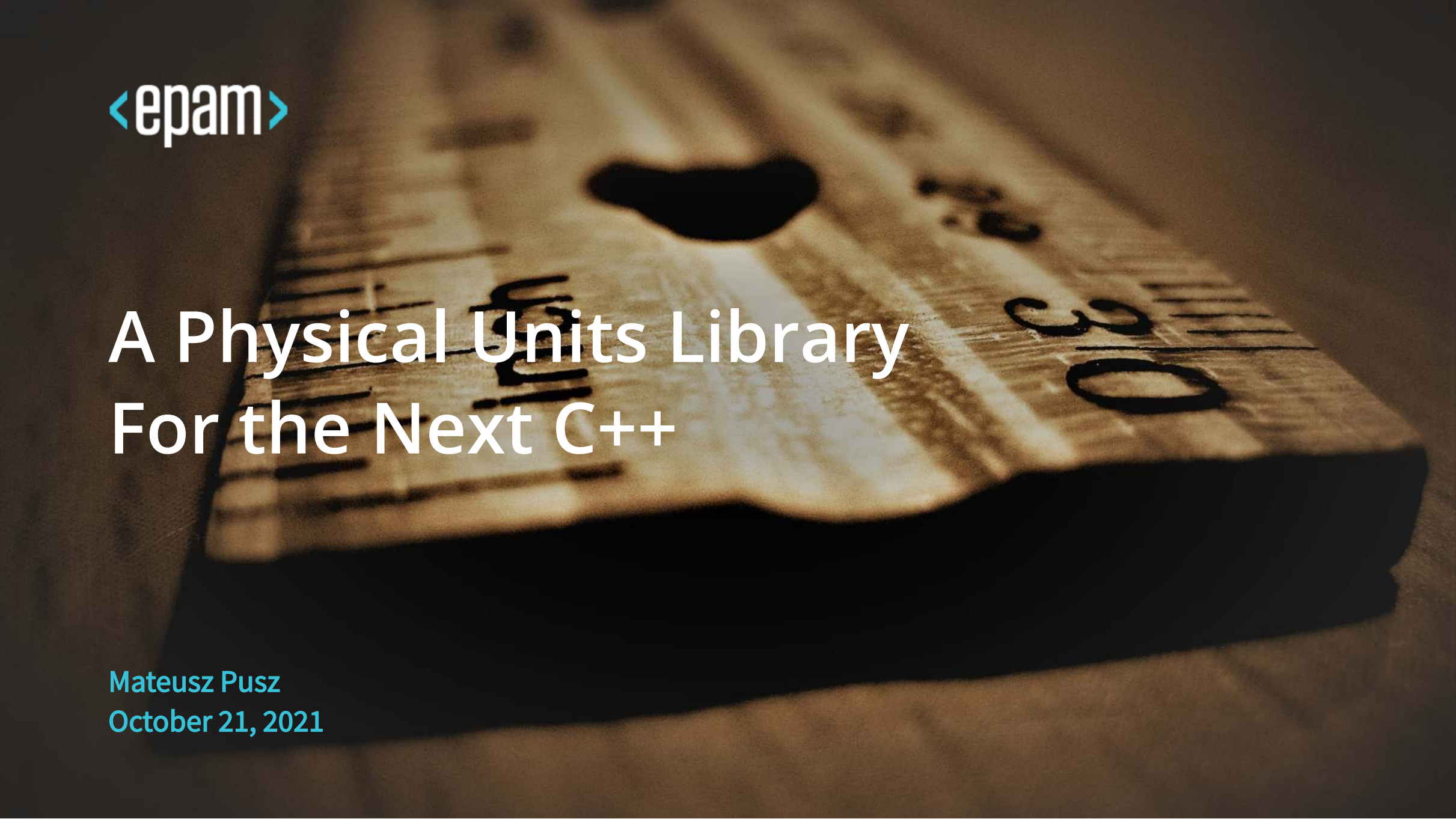
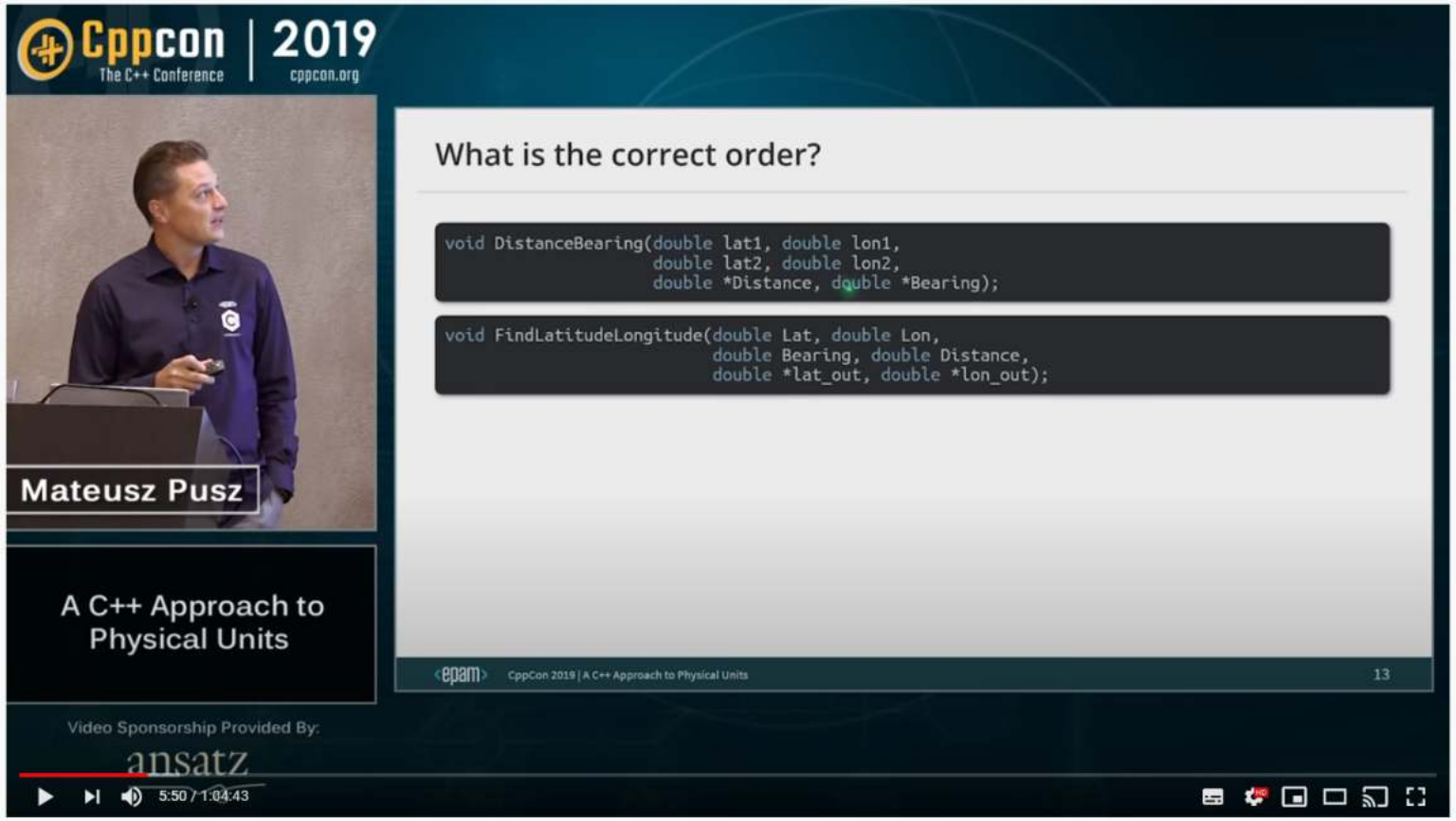


A Physical Units Library For the Next C++

Mateusz Pusz

October 21, 2021

Motivation, Existing Practice, Challenges...



AURORA
CppCon 2019: Mateusz Pusz "A C++ Approach to Physical Units"

Agenda

1. Quick Start
2. Strong Interfaces
3. As fast as (or even faster) than `double`
4. User Experience
5. Framework Basics
6. Environment, compatibility, next steps

Agenda

1. Quick Start
2. Strong Interfaces
3. As fast as (or even faster) than **`double`**
4. User Experience
5. Framework Basics
6. Environment, compatibility, next steps

In Q&A please refer to the slide number.

QUICK START

Physical Units library in a nutshell

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

Physical Units library in a nutshell

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

```
// unit conversions
static_assert(1_q_h == 3600_q_s);
static_assert(1_q_km + 1_q_m == 1001_q_m);
```

Physical Units library in a nutshell

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

```
// unit conversions
static_assert(1_q_h == 3600_q_s);
static_assert(1_q_km + 1_q_m == 1001_q_m);
```

```
// dimension conversions
static_assert(1_q_km / 1_q_s == 1000_q_m_per_s);
static_assert(2_q_km_per_h * 2_q_h == 4_q_km);
static_assert(2_q_km / 2_q_km_per_h == 1_q_h);

static_assert(2_q_m * 3_q_m == 6_q_m2);

static_assert(10_q_km / 5_q_km == 2);

static_assert(1000 / 1_q_s == 1_q_kHz);
```

Literals (godbolt.org/z/G77Eo936r)

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

```
// unit conversions
static_assert(1_q_h == 3600_q_s);
static_assert(1_q_km + 1_q_m == 1001_q_m);
```

```
// dimension conversions
static_assert(1_q_km / 1_q_s == 1000_q_m_per_s);
static_assert(2_q_km_per_h * 2_q_h == 4_q_km);
static_assert(2_q_km / 2_q_km_per_h == 1_q_h);

static_assert(2_q_m * 3_q_m == 6_q_m2);

static_assert(10_q_km / 5_q_km == 2);

static_assert(1000 / 1_q_s == 1_q_kHz);
```

References (godbolt.org/z/EPqTc57jx)

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

```
// unit conversions
static_assert(1 * h == 3600 * s);
static_assert(1 * km + 1 * m == 1001 * m);
```

```
// dimension conversions
inline constexpr auto kmph = km / h;
static_assert(1 * km / (1 * s) == 1000 * (m / s));
static_assert(2 * kmph * (2 * h) == 4 * km);
static_assert(2 * km / (2 * kmph) == 1 * h);

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

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

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

Aliases: Terse (godbolt.org/z/476sh6ohK)

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

```
// unit conversions
static_assert(h(1) == s(3600));
static_assert(km(1) + m(1) == m(1001));
```

```
// dimension conversions
static_assert(km(1) / s(1) == m_per_s(1000));
static_assert(km_per_h(2) * h(2) == km(4));
static_assert(km(2) / km_per_h(2) == h(1));

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

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

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

Aliases: Verbose (godbolt.org/z/EefKedner)

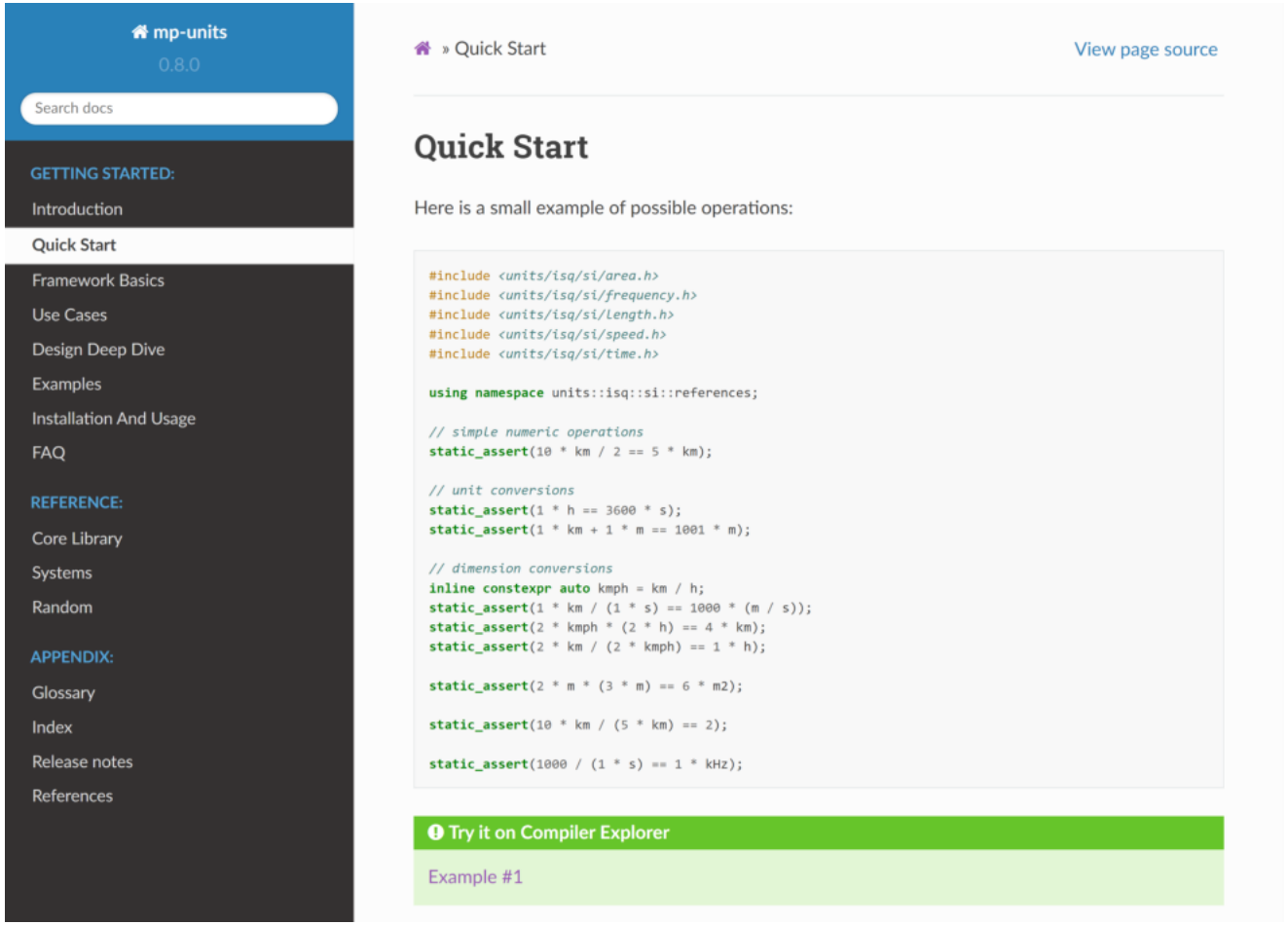
```
// simple numeric operations
static_assert(length::km(10) / 2 == length::km(5));
```

```
// unit conversions
static_assert(time::h(1) == time::s(3600));
static_assert(length::km(1) + length::m(1) == length::m(1001));
```

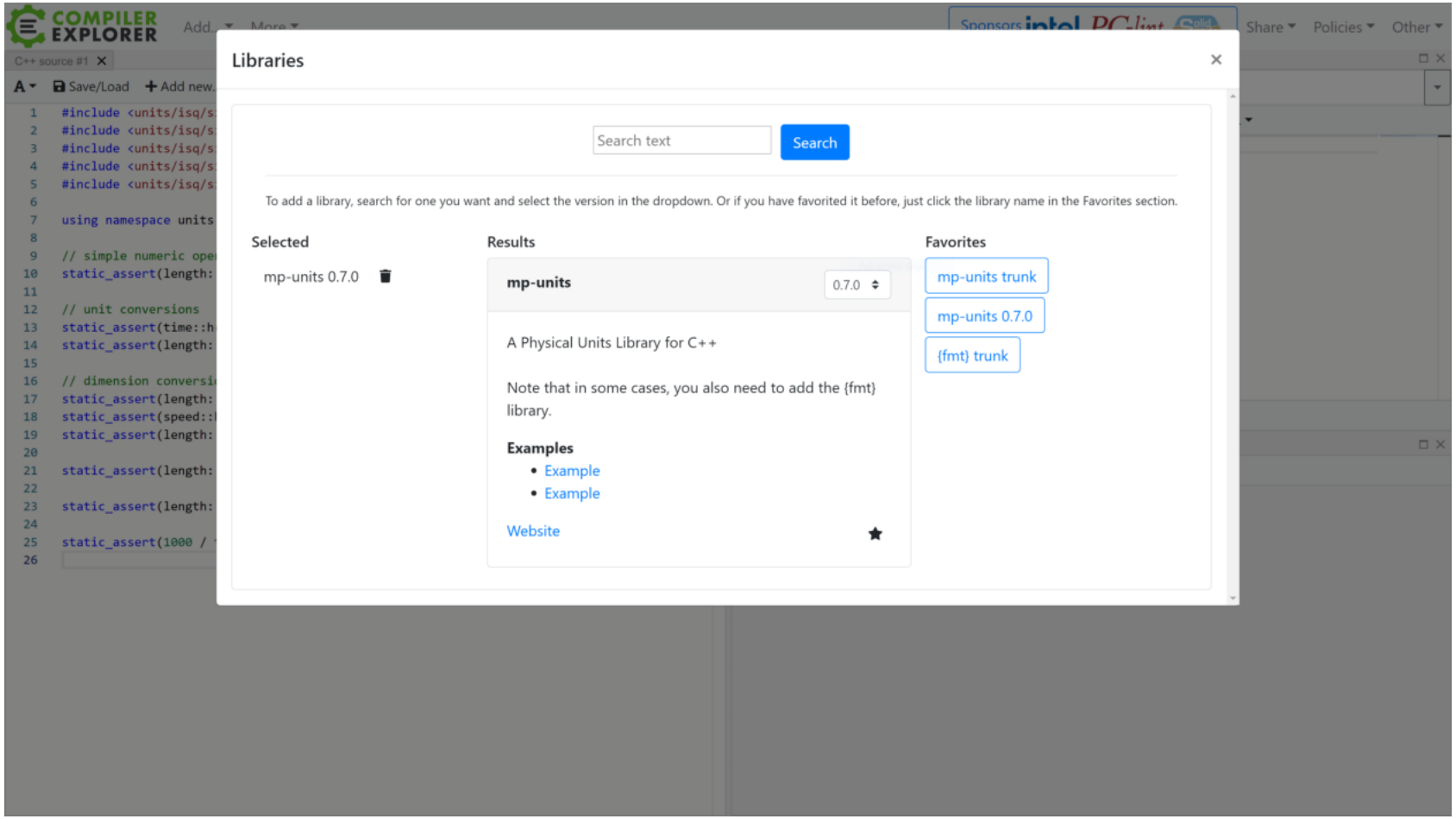
```
// dimension conversions
static_assert(length::km(1) / time::s(1) == speed::m_per_s(1000));
static_assert(speed::km_per_h(2) * time::h(2) == length::km(4));
static_assert(length::km(2) / speed::km_per_h(2) == time::h(1));

static_assert(length::m(2) * length::m(3) == area::m2(6));

static_assert(length::km(10) / length::km(5) == 2);

static_assert(1000 / time::s(1) == frequency::kHz(1));
```

