                                           ^
note: because 'detail::UnitCompatibleWith<si::second, unit, quantity_spec>' evaluated to false
  195 |   template<detail::UnitCompatibleWith<unit, quantity_spec> ToU>
      |            ^
note: because '!AssociatedUnit<si::second>' evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |    ^
note: and 'UnitOf<si::second, kind_of_<derived_quantity_spec<power<length, 2>, per<time> >>{}>'
      evaluated to false
  209 |   (!AssociatedUnit<U> || UnitOf<U, QS>)&&detail::UnitConvertibleTo<FromU, U{}>;
      |                           ^
note: because 'detail::QuantitySpecConvertibleTo<get_quantity_spec(si::second{}),
      kind_of_<derived_quantity_spec<power<length, 2>, per<time> >>{}>'
      evaluated to false
  186 |                 detail::QuantitySpecConvertibleTo<get_quantity_spec(U{}), QS> &&
      |                 ^
note: because 'implicitly_convertible(kind_of_<struct time>{},
      kind_of_<derived_quantity_spec<power<length, 2>, per<time> >>{})' evaluated to false
  146 |   implicitly_convertible(From, To);
      |   ^
```

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]}",
             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]}",
             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]}",
             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]}",
             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]}",
             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]}",
             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]}",
             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]}",
             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]}",
             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]}",
      |                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]}",
             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]}",
             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]}",
             time_to_goal(distance_m, speed));
```

```
In function 'quantity<si::second()> time_to_goal(quantity<si::metre()>,
                                  quantity<derived_unit<si::kilo_<si::metre>, per<non_si::hour> >()>)':
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;
}
```

Generic interfaces backed up with concepts allow us to perform the most efficient calculations without the need to rescale the units back and forth all the time while still being type-safe.

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]}", 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]}", 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]}", 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]}", 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>]
   10 | 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
   11 |                                         QuantityOf<isq::speed> auto speed)
      |                                         ^
note: because 'QuantitySpecOf<std::remove_const_t<decltype(quantity<derived_unit<second, per<kilo_<metre> > >{}, double>::quantity_spec)>, struct speed{}>' evaluated to false
   65 | concept QuantityOf = Quantity<Q> && QuantitySpecOf<std::remove_const_t<decltype(Q::quantity_spec)>, QS>;
      |                      ^
note: because 'detail::QuantitySpecConvertibleTo<kind_of_<derived_quantity_spec<isq::time, per<isq::length> > >{}, struct speed{}>' evaluated to false
  161 |   QuantitySpec<T> && QuantitySpec<decltype(QS)> && detail::QuantitySpecConvertibleTo<T{}, QS> &&
      |                                                                                     ^
note: because 'implicitly_convertible(kind_of_<derived_quantity_spec<isq::time, per<isq::length> > >{}, struct speed{})' evaluated to false
  146 |   implicitly_convertible(From, To);
      |   ^
1 error generated.
```

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

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

Unit definitions

```
namespace si {

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









}
```

Unit definitions

```
namespace si {

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

inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second final : 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> constexpr kilo_<decltype(U)> kilo;

inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second final : 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> constexpr kilo_<decltype(U)> kilo;

inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct second final : 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 provided in an isolated namespace and are opt-in as they may easily collide with the user's identifiers.

Unit definitions

```
namespace non_si {

inline constexpr struct minute final : named_unit<"min", mag<60> * si::second> {} minute;
inline constexpr struct hour final : 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 final : named_unit<"min", mag<60> * si::second> {} minute;
inline constexpr struct hour final : 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 final : named_unit<"yd", mag_ratio<9'144, 10'000> * si::metre> {} yard;
inline constexpr struct mile final : named_unit<"mi", mag<1760> * yard> {} mile;

namespace unit_symbols {

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

}
}
```

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 << q1.in<double>(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.

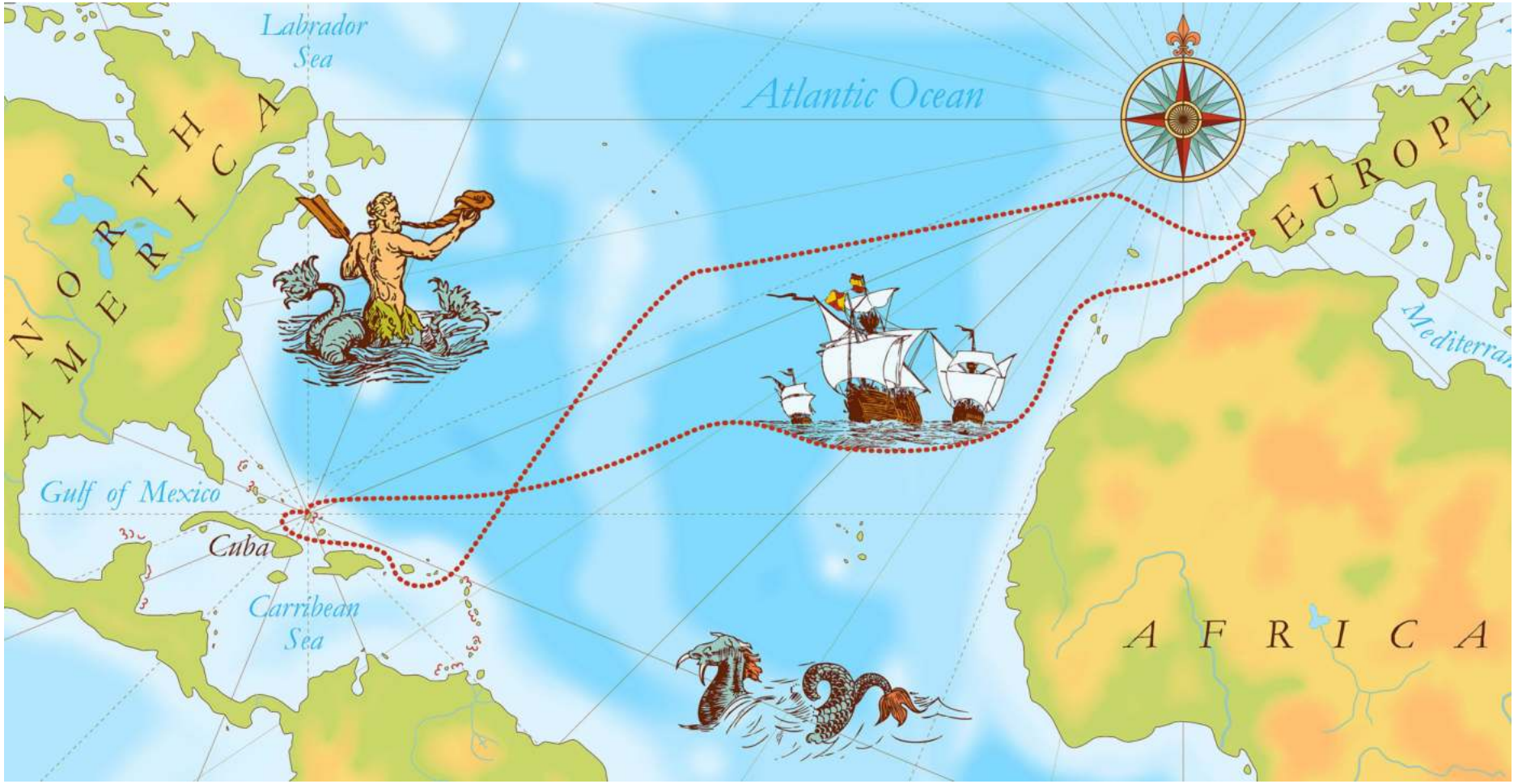
Preventing truncation of data

```
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?