Software Engineering Is About Tradeoffs

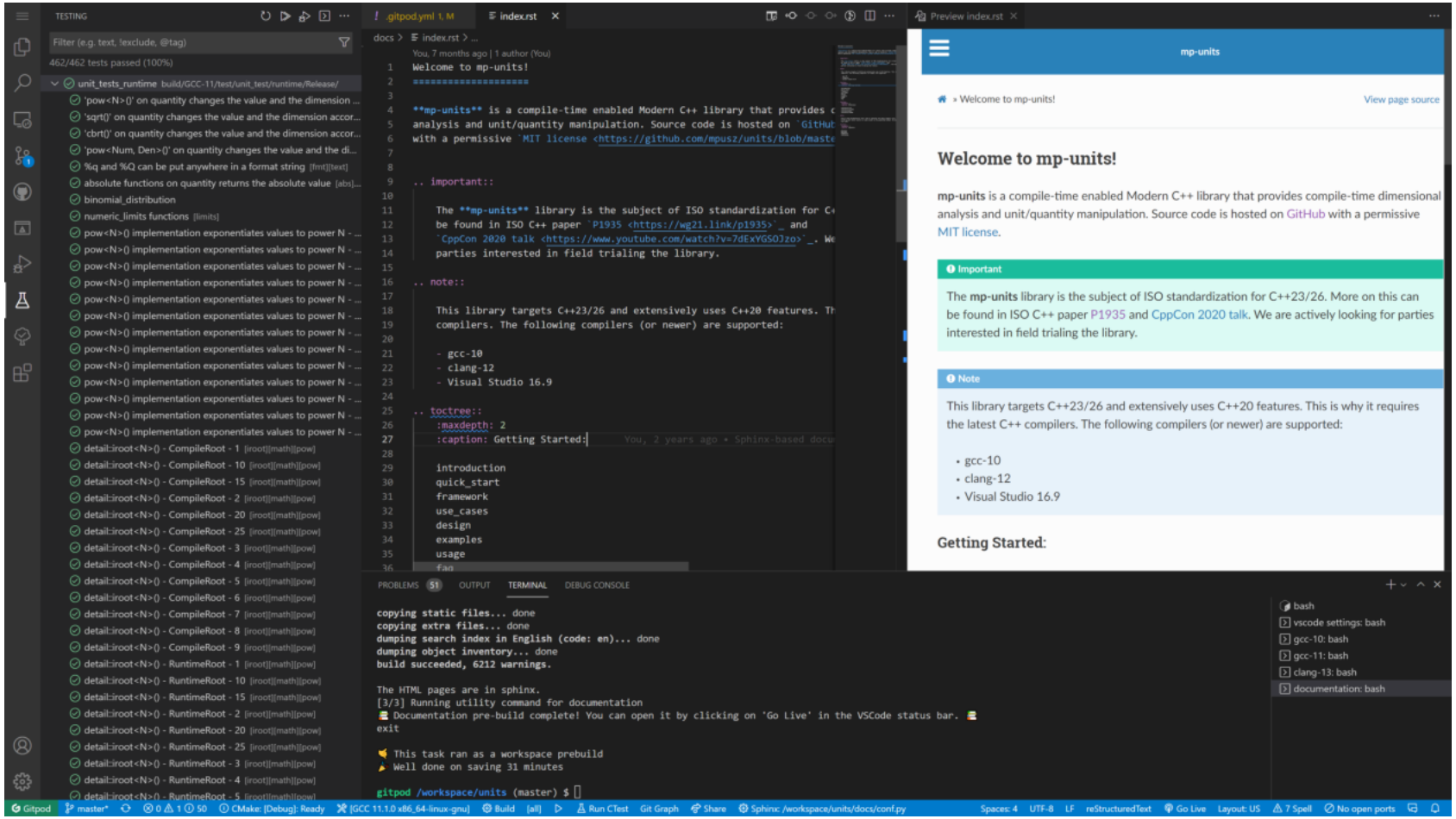
mp-units Documentation (mpusz.github.io/units)



mp-units on the Compiler Explorer



mp-units in GitPod



mp-units in Conan

CONAN CENTER (OFFICIAL RELEASES)

```
[requires]
mp-units/0.7.0

[generators]
CMakeToolchain
CMakeDeps
```

```
conan install .. -pr <your_conan_profile> -s compiler.cppstd=20 -b=missing
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
cmake --build .
```

mp-units in Conan

LIVE AT HEAD

```
[requires]
mp-units/0.8.0@mpusz/testing

[generators]
CMakeToolchain
CMakeDeps
```

```
conan remote add conan-mpusz https://mpusz.jfrog.io/artifactory/api/conan/conan-oss
```

```
conan install .. -pr <your_conan_profile> -s compiler.cppstd=20 -b=outdated -u
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
cmake --build .
```

Requirements

- **Compile-time** safety

Requirements

- Compile-time safety
- The best possible user experience
  - compiler errors
  - debugging

Requirements

- **Compile-time** safety
- The best possible **user experience**
  - compiler errors
  - debugging
- **As fast as `double`**

Requirements

- **Compile-time** safety
- The best possible **user experience**
  - compiler errors
  - debugging
- **As fast as `double`**
- Easy **extensibility**

Requirements

- Compile-time safety
- The best possible user experience
  - compiler errors
  - debugging
- As fast as `double`
- Easy extensibility
- No macros in the user interface

Requirements

- Compile-time safety
- The best possible user experience
  - compiler errors
  - debugging
- As fast as `double`
- Easy extensibility
- No macros in the user interface
- No external dependencies
- Possibility to be standardized as a freestanding part of the C++ Standard Library

STRONG INTERFACES

Toy example

```
/* speed */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

Toy example

```
/* speed */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

Toy example

```
/* speed */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- **Compile-time safety** to make sure that the result is of a correct dimension

Toy example

```
/* speed */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- **Compile-time safety** to make sure that the result is of a correct dimension
- Support for **multiple units and unit prefixes**

Toy example

```
/* speed */ avg_speed(/* length */ distance, /* time */ duration)
{
  return distance / duration;
}
```

```
const auto kmph = avg_speed(/* 220 km */, /* 2 hours */);
std::cout << /* kmph */ << " km/h\n";
```

```
const auto mph = avg_speed(/* 140 miles */, /* 2 hours */);
std::cout << /* mph */ << " mph\n";
```

- **Compile-time safety** to make sure that the result is of a correct dimension
- Support for **multiple units and unit prefixes**
- **No runtime overhead**
  - no additional intermediate conversions
  - as fast as a custom code implemented with **`double`**s

SI coherent units: `double` (godbolt.org/z/Wdc5Mo)

```
#define H_TO_S(duration) ((duration) * 3600)

#define KM_TO_M(distance) ((distance) * 1000)
#define MPS_TO_KMPH(velocity) ((velocity) * 3600 / 1000)

#define MI_TO_M(distance) ((distance) * 1609.344)
#define MPS_TO_MIPH(velocity) ((velocity) * 3600 / 1609.344)
```

SI coherent units: `double` (godbolt.org/z/Wdc5Mo)

```
#define H_TO_S(duration) ((duration) * 3600)

#define KM_TO_M(distance) ((distance) * 1000)
#define MPS_TO_KMPH(velocity) ((velocity) * 3600 / 1000)

#define MI_TO_M(distance) ((distance) * 1609.344)
#define MPS_TO_MIPH(velocity) ((velocity) * 3600 / 1609.344)
```

```
///
/// @brief Calculates average speed
///
/// @param d distance in metres
/// @param t time in seconds
/// @return speed in metres per second
///
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

SI coherent units: `double` (godbolt.org/z/Wdc5Mo) (Continued....)

```
auto toy_example_1(double d, double t)
{
  return MPS_TO_KMPH(avg_speed(KM_TO_M(d), H_TO_S(t)));
}

auto toy_example_2(double d, double t)
{
  return MPS_TO_MIPH(avg_speed(MI_TO_M(d), H_TO_S(t)));
}
```

SI coherent units: `double` (godbolt.org/z/Wdc5Mo) (Continued....)

```
auto toy_example_1(double d, double t)
{
  return MPS_TO_KMPH(avg_speed(KM_TO_M(d), H_TO_S(t)));
}

auto toy_example_2(double d, double t)
{
  return MPS_TO_MIPH(avg_speed(MI_TO_M(d), H_TO_S(t)));
}
```

```
std::cout << toy_example_1(220, 2) << "\n";  // prints "110"
std::cout << toy_example_2(140, 2) << "\n";  // prints "70"
```

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8)

```cpp
#include <boost/units/base_units/metric/hour.hpp>
#include <boost/units/base_units/us/mile.hpp>
#include <boost/units/io.hpp>
#include <boost/units/make_scaled_unit.hpp>
#include <boost/units/quantity.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/time.hpp>
#include <boost/units/systems/si/velocity.hpp>
#include <boost/units/systems/si/prefixes.hpp>

namespace bu = boost::units;

constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d / t;
}
```

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8)

```cpp
#include <boost/units/base_units/metric/hour.hpp>
#include <boost/units/base_units/us/mile.hpp>
#include <boost/units/io.hpp>
#include <boost/units/make_scaled_unit.hpp>
#include <boost/units/quantity.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/time.hpp>
#include <boost/units/systems/si/velocity.hpp>
#include <boost/units/systems/si/prefixes.hpp>

namespace bu = boost::units;

constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d / t;
}
```

Not easy to include all the necessary header files properly.
Compilation errors hard to understand.

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8) (Continued...)

```
using kilometer_base_unit = bu::make_scaled_unit<bu::si::length, bu::scale<10, bu::static_rational<3>>>::type;
using length_kilometer = kilometer_base_unit::unit_type;

using length_mile = bu::us::mile_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(miles, length_mile);

using time_hour = bu::metric::hour_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(hours, time_hour);

using velocity_kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
BOOST_UNITS_STATIC_CONSTANT(kilometers_per_hour, velocity_kilometers_per_hour);

using velocity_miles_per_hour = bu::divide_typeof_helper<length_mile, time_hour>::type;
BOOST_UNITS_STATIC_CONSTANT(miles_per_hour, velocity_miles_per_hour);
```

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8) (Continued...)

```
using kilometer_base_unit = bu::make_scaled_unit<bu::si::length, bu::scale<10, bu::static_rational<3>>>::type;
using length_kilometer = kilometer_base_unit::unit_type;

using length_mile = bu::us::mile_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(miles, length_mile);

using time_hour = bu::metric::hour_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(hours, time_hour);

using velocity_kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
BOOST_UNITS_STATIC_CONSTANT(kilometers_per_hour, velocity_kilometers_per_hour);

using velocity_miles_per_hour = bu::divide_typeof_helper<length_mile, time_hour>::type;
BOOST_UNITS_STATIC_CONSTANT(miles_per_hour, velocity_miles_per_hour);
```

Not easy at all to work with non-coherent units.

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8) (Continued...)

```
auto toy_example_1(bu::quantity<length_kilometer> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_kilometers_per_hour>(v);
}
```

```
auto toy_example_2(bu::quantity<length_mile> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_miles_per_hour>(v);
}
```

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8) (Continued...)

```
auto toy_example_1(bu::quantity<length_kilometer> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_kilometers_per_hour>(v);
}
```

```
auto toy_example_2(bu::quantity<length_mile> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_miles_per_hour>(v);
}
```

No implicit conversions between quantities of the same dimension and compatible units.

SI coherent units: Boost.Units (godbolt.org/z/G3qqn8) (Continued...)

```
auto toy_example_1(bu::quantity<length_kilometer> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_kilometers_per_hour>(v);
}
```

```
auto toy_example_2(bu::quantity<length_mile> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_miles_per_hour>(v);
}
```

```
const auto v1 = toy_example_1(220 * bu::si::kilo * bu::si::meters, 2 * hours);
const auto v2 = toy_example_2(140 * miles, 2 * hours);
std::cout << v1 << "\n";     // prints "110 k(m h^-1)"
std::cout << v2 << "\n";     // prints "70 mi h^-1"
```

No implicit conversions between quantities of the same dimension and compatible units.

SI coherent units: Nic Holthaus (godbolt.org/z/jj63xW)

```cpp
#include <units.h>

using namespace units;

constexpr velocity::meters_per_second_t avg_speed(length::meter_t d, time::second_t t)
{
  return d / t;
}
```

SI coherent units: Nic Holthaus (godbolt.org/z/jj63xW)

```cpp
#include <units.h>

using namespace units;

constexpr velocity::meters_per_second_t avg_speed(length::meter_t d, time::second_t t)
{
  return d / t;
}
```

```cpp
auto toy_example_1(length::kilometer_t d, time::hour_t t)
{
  return velocity::kilometers_per_hour_t(avg_speed(d, t));
}
```

```cpp
auto toy_example_2(length::mile_t d, time::hour_t t)
{
  return velocity::miles_per_hour_t(avg_speed(d, t));
}
```

SI coherent units: Nic Holthaus (godbolt.org/z/jj63xW)

```cpp
#include <units.h>

using namespace units;

constexpr velocity::meters_per_second_t avg_speed(length::meter_t d, time::second_t t)
{
  return d / t;
}
```

```cpp
auto toy_example_1(length::kilometer_t d, time::hour_t t)
{
  return velocity::kilometers_per_hour_t(avg_speed(d, t));
}
```

```cpp
auto toy_example_2(length::mile_t d, time::hour_t t)
{
  return velocity::miles_per_hour_t(avg_speed(d, t));
}
```

```cpp
using namespace units::literals;
std::cout << toy_example_1(220_km, 2_hr) << "\n";  // prints "30.5556 m s^-1"
std::cout << toy_example_2(140_mi, 2_hr) << "\n";  // prints "31.2928 m s^-1"
```

SI coherent units: mp-units (godbolt.org/z/zeMEhP3MY)

```
#include <units/isq/si/speed.h>
#include <units/isq/si/international/speed.h>
#include <units/quantity_io.h>

using namespace units::aliases::isq::si;

constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d / t;
}
```

SI coherent units: mp-units (godbolt.org/z/zeMEhP3MY)

```
#include <units/isq/si/speed.h>
#include <units/isq/si/international/speed.h>
#include <units/quantity_io.h>

using namespace units::aliases::isq::si;

constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d / t;
}
```

```
auto toy_example_1(length::km<> d, time::h<> t)
{
  return km_per_h(avg_speed(d, t));
}
```

```
auto toy_example_2(international::length::mi<> d, time::h<> t)
{
  return international::mi_per_h(avg_speed(d, t));
}
```

SI coherent units: mp-units (godbolt.org/z/zeMEhP3MY)

```
#include <units/isq/si/speed.h>
#include <units/isq/si/international/speed.h>
#include <units/quantity_io.h>

using namespace units::aliases::isq::si;

constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d / t;
}
```

```
auto toy_example_1(length::km<> d, time::h<> t)
{
  return km_per_h(avg_speed(d, t));
}
```

```
auto toy_example_2(international::length::mi<> d, time::h<> t)
{
  return international::mi_per_h(avg_speed(d, t));
}
```

```
std::cout << toy_example_1(km(220), h(2)) << "\n";                  // prints "110 km/h"
std::cout << toy_example_2(international::mi(140), h(2)) << "\n";  // prints "70 mi/h"
```

A need for generic interfaces

```
km_per_h v1 = avg_speed(km, h);
mi_per_h v2 = avg_speed(mi, h);
```

- We should not pay for any intermediate conversions
- **avg_speed()** should just
  - divide the two arguments
  - return a correct type

Generic code: `double` (godbolt.org/z/dGe8b4)

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

Generic code: **double** (godbolt.org/z/dGe8b4)

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

```
auto toy_example_1(double d_km, double t_h)
{
  return avg_speed(d_km, t_h);
}

auto toy_example_2(double d_mi, double t_h)
{
  return avg_speed(d_mi, t_h);
}
```

Well, it is simple and works, right? ;-)

Generic code: Boost.Units

```
template<typename System1, typename Rep1, typename System2, typename Rep2>
constexpr bu::quantity<typename bu::divide_typeof_helper<
                         bu::unit<bu::length_dimension, System1>,
                         bu::unit<bu::time_dimension, System2>>::type>
avg_speed(bu::quantity<bu::unit<bu::length_dimension, System1>, Rep1> d,
          bu::quantity<bu::unit<bu::time_dimension, System2>, Rep2> t)
{
  return d / t;
}
```

No easy way to constrain a return type.

Generic code: Nic Holthaus

```
template<typename Length, typename Time,
         typename = std::enable_if_t<units::traits::is_length_unit<Length>::value &&
                                     units::traits::is_time_unit<Time>::value>>
constexpr auto avg_speed(Length d, Time t)
{
  const auto v = d / t;
  static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
  return v;
}
```

Again no easy way to constrain a return type.

Generic code: Boost.Units + Concepts (godbolt.org/z/erfbzr)

```
#include <boost/units/is_quantity_of_dimension.hpp>
```

Generic code: Boost.Units + Concepts (godbolt.org/z/erfbzr)

```
#include <boost/units/is_quantity_of_dimension.hpp>
```

```
template<typename Quantity, typename Dimension>
concept QuantityOf = bu::is_quantity_of_dimension<Quantity, Dimension>::value;

template<typename Quantity>
concept Length = QuantityOf<Quantity, bu::length_dimension>;

template<typename Quantity>
concept Time = QuantityOf<Quantity, bu::time_dimension>;

template<typename Quantity>
concept Velocity = QuantityOf<Quantity, bu::velocity_dimension>;
```

Generic code: Boost.Units + Concepts (godbolt.org/z/erfbzr)

```
#include <boost/units/is_quantity_of_dimension.hpp>
```

```
template<typename Quantity, typename Dimension>
concept QuantityOf = bu::is_quantity_of_dimension<Quantity, Dimension>::value;

template<typename Quantity>
concept Length = QuantityOf<Quantity, bu::length_dimension>;

template<typename Quantity>
concept Time = QuantityOf<Quantity, bu::time_dimension>;

template<typename Quantity>
concept Velocity = QuantityOf<Quantity, bu::velocity_dimension>;
```

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

Generic code: Nic Holthaus + Concepts (godbolt.org/z/zen4KE)

```
template<typename T>
concept Length = units::traits::is_length_unit<T>::value;

template<typename T>
concept Time = units::traits::is_time_unit<T>::value;

template<typename T>
concept Velocity = units::traits::is_velocity_unit<T>::value;
```

Generic code: Nic Holthaus + Concepts (godbolt.org/z/zen4KE)

```
template<typename T>
concept Length = units::traits::is_length_unit<T>::value;

template<typename T>
concept Time = units::traits::is_time_unit<T>::value;

template<typename T>
concept Velocity = units::traits::is_velocity_unit<T>::value;
```

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

Generic code: mp-units (godbolt.org/z/14GK4dEo6)

```
using namespace units::isq;
```

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

Generic code: mp-units (godbolt.org/z/14GK4dEo6)

```
using namespace units::isq;
```

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
const Speed auto v1 = avg_speed(km(220), h(2));
const Speed auto v2 = avg_speed(international::mi(140), h(2));
std::cout << v1 << '\n';
std::cout << v2 << '\n';
```

Generic code: mp-units (godbolt.org/z/14GK4dEo6)

```
using namespace units::isq;
```

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
const Speed auto v1 = avg_speed(km(220), h(2));
const Speed auto v2 = avg_speed(international::mi(140), h(2));
std::cout << v1 << '\n';
std::cout << v2 << '\n';
```

```
110 km/h
70 mi/h
```

Generic code: Not only about handling different units

```
using namespace units::isq;
```

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
using namespace std::math;
const Speed auto v1 = avg_speed(km(fs_vector<int, 3>{ 40, 20, 30 }), h(2));
const Speed auto v2 = avg_speed(international::mi(fs_vector<int, 3>{ 40, 20, 30 }), h(2));
std::cout << v1 << '\n';
std::cout << v2 << '\n';
```

```
|        20        10        15 | km/h
|        20        10        15 | mi/h
```

AS FAST AS (OR EVEN FASTER) THAN `double`

SI coherent units: `double` (godbolt.org/z/eq3jWq)