```cpp
// 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 final : named_unit<"deg", mag_ratio<5667, 100> * Mile> {};
```

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 final : named_unit<"deg", mag_ratio<5667, 100> * Mile> {};
```

```
inline constexpr struct roman_foot final : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace final : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile final : 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 final : named_unit<"deg", mag_ratio<5667, 100> * Mile> {};
```

```
inline constexpr struct roman_foot final : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace final : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile final : 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 final : 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 final : named_unit<"deg", mag_ratio<5667, 100> * Mile> {};
```

```
inline constexpr struct roman_foot final : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace final : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile final : 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 final : named_unit<"mi_a", mag<2163> * si::metre> {} arabic_mile;
```

```
// 1 minute of arc along the Earth's equator
inline constexpr struct geographical_mile final :
    named_unit<"mi_g", mag_ratio<18'553, 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 final : named_unit<"deg", mag_ratio<5667, 100> * Mile> {};
```

```
inline constexpr struct roman_foot final : named_unit<"ft_r", mag<296> * si::milli<si::metre>> {} roman_foot;
inline constexpr struct roman_pace final : named_unit<"pace_r", mag<5> * roman_foot> {} roman_pace;
inline constexpr struct roman_mile final : 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 final : named_unit<"mi_a", mag<2163> * si::metre> {} arabic_mile;
```

```
// 1 minute of arc along the Earth's equator
inline constexpr struct geographical_mile final :
    named_unit<"mi_g", mag_ratio<18'553, 10> * si::metre> {} geographical_mile;
```

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

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. * equator_degree;
```

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. * equator_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. * equator_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;
```

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

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

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

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

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...

Not that easy at it seems

Implementing a physical quantities and units library is much harder than it may initially appear.

Not that easy at it seems

Implementing a physical quantities and units library is much harder than it may initially appear.

Besides the obvious candidates there are many other possible safety issues and corner cases to cover.

`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
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
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

```
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());
```

```
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

```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
```

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.

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 floor_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 floor_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 floor() 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 floor_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 floor() 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.

Discriminating between different lengths matters!

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.

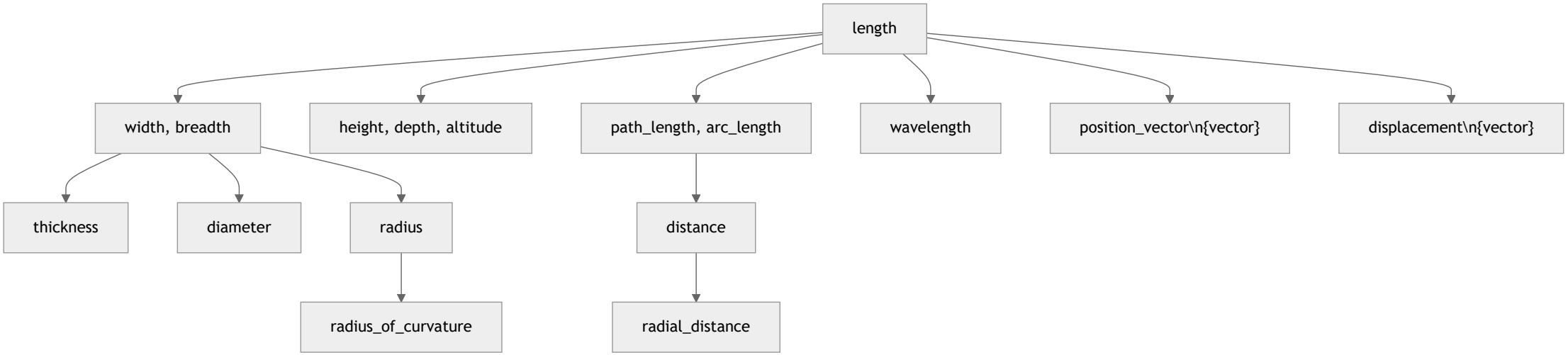
- **More than one quantity** may be defined **for the same dimension**
  - quantities of *different kinds* (e.g., *frequency*, *modulation rate*, *activity*, ...)
  - quantities of the *same kind* (e.g., *length*, *width*, *altitude*, *distance*, *radius*, *wavelength*, *position vector*, ...)
- Quantities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



Simple user extensions and composability

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