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}

auto toy_example_1(double d, double t)
{
  return MPS_TO_KMPH(avg_speed(KM_TO_M(d), H_TO_S(t)));
}
```

```
toy_example_1(...):
        movsd   xmm2, QWORD PTR .LC0[rip]
        movsd   xmm3, QWORD PTR .LC1[rip]
        mulsd   xmm0, xmm2
        mulsd   xmm1, xmm3
        divsd   xmm0, xmm1
        mulsd   xmm0, xmm3
        divsd   xmm0, xmm2
        ret
```

- 3x multiply
- 2x divide

SI coherent units: Boost.Units (godbolt.org/z/5o3xx3)

```cpp
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d, bu::quantity<bu::si::time> t)
{
  return d / t;
}

auto toy_example_1(bu::quantity<length_kilometer> d, bu::quantity<time_hour> t)
{
  const auto v = avg_speed(bu::quantity<bu::si::length>(d), bu::quantity<bu::si::time>(t));
  return bu::quantity<velocity_kilometers_per_hour>(v);
}
```

```
toy_example_1(...):
        movsd   xmm0, QWORD PTR .LC0[rip]
        movsd   xmm1, QWORD PTR .LC1[rip]
        mov     rax, rdi
        mulsd   xmm0, QWORD PTR [rsi]
        mulsd   xmm1, QWORD PTR [rdx]
        divsd   xmm0, xmm1
        mulsd   xmm0, QWORD PTR .LC2[rip]
        movsd   QWORD PTR [rdi], xmm0
        ret
```

- 3x multiply
- 1x divide

SI coherent units: Nic Holthaus (godbolt.org/z/4fEqbG)

```
constexpr velocity::meters_per_second_t avg_speed(length::meter_t d, time::second_t t)
{
  return d / t;
}

auto toy_example_1(length::kilometer_t d, time::hour_t t)
{
  return velocity::kilometers_per_hour_t(avg_speed(d, t));
}
```

```
toy_example_1(...):
        mulsd   xmm0, QWORD PTR .LC0[rip]
        mulsd   xmm1, QWORD PTR .LC1[rip]
        divsd   xmm0, xmm1
        mulsd   xmm0, QWORD PTR .LC2[rip]
        divsd   xmm0, QWORD PTR .LC3[rip]
        ret
```

- 3x multiply
- 2x divide

SI coherent units: mp-units (godbolt.org/z/8Mxorvs13)

```
constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d / t;
}

auto toy_example_1(length::km<> d, time::h<> t)
{
  return km_per_h(avg_speed(d, t));
}
```

```
toy_example_1(...):
        mulsd   xmm0, QWORD PTR .LC0[rip]
        mulsd   xmm1, QWORD PTR .LC1[rip]
        divsd   xmm0, xmm1
        mulsd   xmm0, QWORD PTR .LC2[rip]
        ret
```

- 3x multiply
- 1x divide

Generic code: **double** (godbolt.org/z/zEGcPr)

```
constexpr double avg_speed(double d, double t)
{
  return d / t;
}

auto toy_example_1(double d, double t)
{
  return avg_speed(d, t);
}
```

```
toy_example_1(...):
        divsd   xmm0, xmm1
        ret
```

Generic code: Boost.Units (godbolt.org/z/Y7zKr6)

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}

auto toy_example_1(bu::quantity<length_kilometer> d, bu::quantity<time_hour> t)
{
  return avg_speed(d, t);
}
```

```
toy_example_1(...):
        movsd   xmm0, QWORD PTR [rsi]
        mov     rax, rdi
        divsd   xmm0, QWORD PTR [rdx]
        movsd   QWORD PTR [rdi], xmm0
        ret
```

Generic code: Nic Holthaus (godbolt.org/z/87avz4)

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}

auto toy_example_1(length::kilometer_t d, time::hour_t t)
{
  return avg_speed(d, t);
}
```

```
toy_example_1(...):
        divsd   xmm0, xmm1
        ret
```

Generic code: mp-units (godbolt.org/z/Gsc3nEe3T)

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}

auto toy_example_1(length::km<> d, time::h<> t)
{
  return avg_speed(d, t);
}
```

```
toy_example_1(...):
        divsd   xmm0, xmm1
        ret
```

USER EXPERIENCE

SI coherent units: `double`

```
constexpr double avg_speed(double d, double t)
{
  return d * t;
}
```

Compiles fine with all sorts of bugs. Errors at runtime.

SI coherent units: Boost.Units

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d * t;
}
```

```
<source>: In function 'constexpr boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<
boost::units::length_base_dimension, boost::units::static_rational<1> >, boost::units::list<boost::units::dim<
boost::units::time_base_dimension, boost::units::static_rational<-1> >, boost::units::dimensionless_type> >,
boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit, boost::units::list<
boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10,
boost::units::static_rational<3> > >, boost::units::list<boost::units::si::second_base_unit, boost::units::list<
boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<
boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit, boost::units::list<
boost::units::angle::radian_base_unit, boost::units::list<boost::units::angle::steradian_base_unit,
boost::units::dimensionless_type> > > > > > > > > > > avg_speed(boost::units::quantity<boost::units::unit<
boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1> >,
boost::units::dimensionless_type>, boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit,
boost::units::list<boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10,
boost::units::static_rational<3> > >, boost::units::list<boost::units::si::second_base_unit,
boost::units::list<boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit,
boost::units::list<boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit,
boost::units::list<boost::units::angle::radian_base_unit, boost::units::list<boost::units::angle::steradian_base_unit,
boost::units::dimensionless_type> > > > > > > > > > >, boost::units::quantity<boost::units::unit<boost::units::list<
boost::units::dim<boost::units::time_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
...
```

SI coherent units: Boost.Units (Continued...)

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d * t;
}
```

```
...
boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit, boost::units::list<
boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit,
boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit,
boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > > >)':

<source>:16:12: error: could not convert 'boost::units::operator*(const boost::units::quantity<Unit1, X>&, const
boost::units::quantity<Unit2, Y>&) [with Unit1 = boost::units::unit<boost::units::list<boost::units::dim<
boost::units::length_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit, boost::units::list<
boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit,
boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit,
boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > > >;
Unit2 = boost::units::unit<boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::homogeneous_system<
...
```

SI coherent units: Boost.Units (Continued...)

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d * t;
}
```

```
...
boost::units::list<boost::units::si::meter_base_unit, boost::units::list<boost::units::scaled_base_unit<
boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::list<
boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit, boost::units::list<
boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit, boost::units::list<
boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit, boost::units::list<
boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > > >; X = double; Y = double;
typename boost::units::multiply_typeof_helper<boost::units::quantity<Unit1, X>, boost::units::quantity<Unit2, Y> >::type =
boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1> >, boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type> >, boost::units::homogeneous_system<
boost::units::list<boost::units::si::meter_base_unit, boost::units::list<boost::units::scaled_base_unit<
boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit,
boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit,
boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > >, void>,
...
```

SI coherent units: Boost.Units (Continued...)

```
constexpr bu::quantity<bu::si::velocity> avg_speed(bu::quantity<bu::si::length> d,
                                                   bu::quantity<bu::si::time> t)
{
  return d * t;
}
```

```
...
double>](t)' from 'quantity<unit<list<[...],list<dim<[...],static_rational<1>>,[...]>>,[...],[...]>,[...]>' to
'quantity<unit<list<[...],list<dim<[...],static_rational<-1>>,[...]>>,[...],[...]>,[...]>'

   16 |   return d * t;
      |          ~~^~~
      |            |
      |            quantity<unit<list<[...],list<dim<[...],static_rational<1>>,[...]>>,[...],[...]>,[...]>

Compiler returned: 1
```

SI coherent units: Nic Holthaus

```
constexpr velocity::meters_per_second_t avg_speed(length::meter_t d, time::second_t t)
{
  return d * t;
}
```

```
units.h: In instantiation of 'constexpr T units::convert(const T&) [with UnitFrom = units::unit<std::ratio<1>,
units::base_unit<std::ratio<1>, std::ratio<0, 1>, std::ratio<1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>,
std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1> >, std::ratio<0, 1>, std::ratio<0, 1> >; UnitTo = units::unit<
std::ratio<1>, units::base_unit<std::ratio<1>, std::ratio<0, 1>, std::ratio<-1> > >; T = double]':

units.h:1956:41:   required from 'constexpr units::unit_t<Units, T, NonLinearScale>::unit_t(const units::unit_t<
UnitsRhs, Ty, NlsRhs>&) [with UnitsRhs = units::unit<std::ratio<1>, units::base_unit<std::ratio<1>, std::ratio<0, 1>,
std::ratio<1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>,
std::ratio<0, 1> >, std::ratio<0, 1>, std::ratio<0, 1> >; Ty = double; NlsRhs = units::linear_scale; Units =
units::unit<std::ratio<1>, units::base_unit<std::ratio<1>, std::ratio<0, 1>, std::ratio<-1> > >; T = double;
NonLinearScale = units::linear_scale]'

<source>:7:14:   required from here

units.h:1620:64: error: static assertion failed: Units are not compatible.

Compiler returned: 1
```

SI coherent units: mp-units

```
constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d * t;
}
```

```
<source>: In function 'constexpr units::aliases::isq::si::speed::m_per_s<> avg_speed(
  units::aliases::isq::si::length::m<>, units::aliases::isq::si::time::s<>)':

<source>:8:12: error: could not convert 'units::operator*<units::quantity<units::isq::si::dim_length,
units::isq::si::metre, double>, units::quantity<units::isq::si::dim_time, units::isq::si::second,
double> >(d, t)' from 'quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, 1, 1>,
units::exponent<units::isq::si::dim_time, 1, 1> >,units::unknown_coherent_unit,[...]>' to
'quantity<units::isq::si::dim_speed,units::isq::si::metre_per_second,[...]>'

    8 |   return d * t;
      |          ~~^~~
      |            |
      |            quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, 1, 1>,
      |                     units::exponent<units::isq::si::dim_time, 1, 1> >,units::unknown_coherent_unit,[...]>

Compiler returned: 1
```

SI coherent units: mp-units

```
constexpr speed::m_per_s<> avg_speed(length::m<> d, time::s<> t)
{
  return d * t;
}
```

```
<source>: In function 'constexpr units::aliases::isq::si::speed::m_per_s<> avg_speed(
  units::aliases::isq::si::length::m<>, units::aliases::isq::si::time::s<>)':

<source>:8:12: error: could not convert 'units::operator*<units::quantity<units::isq::si::dim_length,
units::isq::si::metre, double>, units::quantity<units::isq::si::dim_time, units::isq::si::second,
double> >(d, t)' from 'quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, 1, 1>,
units::exponent<units::isq::si::dim_time, 1, 1> >,units::unknown_coherent_unit,[...]>' to
'quantity<units::isq::si::dim_speed,units::isq::si::metre_per_second,[...]>'

    8 |   return d * t;
      |          ~~^~~
      |            |
      |            quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, 1, 1>,
      |                     units::exponent<units::isq::si::dim_time, 1, 1> >,units::unknown_coherent_unit,[...]>

Compiler returned: 1
```

Nicely named strong types are preserved thanks to avoiding type aliasing.

Dimension mismatch: Boost.Units

```cpp
#include <boost/units/quantity.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/time.hpp>
#include <boost/units/systems/si/velocity.hpp>
#include <boost/units/systems/si/acceleration.hpp>
#include <boost/units/systems/si/prefixes.hpp>

namespace bu = boost::units;

bu::quantity<bu::si::acceleration> a = 100. * bu::si::meters / (10 * bu::si::second);
```

Dimension mismatch: Boost.Units

```cpp
#include <boost/units/quantity.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/time.hpp>
#include <boost/units/systems/si/velocity.hpp>
#include <boost/units/systems/si/acceleration.hpp>
#include <boost/units/systems/si/prefixes.hpp>

namespace bu = boost::units;

bu::quantity<bu::si::acceleration> a = 100. * bu::si::meters / (10 * bu::si::second);
```

```
<source>:10:62: error: conversion from 'quantity<unit<list<[...],list<dim<[...],static_rational<-1>>,[...]>>,[...]>,[...]>'
 to non-scalar type 'quantity<unit<list<[...],list<dim<[...],static_rational<-2>>,[...]>>,[...]>,[...]>' requested

   10 | bu::quantity<bu::si::acceleration> a = 100. * bu::si::meters / (10 * bu::si::second);
      |                                        ~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~

Compiler returned: 1
```

Dimension mismatch: Nic Holthaus

```
#include <units.h>

using namespace units;
using namespace units::literals;

acceleration::meters_per_second_squared_t a = 100_m / 10_s;
```

Dimension mismatch: Nic Holthaus

```cpp
#include <units.h>

using namespace units;
using namespace units::literals;

acceleration::meters_per_second_squared_t a = 100_m / 10_s;
```

```
units.h: In instantiation of 'constexpr T units::convert(const T&) [with UnitFrom = units::unit<std::ratio<1>, units::base_unit<
std::ratio<1>, std::ratio<0, 1>, std::ratio<-1> > >; UnitTo = units::unit<std::ratio<1>, units::base_unit<std::ratio<1>,
std::ratio<0, 1>, std::ratio<-2> > >; T = double]':

units.h:1956:41:   required from 'constexpr units::unit_t<Units, T, NonLinearScale>::unit_t(const units::unit_t<UnitsRhs, Ty,
NlsRhs>&) [with UnitsRhs = units::unit<std::ratio<1>, units::base_unit<std::ratio<1>, std::ratio<0, 1>, std::ratio<-1> > >;
Ty = double; NlsRhs = units::linear_scale; Units = units::unit<std::ratio<1>, units::base_unit<std::ratio<1>, std::ratio<0, 1>,
std::ratio<-2> > >; T = double; NonLinearScale = units::linear_scale]'

<source>:6:55:   required from here

units.h:1620:64: error: static assertion failed: Units are not compatible.

Compiler returned: 1
```

Dimension mismatch: mp-units

```
#include <units/isq/si/acceleration.h>

using namespace units::aliases::isq::si;

acceleration::m_per_s2<> a = m(100) / s(10);
```

Dimension mismatch: mp-units

```cpp
#include <units/isq/si/acceleration.h>

using namespace units::aliases::isq::si;

acceleration::m_per_s2<> a = m(100) / s(10);
```

```
<source>:6:37: error: conversion from 'quantity<units::isq::si::dim_speed,units::isq::si::metre_per_second,[...]>'
 to non-scalar type 'quantity<units::isq::si::dim_acceleration,units::isq::si::metre_per_second_sq,[...]>' requested

    6 | acceleration::m_per_s2<> a = m(100) / s(10);
      |                              ~~~~~~~^~~~~~~

Compiler returned: 1
```

Dimension mismatch: mp-units

```cpp
#include <units/isq/si/acceleration.h>

using namespace units::aliases::isq::si;

acceleration::m_per_s2<> a = m(100) / s(10);
```

```
<source>:6:37: error: conversion from 'quantity<units::isq::si::dim_speed,units::isq::si::metre_per_second,[...]>'
 to non-scalar type 'quantity<units::isq::si::dim_acceleration,units::isq::si::metre_per_second_sq,[...]>' requested

    6 | acceleration::m_per_s2<> a = m(100) / s(10);
      |                              ~~~~~~~^~~~~~~

Compiler returned: 1
```

The library is able to reconstruct a nicely named strong type from pieces (the Downcasting Facility).

Debugging: Boost.Units

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(220 * bu::si::kilo * bu::si::meters, 2 * hours);
```

Debugging: Boost.Units

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(220 * bu::si::kilo * bu::si::meters, 2 * hours);
```

```
Breakpoint 2, avg_speed<boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<
boost::units::length_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list<
boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >,
boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim<
boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::dimensionless_type> > >, void> >,
boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::heterogeneous_system<
boost::units::heterogeneous_system_impl<boost::units::list<boost::units::heterogeneous_system_dim<
boost::units::scaled_base_unit<boost::units::si::second_base_unit, boost::units::scale<60, boost::units::static_rational<2> > >,
boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::dim<
boost::units::time_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>,
boost::units::dimensionless_type> >, void> > > (d=..., t=...) at ../../src/main.cpp:41
41        return d / t;
```

Debugging: Nic Holthaus

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(220_km, 2_hr);
```

Debugging: Nic Holthaus

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(220_km, 2_hr);
```

```
Breakpoint 2, avg_speed<units::unit_t<units::unit<std::ratio<1000, 1>, units::unit<std::ratio<1>, units::base_unit<
std::ratio<1> > >, std::ratio<0, 1>, std::ratio<0, 1> > >, units::unit_t<units::unit<std::ratio<60>, units::unit<
std::ratio<60>, units::unit<std::ratio<1>, units::base_unit<std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<1> > > > > > >
(d=..., t=...) at ../../src/main.cpp:18
18        return d / t;
```

Debugging: mp-units

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(km(220), h(2));
```

Debugging: mp-units

```
constexpr Velocity auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

```
avg_speed(km(220), h(2));
```

```
Breakpoint 2, avg_speed<units::quantity<units::isq::si::dim_length, units::isq::si::kilometre, long>,
units::quantity<units::isq::si::dim_time, units::isq::si::hour, long> > (d=..., t=...) at ../../src/main.cpp:12
12        return d / t;
```

FRAMEWORK BASICS

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo = [](auto&& t) { /* ... */ };
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo = [](auto&& t) { /* ... */ };
```

```
template<typename T> auto foo(T&& t) { /* ... */ }
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo = [](auto&& t) { /* ... */ };
```

```
auto foo(auto&& t) { /* ... */ }
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

```
auto foo = [](auto&& t) { /* ... */ };
```

```
auto foo(auto&& t) { /* ... */ }
```

```
template<typename T> class foo { /* ... */ };
```

How do you feel about such an interface?

```
void* foo(void* t) { /* ... */ }
```

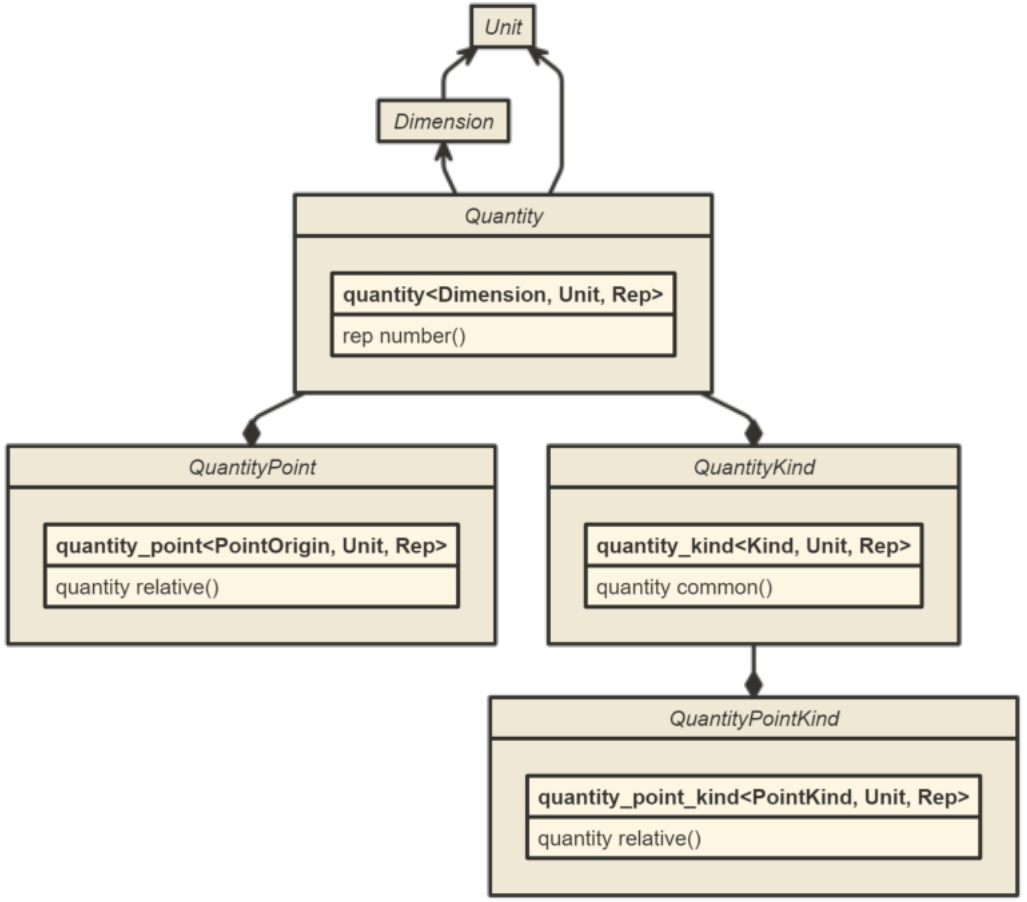
```
auto foo = [](auto&& t) { /* ... */ };
```

```
auto foo(auto&& t) { /* ... */ }
```

```
template<typename T> class foo { /* ... */ };
```

Unconstrained template parameters are the **`void*`** of C++

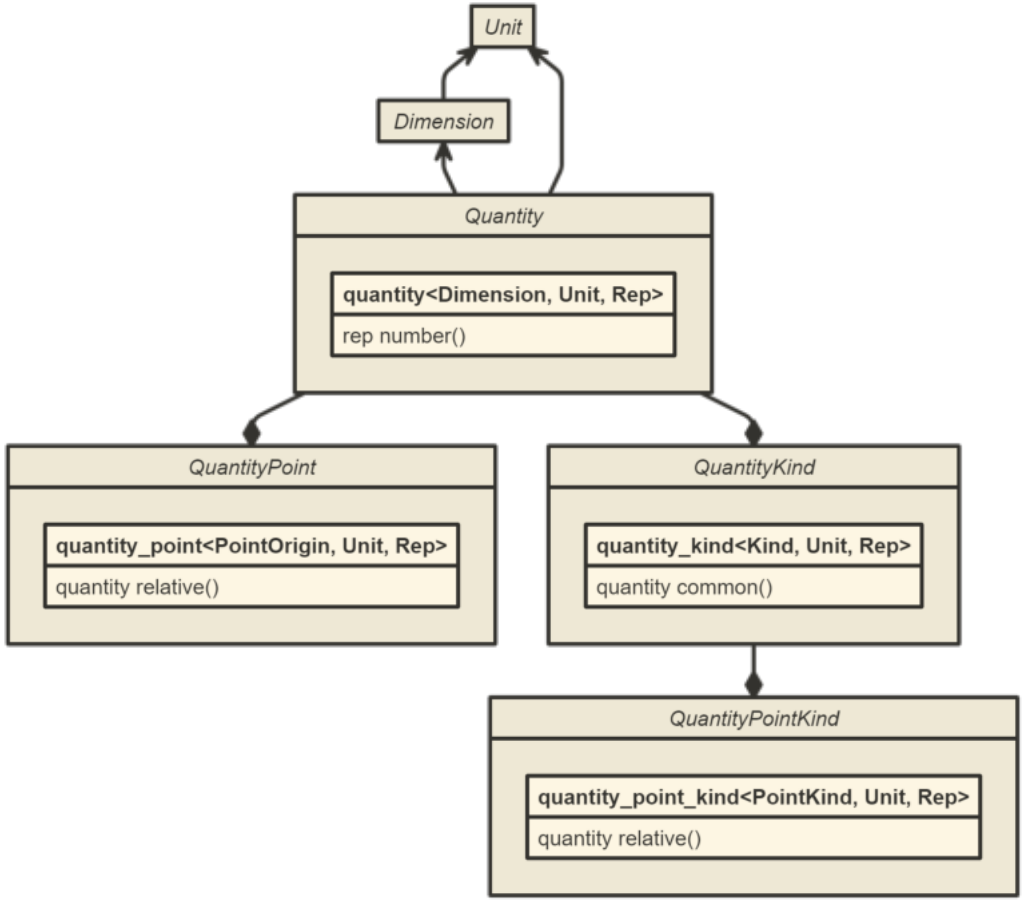
Basic Concepts



Basic Concepts

UNIT

- A basic building block of the library
- Every unit is scaled against its *reference unit*
- Every dimension has a *coherent unit* assigned

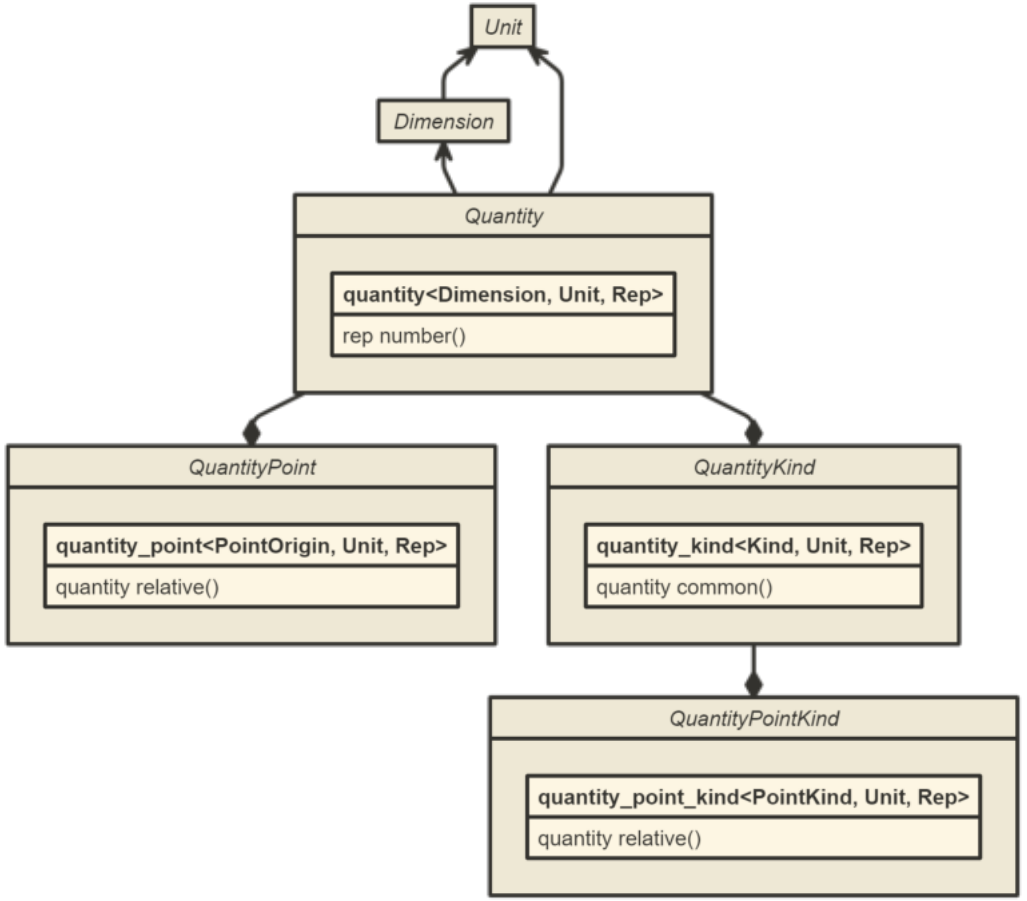


Basic Concepts

UNIT

- A basic building block of the library
- Every unit is scaled against its *reference unit*
- Every dimension has a *coherent unit* assigned

```
struct metre : named_unit<metre, "m", prefix> {};
struct kilometre : prefixed_unit<kilometre,
                                 kilo, metre> {};
```



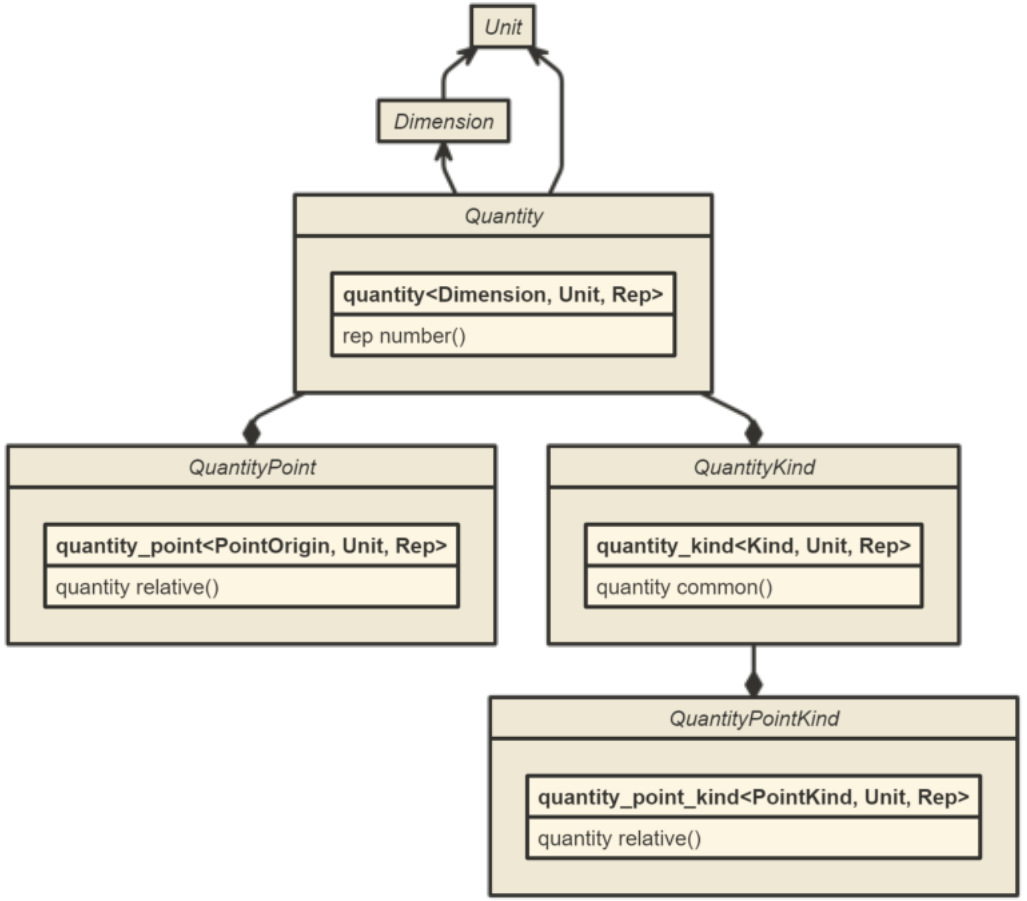
Basic Concepts

UNIT

- A basic building block of the library
- Every unit is scaled against its *reference unit*
- Every dimension has a *coherent unit* assigned

```
struct metre : named_unit<metre, "m", prefix> {};
struct kilometre : prefixed_unit<kilometre,
                                 kilo, metre> {};
```

```
struct second : named_unit<second, "s", prefix> {};
struct minute : named_scaled_unit<minute, "min",
                    no_prefix, ratio(60), second> {};
struct hour : named_scaled_unit<hour, "h",
                    no_prefix, ratio(60), minute> {};
```



Basic Concepts

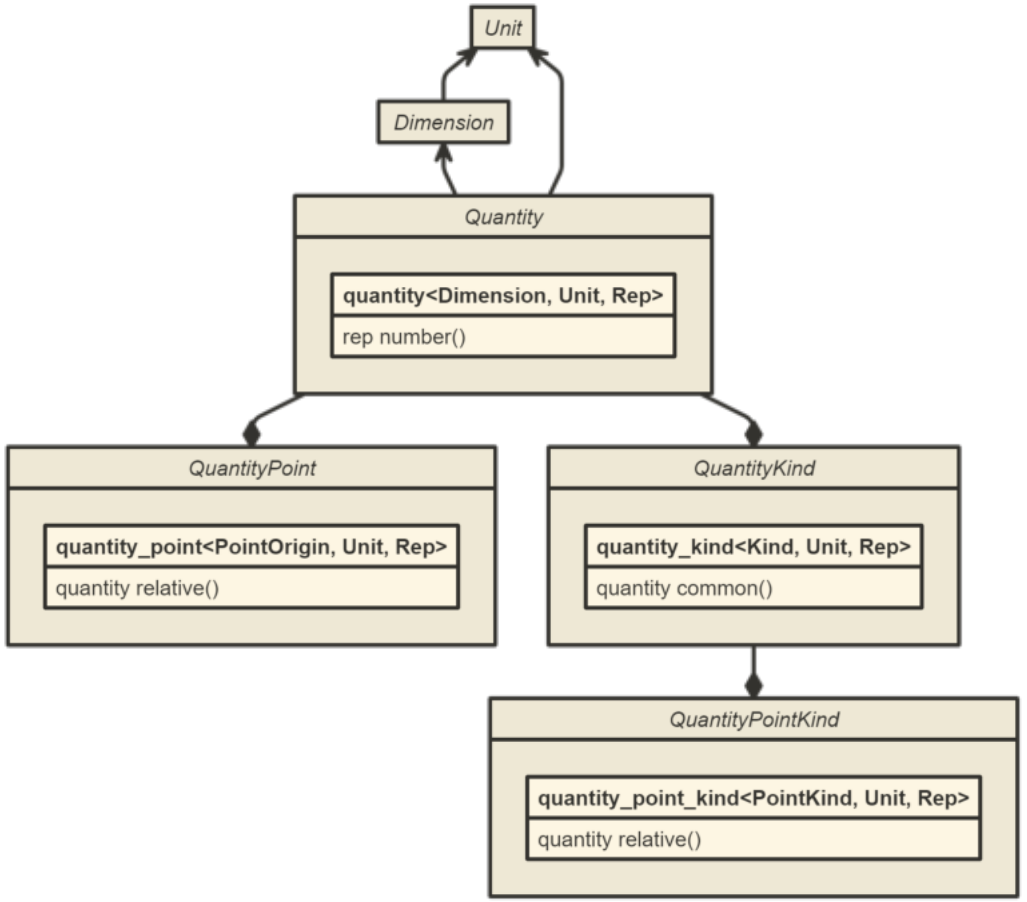
UNIT

- A basic building block of the library
- Every unit is scaled against its *reference unit*
- Every dimension has a *coherent unit* assigned

```
struct metre : named_unit<metre, "m", prefix> {};
struct kilometre : prefixed_unit<kilometre,
                                 kilo, metre> {};
```

```
struct second : named_unit<second, "s", prefix> {};
struct minute : named_scaled_unit<minute, "min",
                    no_prefix, ratio(60), second> {};
struct hour : named_scaled_unit<hour, "h",
                    no_prefix, ratio(60), minute> {};
```

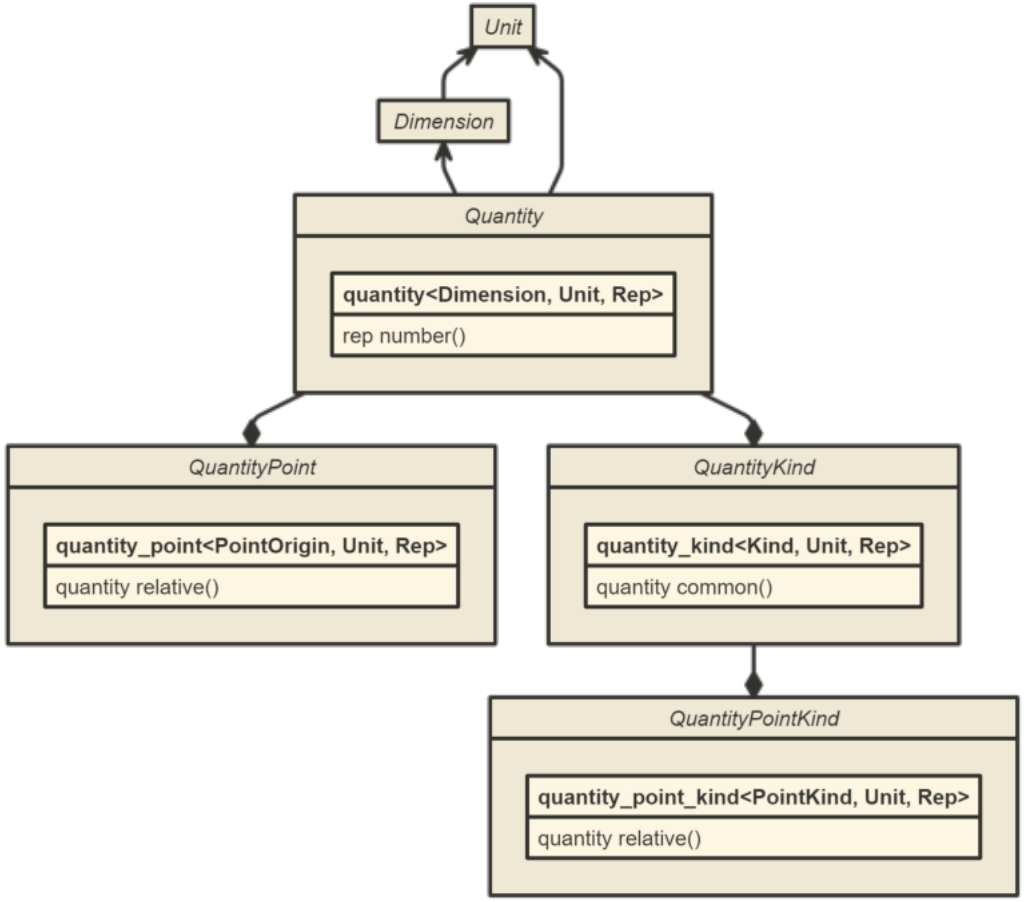
```
struct metre_per_second : unit<metre_per_second> {};
struct kilometre_per_hour : deduced_unit<
  kilometre_per_hour, dim_speed, kilometre, hour> {};
```



Basic Concepts

DIMENSION

- Matches a dimension of either a *base* or *derived* quantity

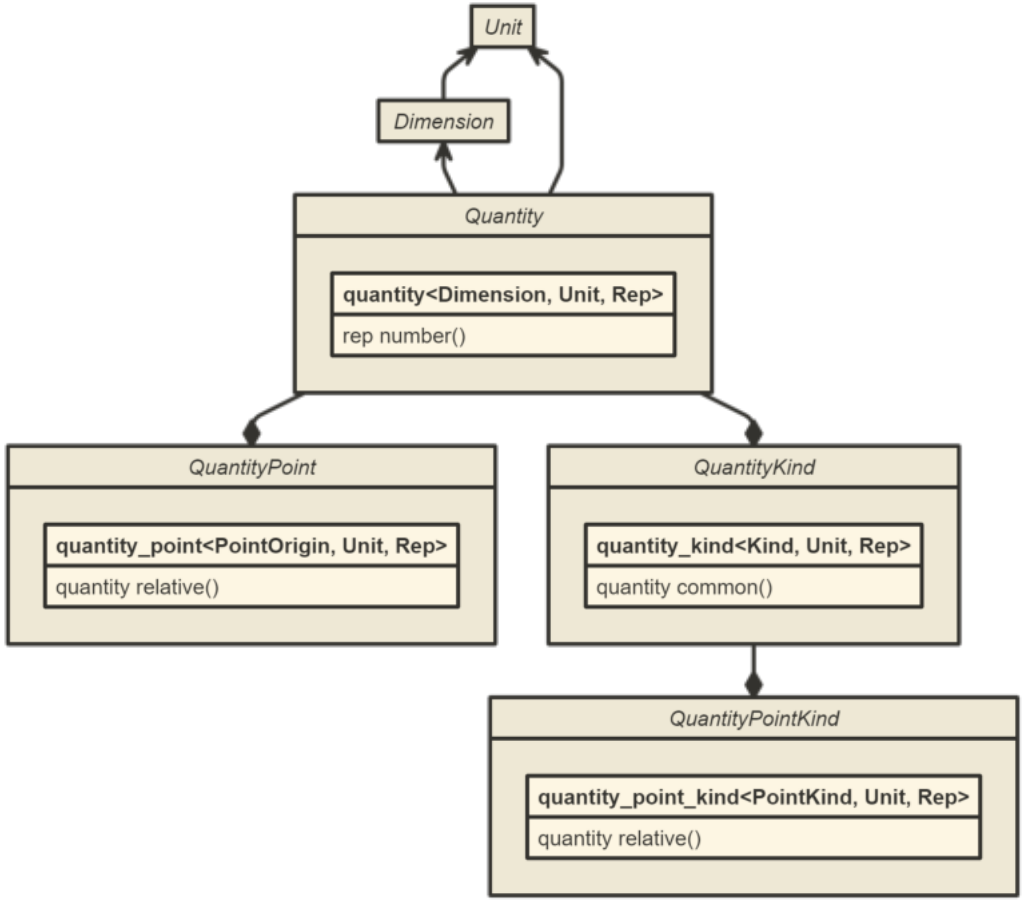


Basic Concepts

DIMENSION

- Matches a dimension of either a *base* or *derived* quantity
- **`base_dimension`** is instantiated with a *unique symbol identifier* and a *base unit*