```
inline constexpr struct horizontal_length final : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area final :
    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 final : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area final :
    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 final : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area final :
    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 final : quantity_spec<isq::length> {} horizontal_length;
inline constexpr struct horizontal_area final :
    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]> floor() 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]> floor() 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]> floor() 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]> floor() 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));
      |                                                                          ^
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):
      |   ^~~
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):
      |       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
```

Discriminating between different lengths 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 hertz
```

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 hertz
```

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

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

```
3.0 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 hertz
```

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

```
3.0 becquerel
```

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 hertz
```

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

```
3.0 becquerel
```

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**

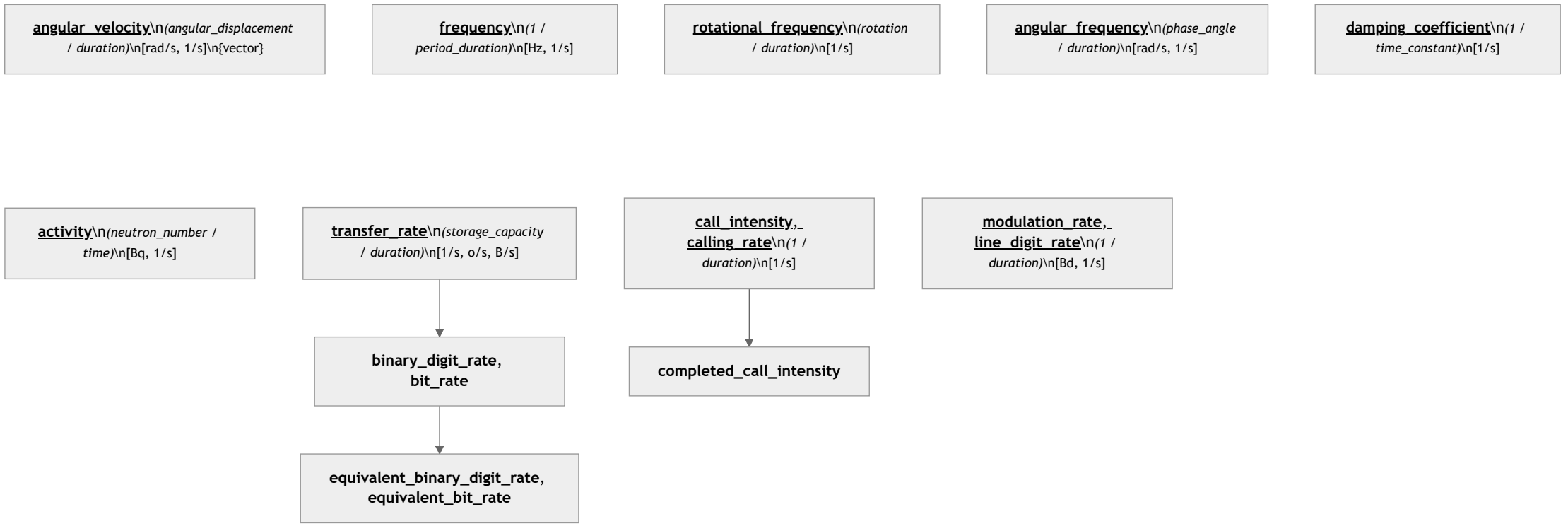
Quantity kinds (ISO 80000)