```
struct dim_length : base_dimension<"L", metre> {};
struct dim_time : base_dimension<"T", second> {};
```



Basic Concepts

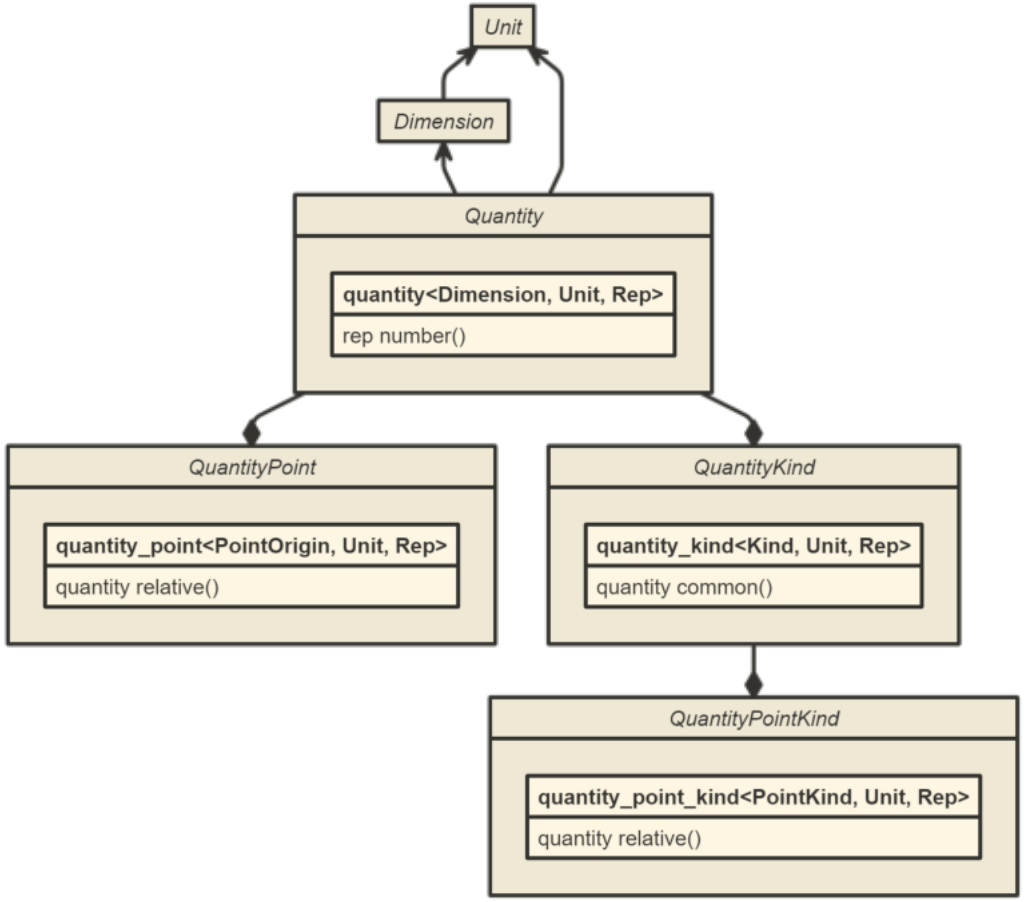
DIMENSION

- Matches a dimension of either a *base* or *derived* quantity
- **`base_dimension`** is instantiated with a *unique symbol identifier* and a *base unit*

```
struct dim_length : base_dimension<"L", metre> {};
struct dim_time : base_dimension<"T", second> {};
```

- **`derived_dimension`** is a *list of exponents* of either base or other derived dimensions

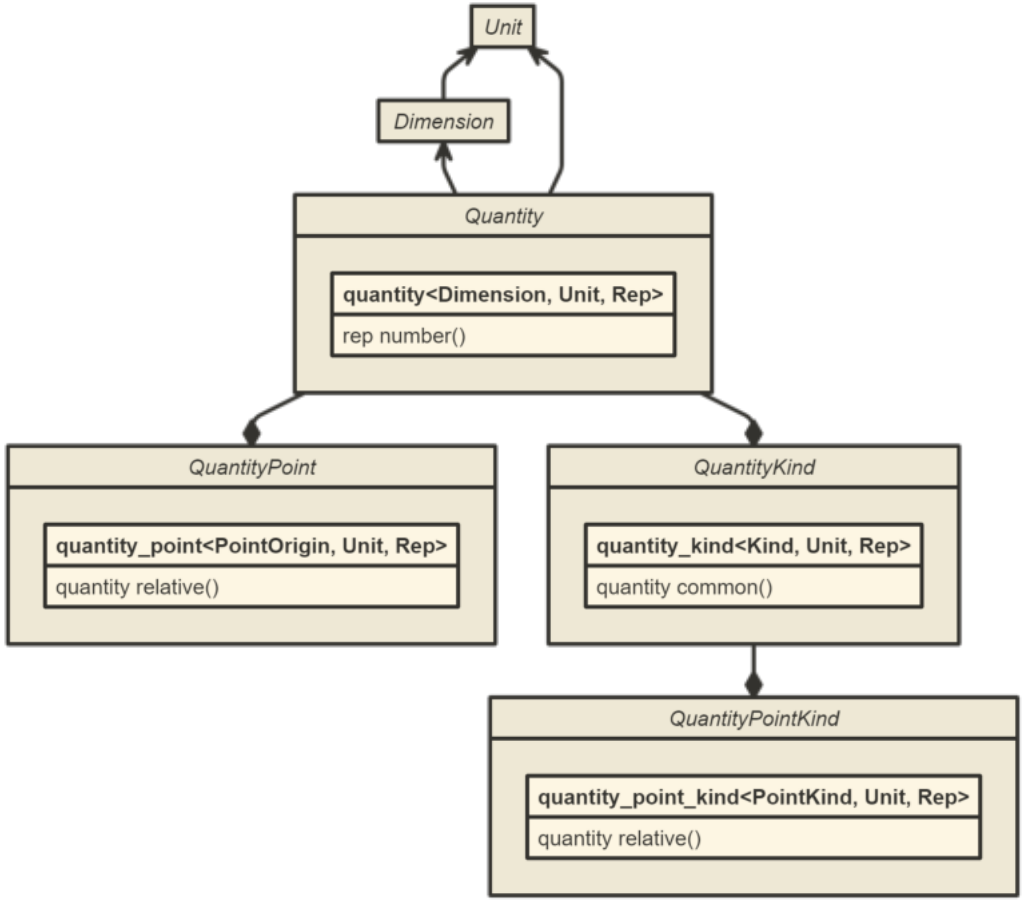
```
struct dim_speed : derived_dimension<dim_speed,
                                     metre_per_second,
                                     exponent<dim_length, 1>,
                                     exponent<dim_time, -1>> {};
```



Basic Concepts

QUANTITY

- A concrete *amount of a unit* for a *specified dimension* with a *specific representation*



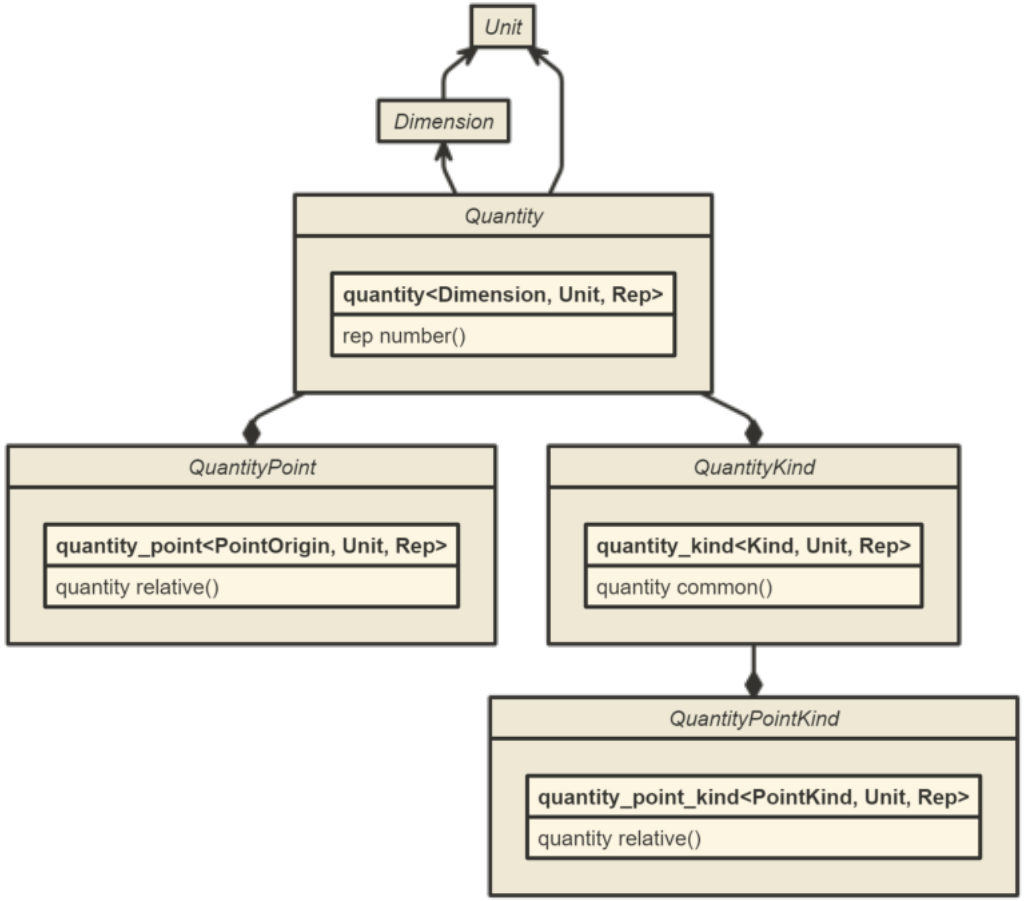
Basic Concepts

QUANTITY

- A concrete *amount of a unit* for a *specified dimension* with a *specific representation*

```
template<Unit U, Representation Rep = double>
using length = quantity<dim_length, U, Rep>;
```

```
si::length<si::kilometre, int> d(3);
```



Basic Concepts

QUANTITY

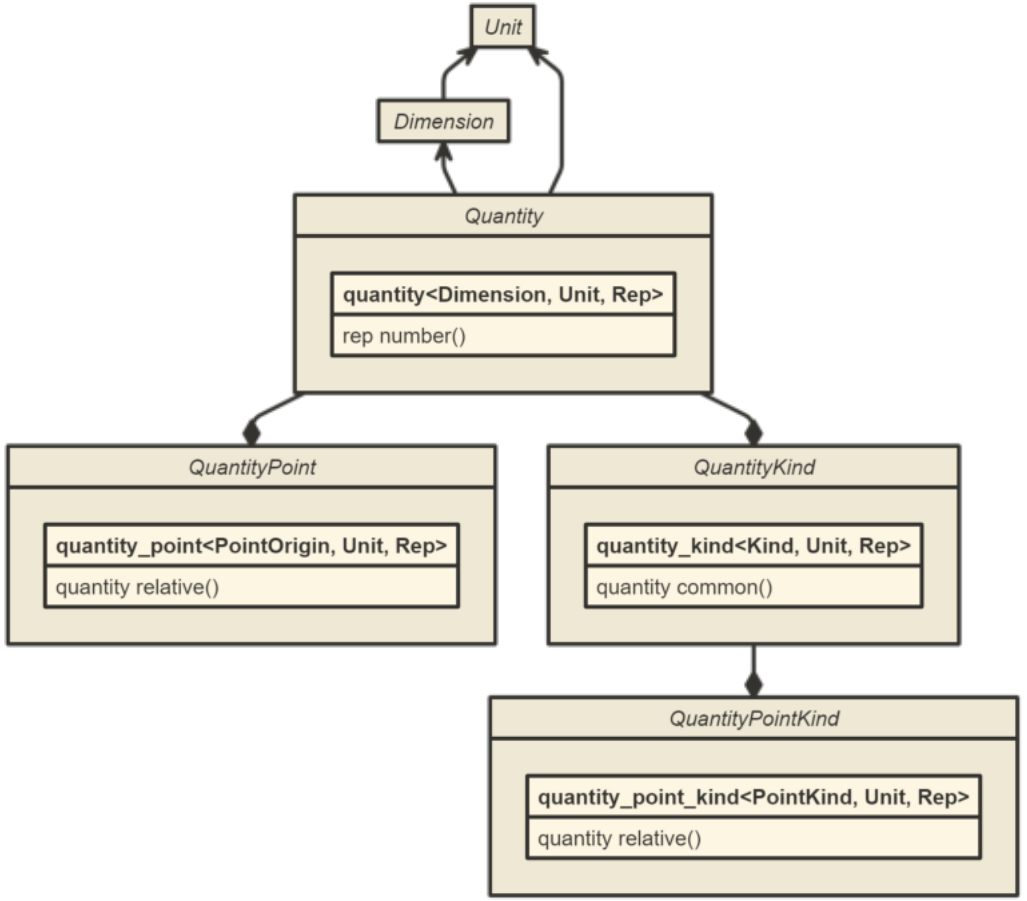
- A concrete *amount of a unit* for a *specified dimension* with a *specific representation*

```
template<Unit U, Representation Rep = double>
using length = quantity<dim_length, U, Rep>;
```

```
si::length<si::kilometre, int> d(3);
```

```
namespace units::aliases::isq::si::inline length {

template<Representation Rep = double>
using km = si::length<si::kilometre, Rep>;

}
```

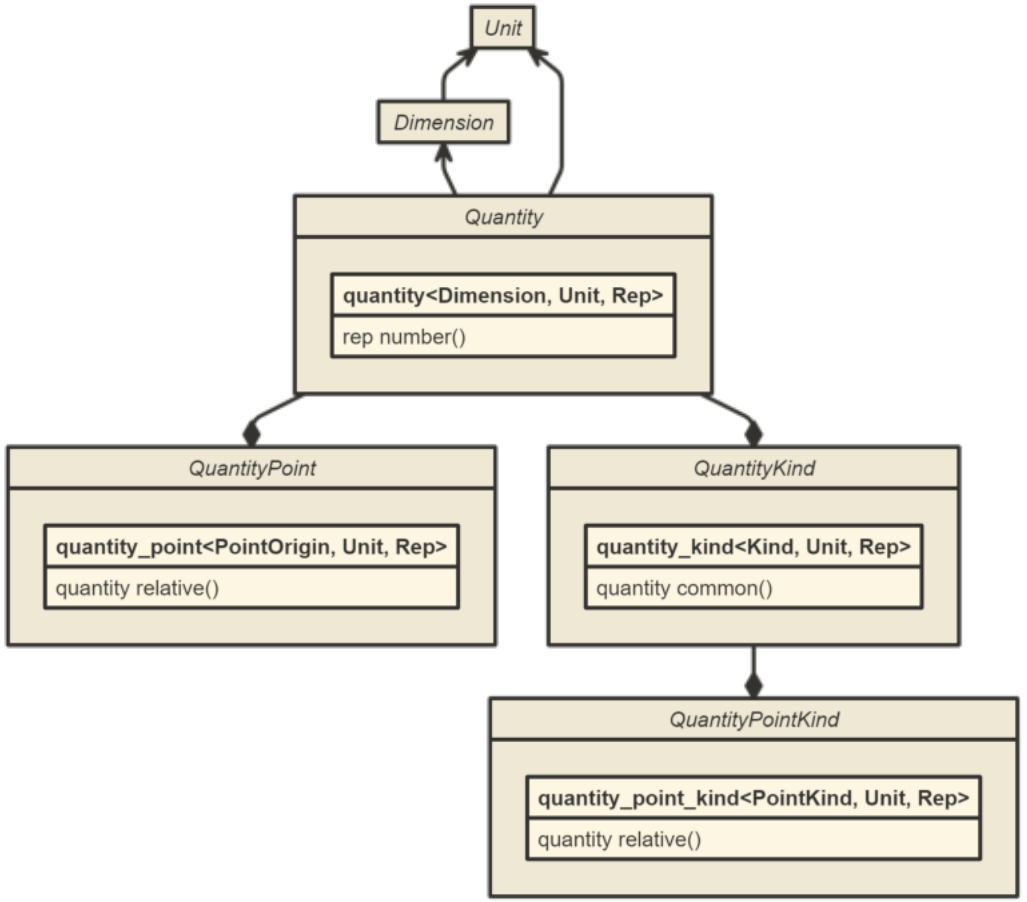
```
si::length::km d(3);
```



Basic Concepts

QUANTITY KIND

- A *quantity of more specific usage*

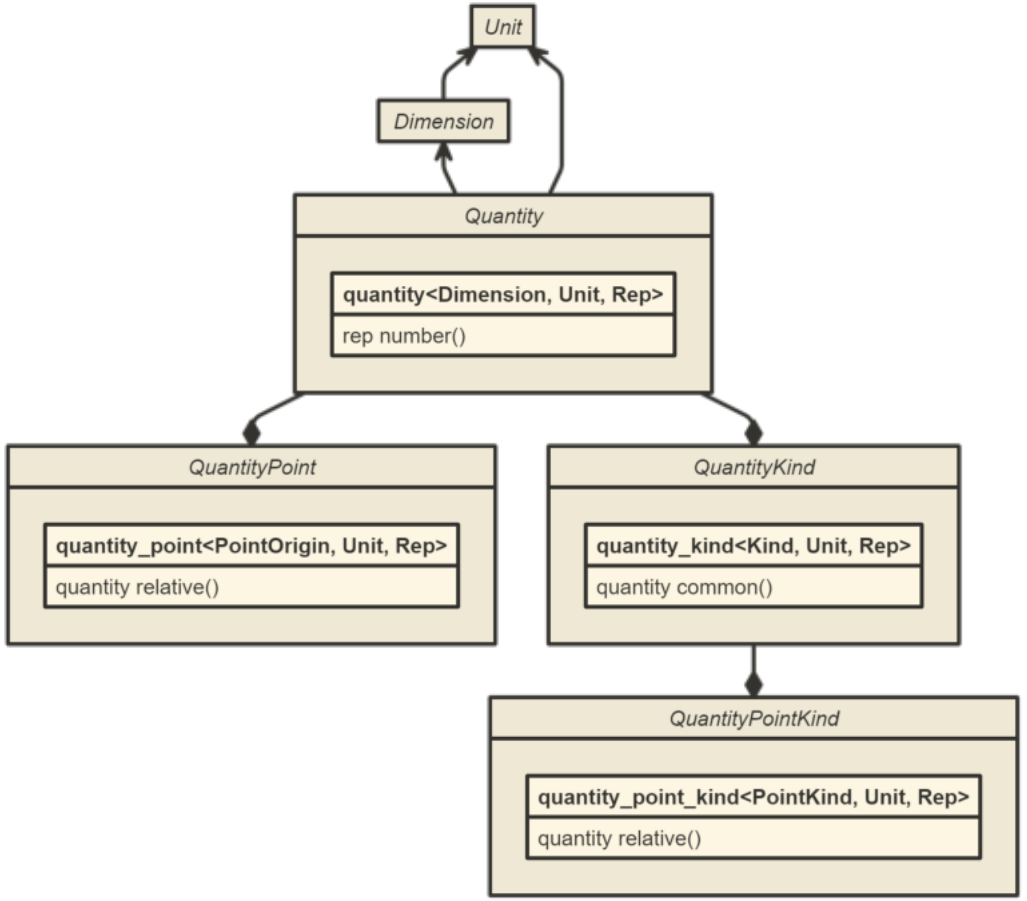


Basic Concepts

QUANTITY KIND

- A *quantity of more specific usage*

```
struct vertical_kind : kind<vertical_kind,
                            si::dim_length> {};
struct horizontal_kind : kind<horizontal_kind,
                              si::dim_length> {};
```



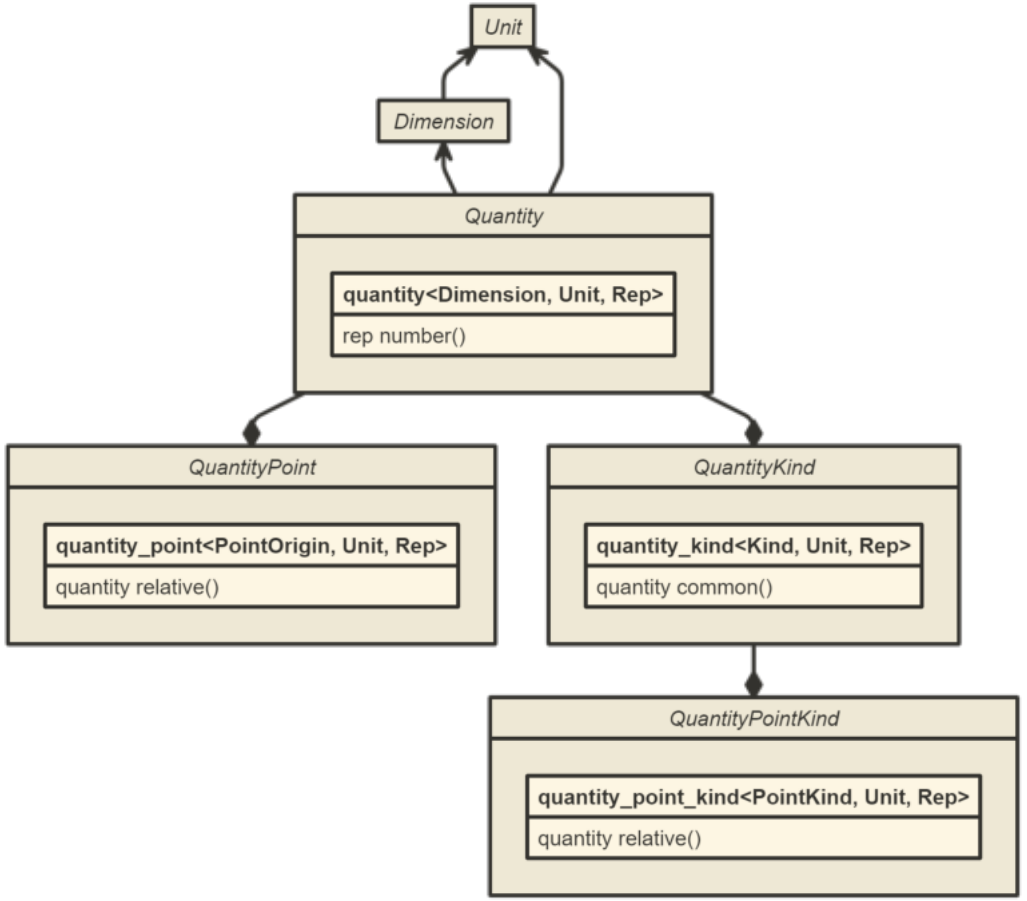
Basic Concepts

QUANTITY KIND

- A *quantity of more specific usage*

```
struct vertical_kind : kind<vertical_kind,
                            si::dim_length> {};
struct horizontal_kind : kind<horizontal_kind,
                              si::dim_length> {};
```

```
using distance = quantity_kind<horizontal_kind,
                               si::kilometre>;
using height = quantity_kind<vertical_kind,
                             si::metre>;
```



Basic Concepts

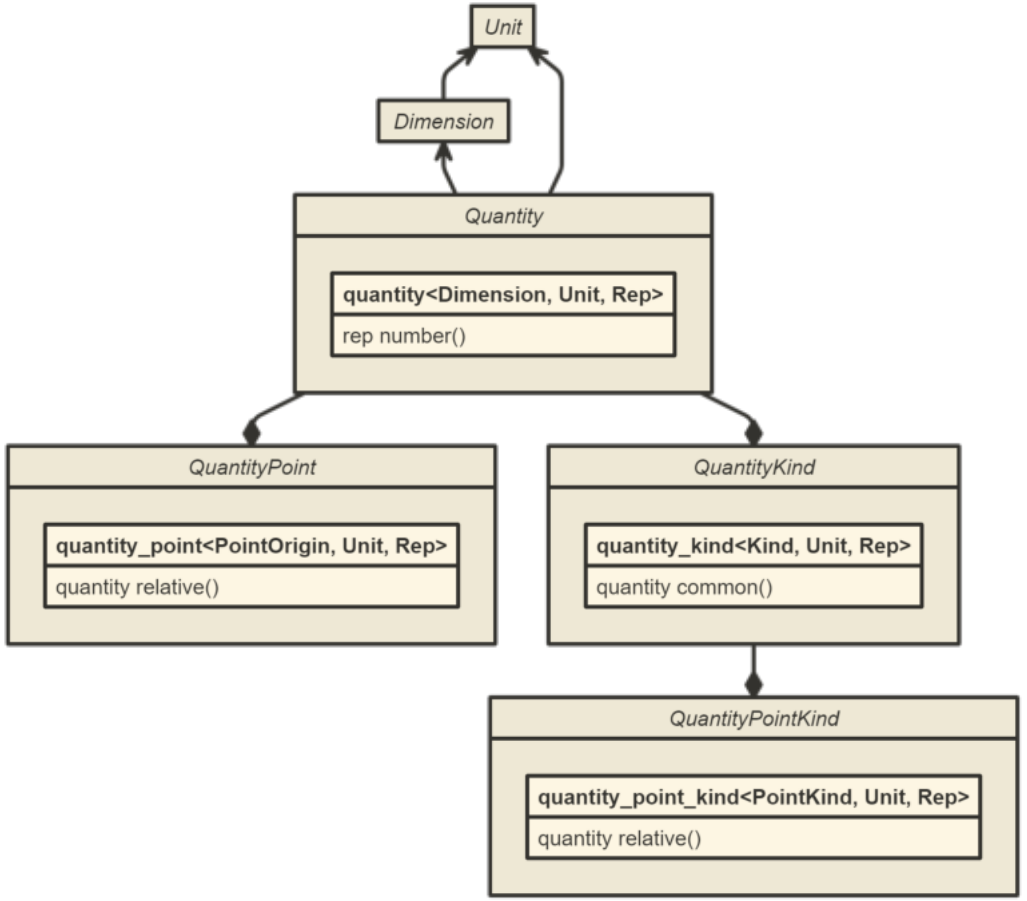
QUANTITY KIND

- A *quantity of more specific usage*

```
struct vertical_kind : kind<vertical_kind,
                            si::dim_length> {};
struct horizontal_kind : kind<horizontal_kind,
                              si::dim_length> {};
```

```
using distance = quantity_kind<horizontal_kind,
                               si::kilometre>;
using height = quantity_kind<vertical_kind,
                             si::metre>;
```

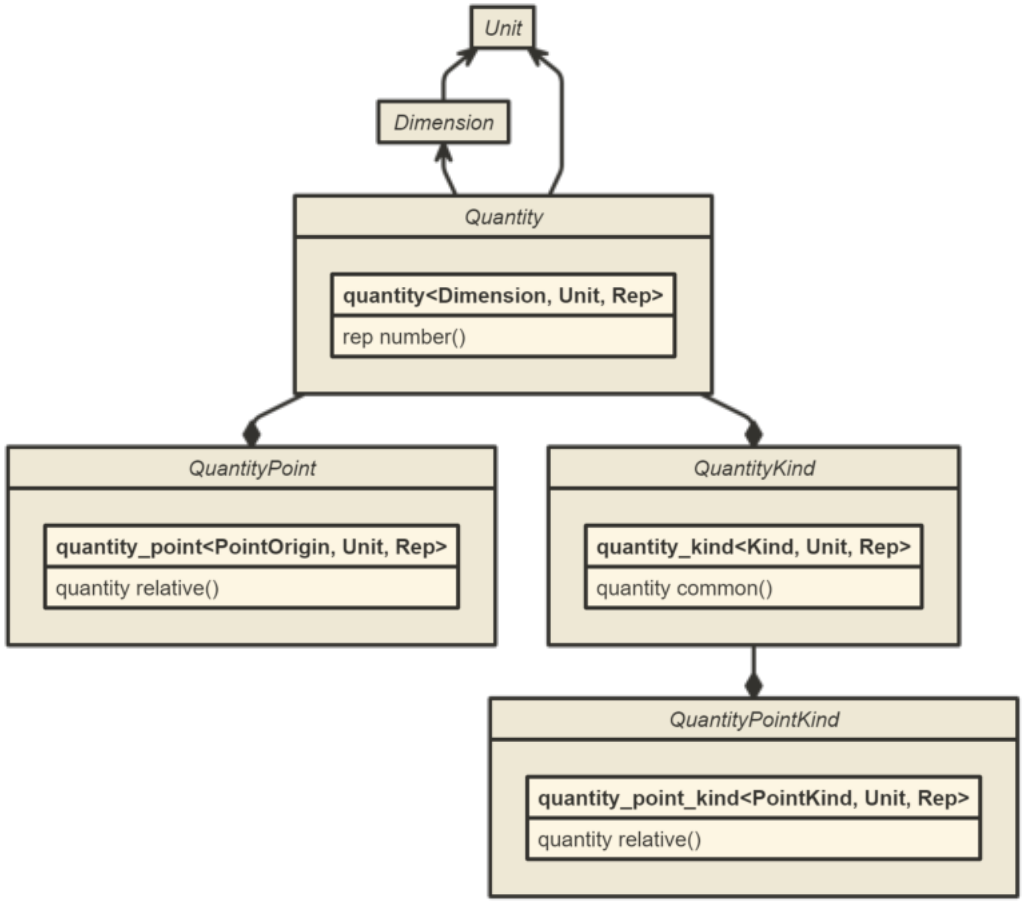
```
struct safety {
  height min_agl_height;
};
```



Basic Concepts

QUANTITY POINT

- An *absolute quantity* with respect to some *origin*

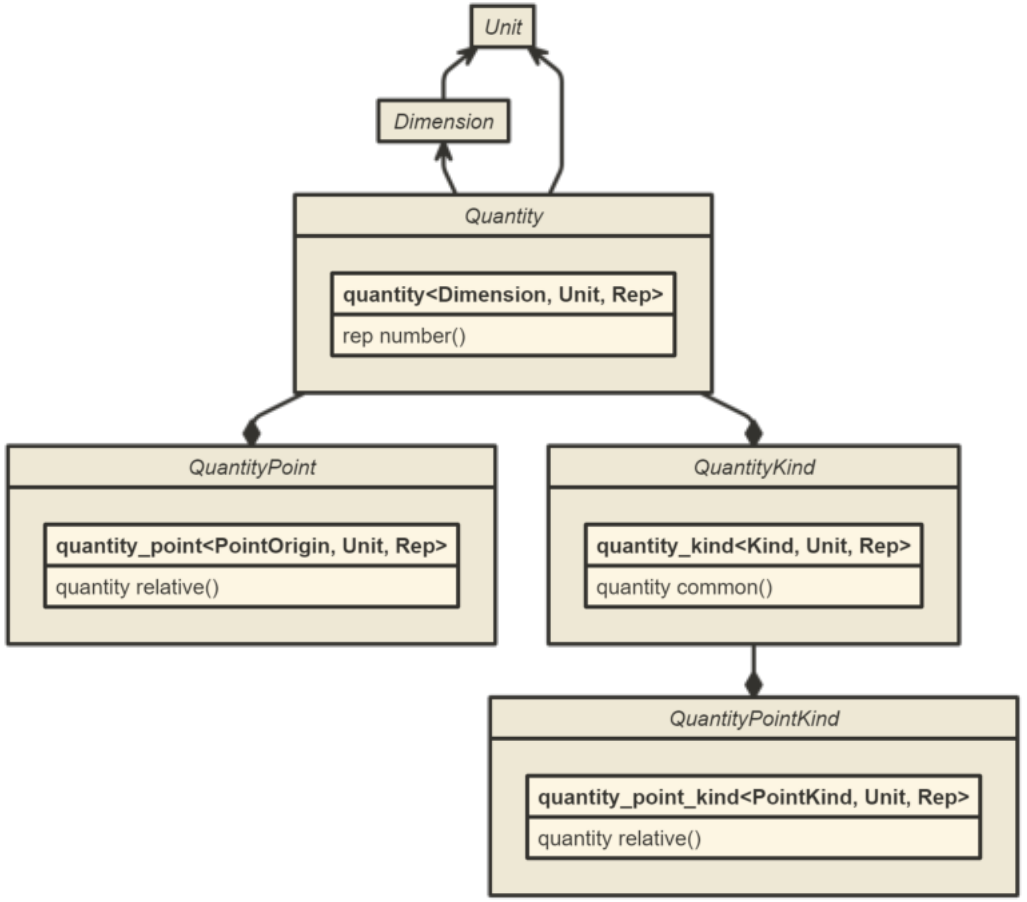


Basic Concepts

QUANTITY POINT

- An *absolute quantity* with respect to some *origin*

```
using timestamp = quantity_point<
          clock_origin<std::chrono::system_clock>,
          si::second>;
```



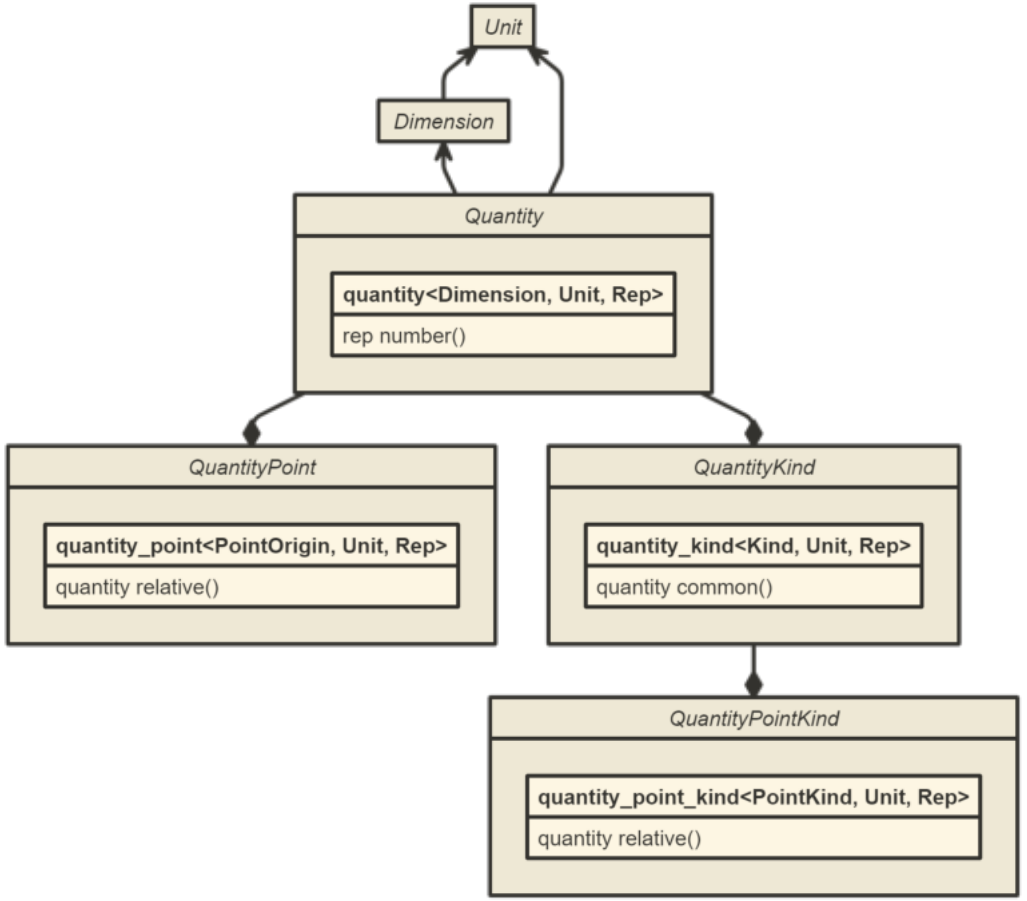
Basic Concepts

QUANTITY POINT

- An *absolute quantity* with respect to some *origin*