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

}
```

SI units for ISQ base quantities

```
namespace mp_units::si {

inline constexpr struct second final : named_unit<"s", kind_of<isq::time>> {} second;
inline constexpr struct metre final : named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct gram final : named_unit<"g", kind_of<isq::mass>> {} gram;
inline constexpr auto kilogram = kilo<gram>;
inline constexpr struct ampere final : named_unit<"A", kind_of<isq::electric_current>> {} ampere;
inline constexpr struct kelvin final : named_unit<"K", kind_of<isq::thermodynamic_temperature>> {} kelvin;
inline constexpr struct mole final : named_unit<"mol", kind_of<isq::amount_of_substance>> {} mole;
inline constexpr struct candela final : 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 final : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian final : named_unit<"sr", square(metre) / square(metre),
                                                     kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz final : named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel final : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton final : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal final : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule final : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt final : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb final : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian final : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian final : named_unit<"sr", square(metre) / square(metre),
                                                     kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz final : named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel final : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton final : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal final : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule final : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt final : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb final : 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>'))
   14 | 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]
  451 | [[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
  450 |   requires detail::CommonlyInvocableQuantities<std::plus<>, quantity<R1, Rep1>, quantity<R2, Rep2>>
      |            ^
note: because 'HaveCommonReference<quantity<hertz{}, int>::reference, quantity<struct becquerel{}, int>::reference>' evaluated to false
   91 |   Quantity<Q1> && Quantity<Q2> && HaveCommonReference<Q1::reference, Q2::reference> &&
      |                                   ^
note: because 'HaveCommonReferenceImpl<si::hertz{}, struct becquerel{}>' evaluated to false
   87 | concept HaveCommonReference = HaveCommonReferenceImpl<R1, R2>;
      |                               ^
note: because 'common_reference(R1, R2)' would be invalid: no matching function for call to 'common_reference'
   84 | concept HaveCommonReferenceImpl = requires { common_reference(R1, R2); };
      |                                              ^
```

It is not only about hertz and becquerel

Most of the libraries on the market happily allow such operations and provide incorrect results.

It is not only about hertz and becquerel

Most of the libraries on the market happily allow such operations and provide incorrect results.

```
quantity<Gy> q = 42 * Sv;                  // Compile-time error
bool b = (1 * rad + 1 * bit) == 2 * sr;    // Compile-time error
```

It is not only about hertz and becquerel

Most of the libraries on the market happily allow such operations and provide incorrect results.

```
quantity<Gy> q = 42 * Sv;                  // Compile-time error
bool b = (1 * rad + 1 * bit) == 2 * sr;    // Compile-time error
```

Dimension is not enough to describe a quantity!

The affine space abstractions

The affine space has two types of entities:

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

- One can do a **limited set of operations** in affine space on points and displacement 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
- **displacement vector** - the difference between two points

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

```
quantity_point temp1 = absolute<deg_C>(21.2);
quantity_point temp2 = absolute<deg_C>(21.4);
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*

The Guardian

Record heat

Malawi swelters with temperatures nearly 68F above average

Fixing The Guardian article

```
namespace si {

inline constexpr struct absolute_zero final :
    absolute_point_origin<absolute_zero, isq::thermodynamic_temperature> {} absolute_zero;

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

inline constexpr struct zeroth_degree_Celsius final :
    relative_point_origin<absolute<milli<kelvin>>(273'150)> {} zeroth_degree_Celsius;

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

}
```

Fixing The Guardian article

```
namespace si {

inline constexpr struct absolute_zero final :
    absolute_point_origin<absolute_zero, isq::thermodynamic_temperature> {} absolute_zero;

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

inline constexpr struct zeroth_degree_Celsius final :
    relative_point_origin<absolute<milli<kelvin>>(273'150)> {} zeroth_degree_Celsius;

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

}
```

```
namespace usc {

inline constexpr struct zeroth_degree_Fahrenheit final :
    relative_point_origin<absolute<mag_ratio<5, 9> * si::degree_Celsius>(-32)> {} zeroth_degree_Fahrenheit;

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

}
```

Fixing The Guardian article

```
const quantity_point max = absolute<deg_C>(43.);
const quantity_point avg = absolute<deg_C>(25.);
const quantity delta = max - avg;
```

Fixing The Guardian article

```
const quantity_point max = absolute<deg_C>(43.);
const quantity_point avg = absolute<deg_C>(25.);
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));
```

Fixing The Guardian article

```
const quantity_point max = absolute<deg_C>(43.);
const quantity_point avg = absolute<deg_C>(25.);
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));
```

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

Fixing The Guardian article

```
const quantity_point max = absolute<deg_C>(43.);
const quantity_point avg = absolute<deg_C>(25.);
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));
```

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

```
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 = absolute<deg_C>(43.);
const quantity_point avg = absolute<deg_C>(25.);
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));
```

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

```
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());
```

```
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 path length measured by the car's odometer,
- 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.

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 ...


CAUTION
Programming
is addictive
(and too much fun)