```
using timestamp = quantity_point<
        clock_origin<std::chrono::system_clock>,
        si::second>;
```

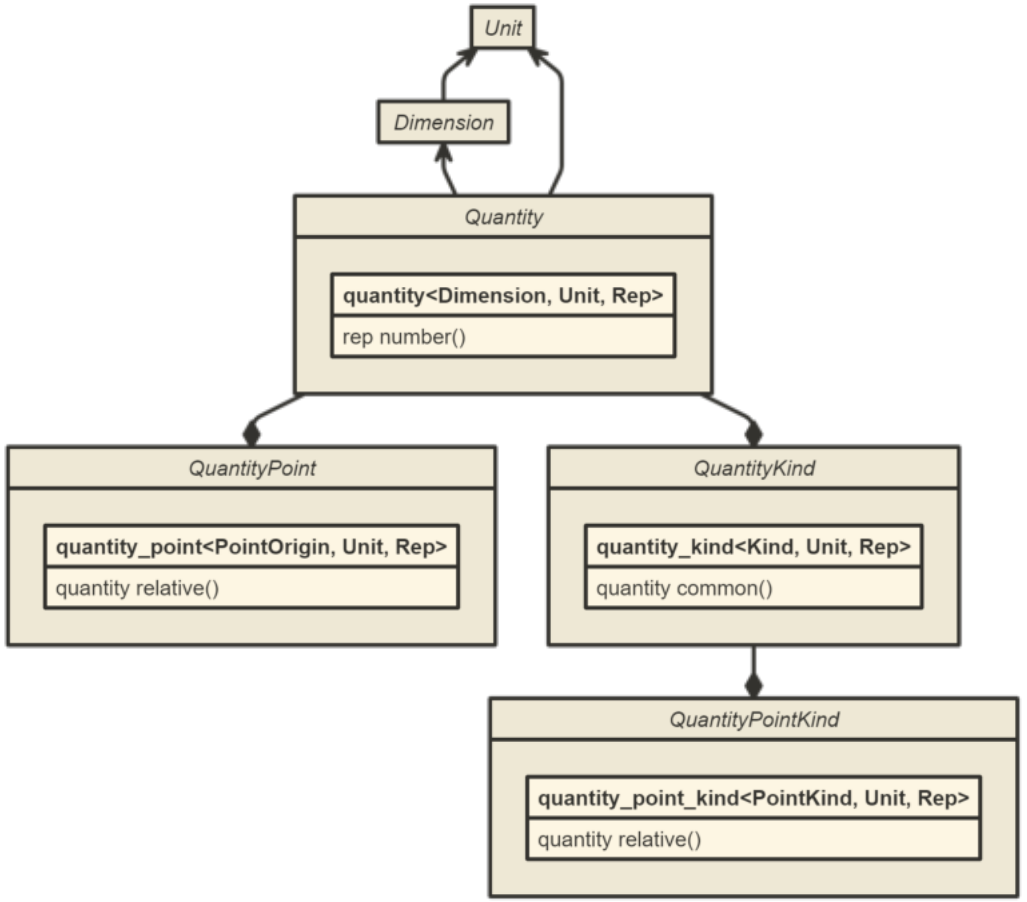
```
using namespace std::chrono;
const timestamp start_time(system_clock::now());
```



Basic Concepts

QUANTITY POINT KIND

- An *absolute quantity kind* with respect to an *origin*

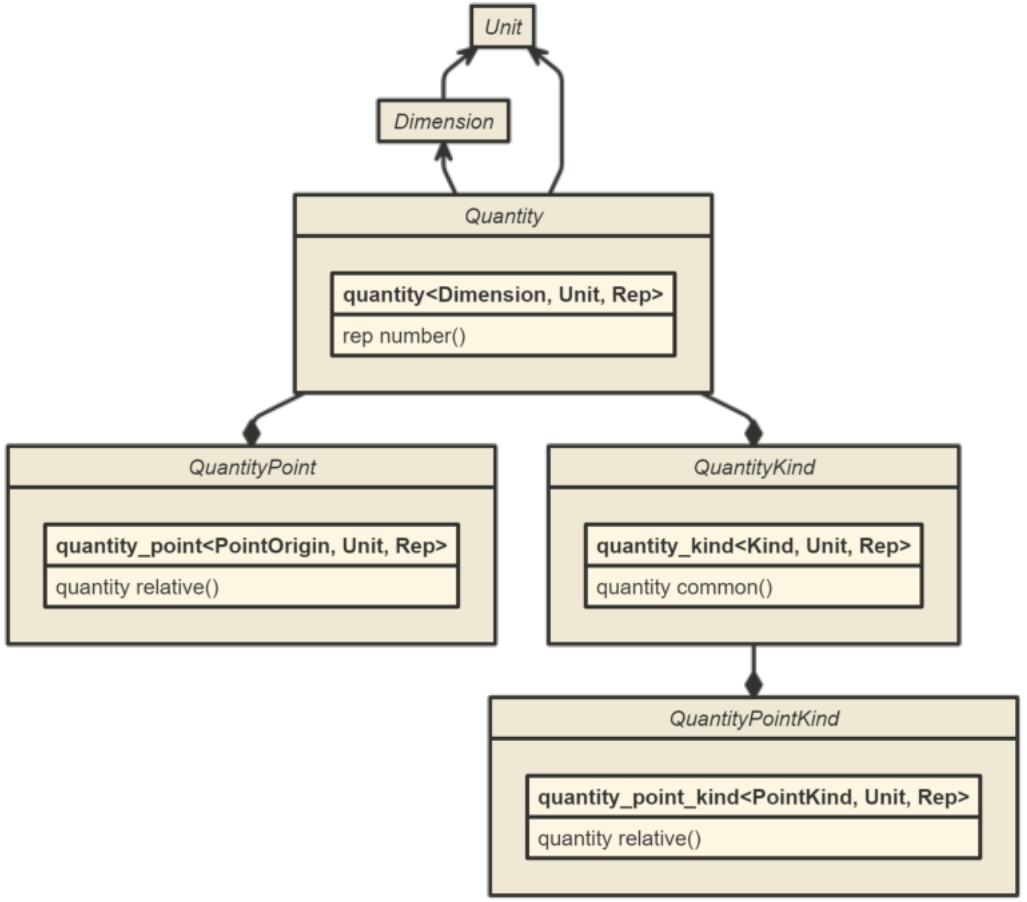


Basic Concepts

QUANTITY POINT KIND

- An *absolute quantity kind* with respect to an *origin*

```
struct vertical_point_kind :
    point_kind<vertical_point_kind, vertical_kind> {};
```



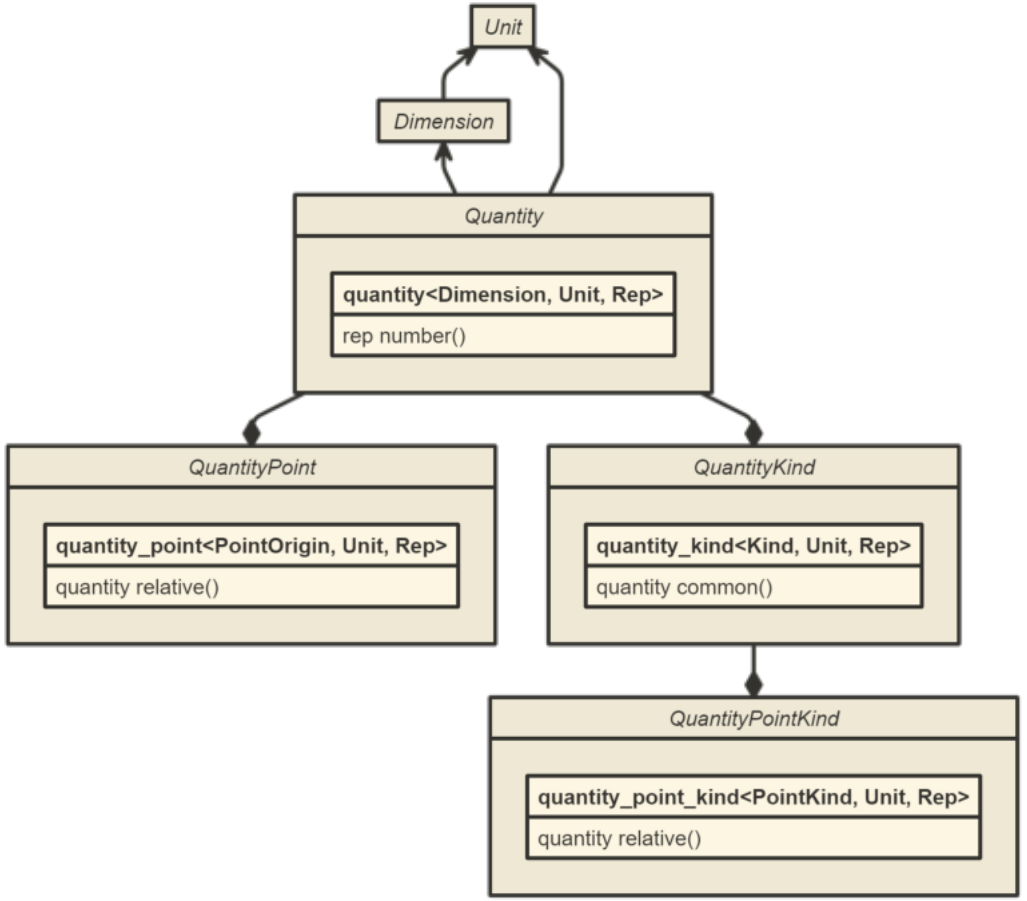
Basic Concepts

QUANTITY POINT KIND

- An *absolute quantity kind* with respect to an *origin*

```
struct vertical_point_kind :
    point_kind<vertical_point_kind, vertical_kind> {};
```

```
using altitude =
    quantity_point_kind<vertical_point_kind,
                        si::international::foot>;
```



Basic Concepts

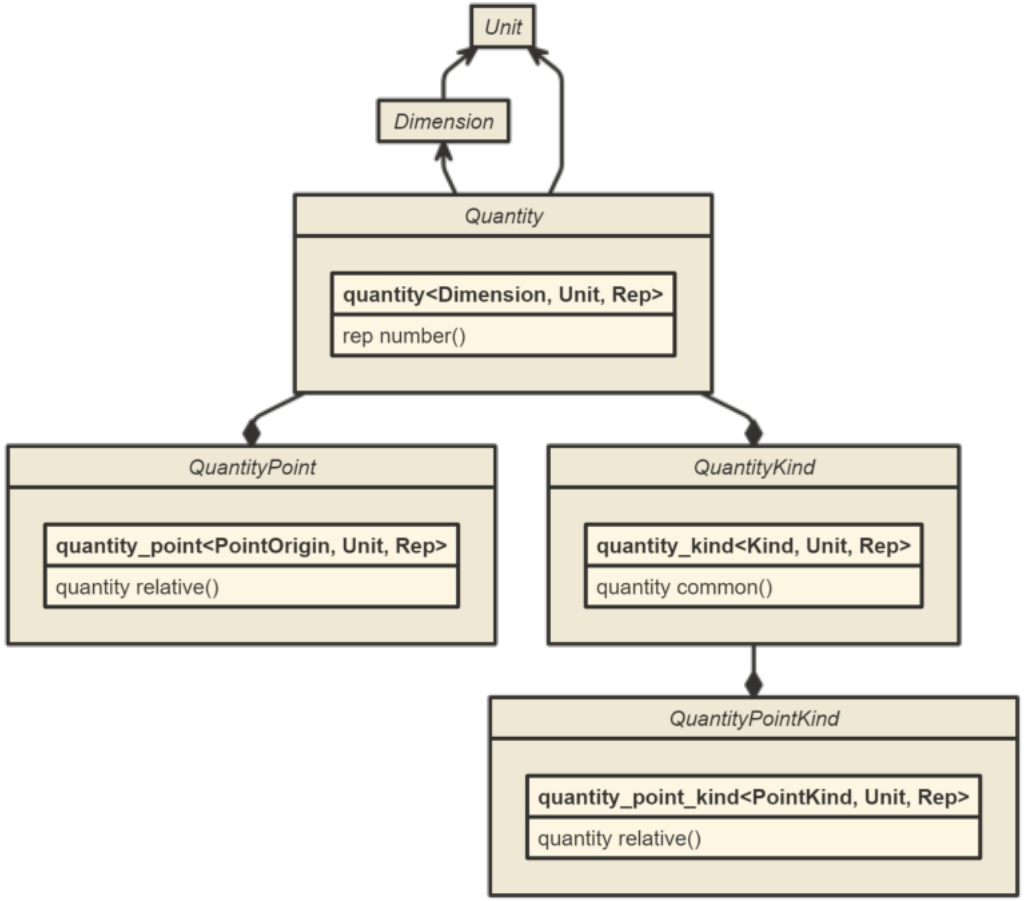
QUANTITY POINT KIND

- An *absolute quantity kind* with respect to an *origin*

```
struct vertical_point_kind :
    point_kind<vertical_point_kind, vertical_kind> {};
```

```
using altitude =
    quantity_point_kind<vertical_point_kind,
                        si::international::foot>;
```

```
struct flight_point {
  timestamp ts;
  altitude alt;
  distance dist;
};
```



Concepts example

```
template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Concepts example

```
template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent<typename T::dimension, D>;

template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Concepts example

```
template<typename T>
concept Quantity = is_specialization_of<quantity>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent<typename T::dimension, D>;

template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Concepts example

```
template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;

template<typename T>
concept Quantity = is_specialization_of<quantity>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent<typename T::dimension, D>;

template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Concepts example

```
template<typename T>
concept DerivedDimension = is_derived_from_specialization_of<T, derived_dimension_base>;

template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;

template<typename T>
concept Quantity = is_specialization_of<quantity>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent<typename T::dimension, D>;

template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Concepts example

```
template<typename T>
concept BaseDimension = is_derived_from_specialization_of_base_dimension<T>;

template<typename T>
concept DerivedDimension = is_derived_from_specialization_of<T, derived_dimension_base>;

template<typename T>
concept Dimension = BaseDimension<T> || DerivedDimension<T>;

template<typename T>
concept Quantity = is_specialization_of<quantity>;

template<typename T, typename D>
concept QuantityOf = Quantity<T> && Dimension<D> && equivalent<typename T::dimension, D>;

template<typename T>
concept Speed = QuantityOf<T, dim_speed>;
```

```
constexpr price calc_fine(Speed auto speed);
```

Explicit conversions

TO QUANTITY

```
std::cout << "Distance: " << quantity_cast<si::length<si::metre, int>>(d) << '\n';
```

Explicit conversions

TO QUANTITY

```
std::cout << "Distance: " << quantity_cast<si::length<si::metre, int>>(d) << '\n';
```

TO DIMENSION

```
std::cout << "Distance: " << quantity_cast<si::dim_length>(d) << '\n';
```

Explicit conversions

TO QUANTITY

```
std::cout << "Distance: " << quantity_cast<si::length<si::metre, int>>(d) << '\n';
```

TO DIMENSION

```
std::cout << "Distance: " << quantity_cast<si::dim_length>(d) << '\n';
```

TO UNIT

```
std::cout << "Distance: " << quantity_cast<si::metre>(d) << '\n';
```

Explicit conversions

TO QUANTITY

```
std::cout << "Distance: " << quantity_cast<si::length<si::metre, int>>(d) << '\n';
```

TO DIMENSION

```
std::cout << "Distance: " << quantity_cast<si::dim_length>(d) << '\n';
```

TO UNIT

```
std::cout << "Distance: " << quantity_cast<si::metre>(d) << '\n';
```

TO REPRESENTATION TYPE

```
std::cout << "Distance: " << quantity_cast<int>(d) << '\n';
```

Explicit conversions (Quantity)

TO QUANTITY

```
std::cout << "Distance: " << si::length<si::metre, int>(d) << '\n';
```

TO DIMENSION

```
std::cout << "Distance: " << quantity_cast<si::dim_length>(d) << '\n';
```

TO UNIT

```
std::cout << "Distance: " << quantity_cast<si::metre>(d) << '\n';
```

TO REPRESENTATION TYPE

```
std::cout << "Distance: " << quantity_cast<int>(d) << '\n';
```

Explicit conversions (Unit-Specific Aliases)

TO QUANTITY

```
std::cout << "Distance: " << si::length::m<int>(d) << '\n';
```

TO DIMENSION

```
std::cout << "Distance: " << quantity_cast<si::dim_length>(d) << '\n';
```

TO UNIT

```
std::cout << "Distance: " << quantity_cast<si::metre>(d) << '\n';
```

TO REPRESENTATION TYPE

```
std::cout << "Distance: " << quantity_cast<int>(d) << '\n';
```

Not just a syntactic sugar

Not just a syntactic sugar

- Constraining *function template return types*

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

Not just a syntactic sugar

- Constraining *function template return types*

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

- Constraining the *deduced types* of user's variables

```
const Speed auto speed = avg_speed(km(220), h(2));
```

Not just a syntactic sugar

- Constraining *function template return types*

```
constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  return d / t;
}
```

- Constraining the *deduced types* of user's variables

```
const Speed auto speed = avg_speed(km(220), h(2));
```

- Constraining *class template parameters* without introducing additional parameters

```
template<typename Q, direction D>
  requires Quantity<Q> || QuantityPoint<Q>
class vector;
```

```
template<Dimension D, ratio R>
  requires UnitRatio<R>
struct unit;
```

Benefits of using C++ Concepts

Benefits of using C++ Concepts

1. Clearly state the design intent of the interface of a class/function template

Benefits of using C++ Concepts

1. Clearly **state the design intent** of the interface of a class/function template
2. **Embedded in a template signature**

Benefits of using C++ Concepts

1. Clearly **state the design intent** of the interface of a class/function template
2. **Embedded in a template signature**
3. **Simplify and extend SFINAE**
   - *disabling specific overloads* for specific constraints (compared to **`std::enable_if`**)
   - constraints *based on dependent member types/functions existence* (compared to **`std::void_t`**)
   - *no dummy template parameters* allocated for SFINAE needs
   - constraining *function return types and deduced types of user's variables*

Benefits of using C++ Concepts

1. Clearly **state the design intent** of the interface of a class/function template
2. **Embedded in a template signature**
3. **Simplify and extend SFINAE**
   - *disabling specific overloads* for specific constraints (compared to **`std::enable_if`**)
   - constraints *based on dependent member types/functions existence* (compared to **`std::void_t`**)
   - *no dummy template parameters* allocated for SFINAE needs
   - constraining *function return types and deduced types of user's variables*
4. Greatly **improve error messages**
   - raise *compilation error* about failed compile-time contract *before instantiating a template*
   - no more errors from *deeply nested implementation details of a function template*

Pre-C++20 unit definition (Nic Holthaus)

```
UNIT_ADD_WITH_METRIC_PREFIXES(length, meter, meters, m, unit<std::ratio<1>, units::category::length_unit>)
```

Pre-C++20 unit definition (Nic Holthaus)

```
UNIT_ADD_WITH_METRIC_PREFIXES(length, meter, meters, m, unit<std::ratio<1>, units::category::length_unit>)
```

```
#define UNIT_ADD(namespaceName, nameSingular, namePlural, abbreviation, /*definition*/...)\
    UNIT_ADD_UNIT_TAGS(namespaceName,nameSingular, namePlural, abbreviation, __VA_ARGS__)\
    UNIT_ADD_UNIT_DEFINITION(namespaceName,nameSingular)\
    UNIT_ADD_NAME(namespaceName,nameSingular, abbreviation)\
    UNIT_ADD_IO(namespaceName,nameSingular, abbreviation)\
    UNIT_ADD_LITERALS(namespaceName,nameSingular, abbreviation)
```

Pre-C++20 unit definition (Nic Holthaus)

```
UNIT_ADD_WITH_METRIC_PREFIXES(length, meter, meters, m, unit<std::ratio<1>, units::category::length_unit>)
```

```
#define UNIT_ADD(namespaceName, nameSingular, namePlural, abbreviation, /*definition*/...)\
    UNIT_ADD_UNIT_TAGS(namespaceName,nameSingular, namePlural, abbreviation, __VA_ARGS__)\
    UNIT_ADD_UNIT_DEFINITION(namespaceName,nameSingular)\
    UNIT_ADD_NAME(namespaceName,nameSingular, abbreviation)\
    UNIT_ADD_IO(namespaceName,nameSingular, abbreviation)\
    UNIT_ADD_LITERALS(namespaceName,nameSingular, abbreviation)
```

```
#define UNIT_ADD_NAME(namespaceName, nameSingular, abbrev)\
template<> inline constexpr const char* name(const namespaceName::nameSingular ## _t&)\
{\
    return #nameSingular;\
}\
template<> inline constexpr const char* abbreviation(const namespaceName::nameSingular ## _t&)\
{\
    return #abbrev;\
}
```

Class Types as Non-Type Template Parameters

```
struct second : named_unit<second, "s", prefix> {};
struct minute : named_scaled_unit<minute, "min", no_prefix, ratio(60), second> {};
```

Class Types as Non-Type Template Parameters

```
struct second : named_unit<second, "s", prefix> {};
struct minute : named_scaled_unit<minute, "min", no_prefix, ratio(60), second> {};
```

```
template<typename Child, basic_symbol_text Symbol, PrefixFamily PF>
struct named_unit : downcast_child<Child, scaled_unit<ratio(1), Child>> {
  static constexpr bool is_named = true;
  static constexpr auto symbol = Symbol;
  using prefix_family = PF;
};
```

Class Types as Non-Type Template Parameters

```
struct second : named_unit<second, "s", prefix> {};
struct minute : named_scaled_unit<minute, "min", no_prefix, ratio(60), second> {};
```

```
template<typename Child, basic_symbol_text Symbol, PrefixFamily PF>
struct named_unit : downcast_child<Child, scaled_unit<ratio(1), Child>> {
  static constexpr bool is_named = true;
  static constexpr auto symbol = Symbol;
  using prefix_family = PF;
};
```

```
template<typename Child, Prefix P, Unit U>
  requires U::is_named && std::same_as<typename P::prefix_family, typename U::prefix_family>
struct prefixed_unit : downcast_child<Child, scaled_unit<P::ratio * U::ratio,
                                                         typename U::reference>> {
  static constexpr bool is_named = true;
  static constexpr auto symbol = P::symbol + U::symbol;
  using prefix_family = no_prefix;
};
```

NTTP in action

BEFORE

```
template<typename ExpList>
struct base_units_ratio;

template<typename E>
struct base_units_ratio<exp_list<E>> {
  using type = exp_ratio<E>::type;
};

template<typename E, typename... Es>
struct base_units_ratio<exp_list<E, Es...>> {
  using type = ratio_multiply<typename exp_ratio<E>::type, typename base_units_ratio<exp_list<Es...>>::type>;
};
```

NTTP in action

BEFORE

```
template<typename ExpList>
struct base_units_ratio;

template<typename E>
struct base_units_ratio<exp_list<E>> {
  using type = exp_ratio<E>::type;
};

template<typename E, typename... Es>
struct base_units_ratio<exp_list<E, Es...>> {
  using type = ratio_multiply<typename exp_ratio<E>::type, typename base_units_ratio<exp_list<Es...>>::type>;
};
```

AFTER

```
template<typename... Es>
constexpr ratio base_units_ratio(exp_list<Es...>)
{
  return (exp_ratio<Es>() * ...);
}
```

Class Types as Non-Type Template Parameters

Usage of class types as non-type template parameters (NTTP) might be *one of the most significant C++ improvements in template metaprogramming* during the last decade

Class Types as Non-Type Template Parameters

Usage of class types as non-type template parameters (NTTP) might be *one of the most significant C++ improvements in template metaprogramming* during the last decade

If a template parameter behaves like a value it probably should be an NTTP.

Text Output (godbolt.org/z/snfhK31j5)

OUTPUT STREAMS

```
using namespace units::aliases::isq::si;
constexpr Speed auto v1 = avg_speed(km(220), h(2));
constexpr Speed auto v2 = avg_speed(mi(140), h(2));
std::cout << v1 << '\n';  // 110 km/h
std::cout << v2 << '\n';  // 70 mi/h
```

Text Output (godbolt.org/z/snfhK31j5)

OUTPUT STREAMS

```
using namespace units::aliases::isq::si;
constexpr Speed auto v1 = avg_speed(km(220), h(2));
constexpr Speed auto v2 = avg_speed(mi(140), h(2));
std::cout << v1 << '\n';  // 110 km/h
std::cout << v2 << '\n';  // 70 mi/h
```

std::format

```
std::cout << std::format("{}", km(123)) << '\n';        // 123 km
std::cout << std::format("{:%Q}", km(123)) << '\n';     // 123
std::cout << std::format("{:%q}", km(123)) << '\n';     // km
std::cout << std::format("{:%Q%q}", km(123)) << '\n';   // 123km
```

{fmt} Grammar

```
units-format-spec   ::=  [fill-and-align] [width] [units-specs]
units-specs         ::=  conversion-spec
                         units-specs conversion-spec
                         units-specs literal-char
literal-char        ::=  any character other than '{' or '}'
conversion-spec     ::=  '%' units-type
units-type          ::=  [units-rep-modifier] 'Q'
                         [units-unit-modifier] 'q'
                         one of "nt%"
units-rep-modifier  ::=  [sign] [#] [precision] [L] [units-rep-type]
units-rep-type      ::=  one of "aAbBdeEfFgGoxX"
units-unit-modifier ::=  'A'
```

Unicode and ASCII-only (godbolt.org/z/4sc3PdExK)

```
std::cout << std::format("{}", resistance::R(10)) << '\n';           // 10 Ω
std::cout << std::format("{:%Q %Aq}", R(10)) << '\n';                // 10 ohm
std::cout << std::format("{}", time::us(125)) << '\n';               // 125 µs
std::cout << std::format("{:%Q %Aq}", us(125)) << '\n';              // 125 us
std::cout << std::format("{}", acceleration::m_per_s2(9.8)) << '\n'; // 9.8 m/s²
std::cout << std::format("{:%Q %Aq}", m_per_s2(9.8)) << '\n';        // 9.8 m/s^2
```

- Unicode output by default
- ASCII-only output with **A** modifier

Linear Algebra vs. Quantities (P1385)

LINEAR ALGEBRA OF QUANTITIES

```
std::math::fs_vector<length::m<>, 3> v = { m(1), m(2), m(3) };
std::math::fs_vector<length::m<>, 3> u = { m(3), m(2), m(1) };
std::math::fs_vector<length::km<>, 3> t = { km(3), km(2), km(1) };
```

Linear Algebra vs. Quantities (P1385)

LINEAR ALGEBRA OF QUANTITIES

```
std::math::fs_vector<length::m<>, 3> v = { m(1), m(2), m(3) };
std::math::fs_vector<length::m<>, 3> u = { m(3), m(2), m(1) };
std::math::fs_vector<length::km<>, 3> t = { km(3), km(2), km(1) };
```

```
std::cout << "v + u     = " << v + u << "\n";
std::cout << "v + t     = " << v + t << "\n";
std::cout << "t_in_m    = " << std::math::fs_vector<length::m<>, 3>(t) << "\n";
std::cout << "v * u     = " << v * u << "\n";
std::cout << "2_q_m * v = " << 2_q_m * v << "\n";
```

Linear Algebra vs. Quantities (P1385)

LINEAR ALGEBRA OF QUANTITIES

```
std::math::fs_vector<length::m<>, 3> v = { m(1), m(2), m(3) };
std::math::fs_vector<length::m<>, 3> u = { m(3), m(2), m(1) };
std::math::fs_vector<length::km<>, 3> t = { km(3), km(2), km(1) };
```

```
std::cout << "v + u     = " << v + u << "\n";
std::cout << "v + t     = " << v + t << "\n";
std::cout << "t_in_m    = " << std::math::fs_vector<length::m<>, 3>(t) << "\n";
std::cout << "v * u     = " << v * u << "\n";
std::cout << "2_q_m * v = " << 2_q_m * v << "\n";
```

```
v + u     = |       4 m       4 m       4 m |
v + t     = |    3001 m    2002 m    1003 m |
t_in_m    = |    3000 m    2000 m    1000 m |
v * u     = 10 m²
2_q_m * v = |     2 m²     4 m²     6 m² |
```

Linear Algebra vs. Quantities (P1385)

QUANTITIES OF LINEAR ALGEBRA TYPES

```
length::m v(std::math::fs_vector<int, 3>{ 1, 2, 3 });
length::m u(std::math::fs_vector<int, 3>{ 3, 2, 1 });
length::km t(std::math::fs_vector<int, 3>{ 3, 2, 1 });
```

Linear Algebra vs. Quantities (P1385)

QUANTITIES OF LINEAR ALGEBRA TYPES

```
length::m v(std::math::fs_vector<int, 3>{ 1, 2, 3 });
length::m u(std::math::fs_vector<int, 3>{ 3, 2, 1 });
length::km t(std::math::fs_vector<int, 3>{ 3, 2, 1 });
```

```
std::cout << "v + u     = " << v + u << "\n";
std::cout << "v + t     = " << v + t << "\n";
std::cout << "t_in_m    = " << quantity_cast<si::metre>(t) << "\n";
std::cout << "v * u     = " << v * u << "\n";
std::cout << "2_q_m * v = " << 2_q_m * v << "\n";
```

```
v + u     = |         4         4         4 | m
v + t     = |      3001      2002      1003 | m
t_in_m    = |      3000      2000      1000 | m
v * u     = 10 m²
2_q_m * v = |         2         4         6 | m²
```

ENVIRONMENT, COMPATIBILITY, NEXT STEPS

C++20 in the library

LANGUAGE

- Concept
- Class NTTPs
- Consistent and Defaulted Comparison
- **`explicit(bool)`**
- Down with **`typename`**
- Lambdas in unevaluated contexts
- Immediate functions (**`consteval`**) (TBD)
- Modules (TBD)

LIBRARY

- **`constexpr`** algorithms
- Concepts Library
- **`{fmt}`**

Compilers support

- gcc-10
- clang-12
- Visual Studio 16.9

Plans for C++ Standardization

- ISO C++ Committee dedicated *many hours* for this subject already

Plans for C++ Standardization

- ISO C++ Committee dedicated *many hours* for this subject already
- A lot of *interest in the industry*

Plans for C++ Standardization

- ISO C++ Committee dedicated *many hours* for this subject already
- A lot of *interest in the industry*
- Really *positive feedback* so far

Plans for C++ Standardization

- ISO C++ Committee dedicated *many hours* for this subject already
- A lot of *interest in the industry*
- Really *positive feedback* so far
- **Hopefully C++26**
  - we need more field experience and feedback
  - C++26 train arrives soon
  - COVID-19 does not help

Plans for C++ Standardization

- ISO C++ Committee dedicated *many hours* for this subject already
- A lot of *interest in the industry*
- Really *positive feedback* so far
- **Hopefully C++26**
  - we need more field experience and feedback
  - C++26 train arrives soon
  - COVID-19 does not help

We want to ensure that the library is ready before we start the process.

Please try and tell us about your experience or requirements

COMPANIES

- Production/POC use
- Requirements

Please try and tell us about your experience or requirements

COMPANIES

- Production/POC use
- Requirements

AUTHORS OF OTHER LIBRARIES

- Implementation experience
- Production feedback

Please try and tell us about your experience or requirements

COMPANIES

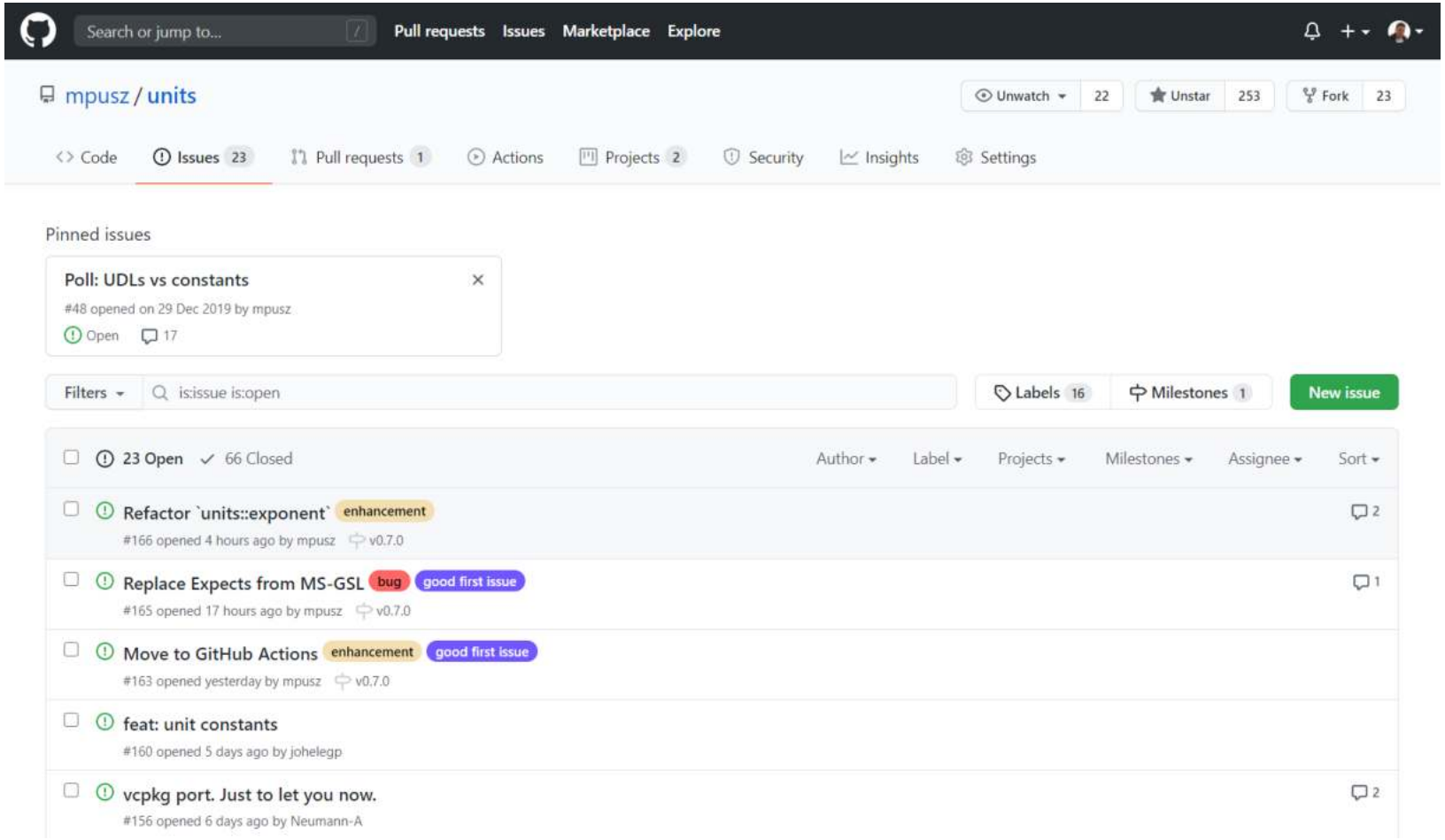
- Production/POC use
- Requirements

AUTHORS OF OTHER LIBRARIES

- Implementation experience
- Production feedback

GitHub Issues are not only to complain about the issues. Please also let us know if you are a happy user :-)

Design Discussions, Issues, Next Steps, Feedback (github.com/mpusz/units/issues)



Thank You! It is not just me...

Thank You! It is not just me...

EXISTING PRACTICE

- Matthias Christian Schabel & Steven Watanabe (Boost.Units)
- Nic Holthaus (github.com/nholthaus/units)
- Martin Moene (github.com/martinmoene/PhysUnits-CT)
- Jan A. Sende (github.com/jansende/benri)
- Others...

Thank You! It is not just me...

EXISTING PRACTICE

- Matthias Christian Schabel & Steven Watanabe (Boost.Units)
- Nic Holthaus (github.com/nholthaus/units)
- Martin Moene (github.com/martinmoene/PhysUnits-CT)
- Jan A. Sende (github.com/jansende/benri)
- Others...

C++ Physical Units library is nothing new. We have a production experience with various implementations for many years now.

Thank You! It is not just me...

CONTRIBUTORS

- Johel Ernesto Guerrero Peña (@JohelEGP)
- Riccardo Brugo (@rbrugo)
- Ramzi Sabra (@yasamoka)
- Andy Little (@kwikius)
- Ralph J. Steinhagen (@RalphSteinhagen)
- Oliver Schönrock (@oschonrock)
- Michael Ford (@mikeford1)
- Jan A. Sende (@jansende)
- @i-ky
- Others...

Thank You! It is not just me...

CONTRIBUTORS

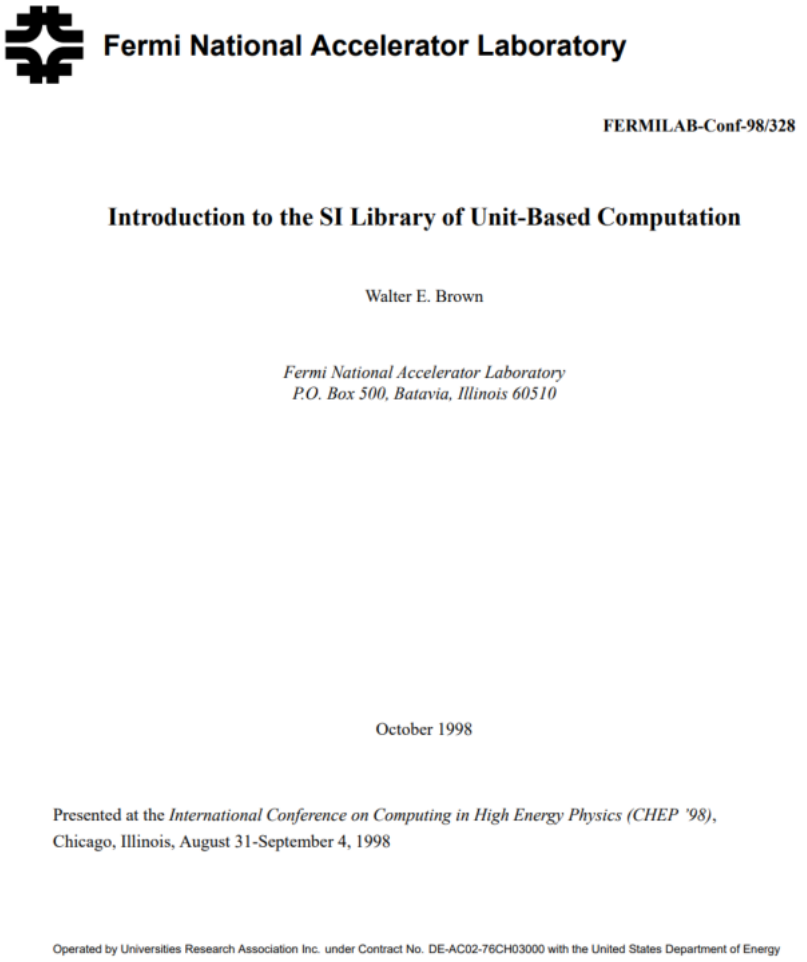
- Johel Ernesto Guerrero Peña (@JohelEGP)
- Riccardo Brugo (@rbrugo)
- Ramzi Sabra (@yasamoka)
- Andy Little (@kwikius)
- Ralph J. Steinhagen (@RalphSteinhagen)
- Oliver Schönrock (@oschonrock)
- Michael Ford (@mikeford1)
- Jan A. Sende (@jansende)
- @i-ky
- Others...

Contribution is not only about co-developing the code but also about taking part in discussions, sharing ideas, or stating requirements.

Thank You! It is not just me...

SPECIAL THANKS

- Walter Brown



Fermi National Accelerator Laboratory

FERMILAB-Conf-98/328

Introduction to the SI Library of Unit-Based Computation

Walter E. Brown

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

October 1998

Presented at the *International Conference on Computing in High Energy Physics (CHEP '98)*, Chicago, Illinois, August 31-September 4, 1998

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CH03000 with the United States Department of Energy

Thank You! It is not just me...

SPECIAL THANKS

- Howard Hinnant

Thank You! It is not just me...

SPECIAL THANKS

- GCC developers


CAUTION
Programming
is addictive
(and too much fun)