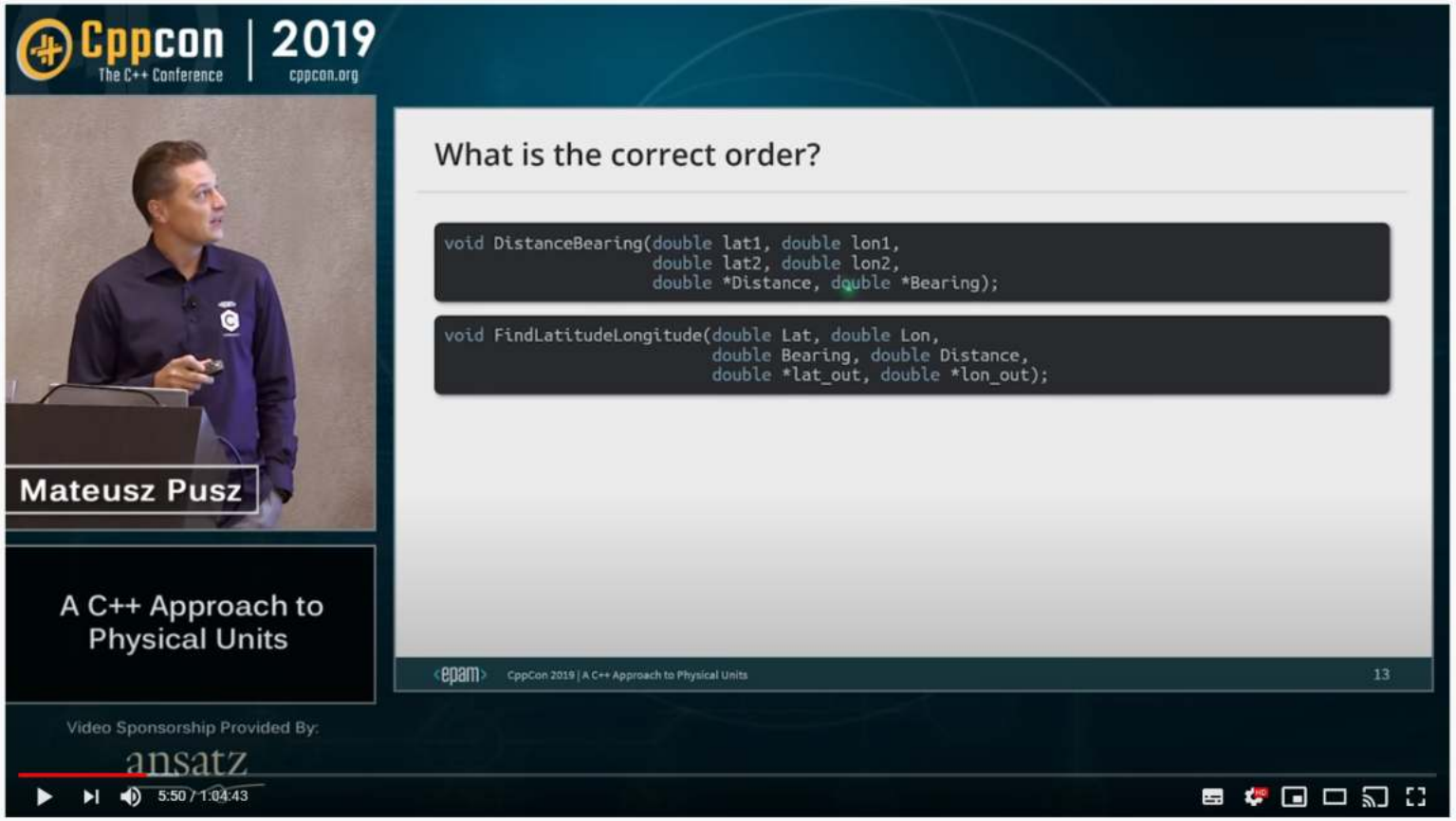
A Physical Units Library For the Next C++

Mateusz Pusz

September 15, 2020

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);
```

User Defined Literals (UDLs)

- The prefix **_q_** will be replaced with **q_** if the library will end up in the C++ Standard Library

User Defined Literals (UDLs)

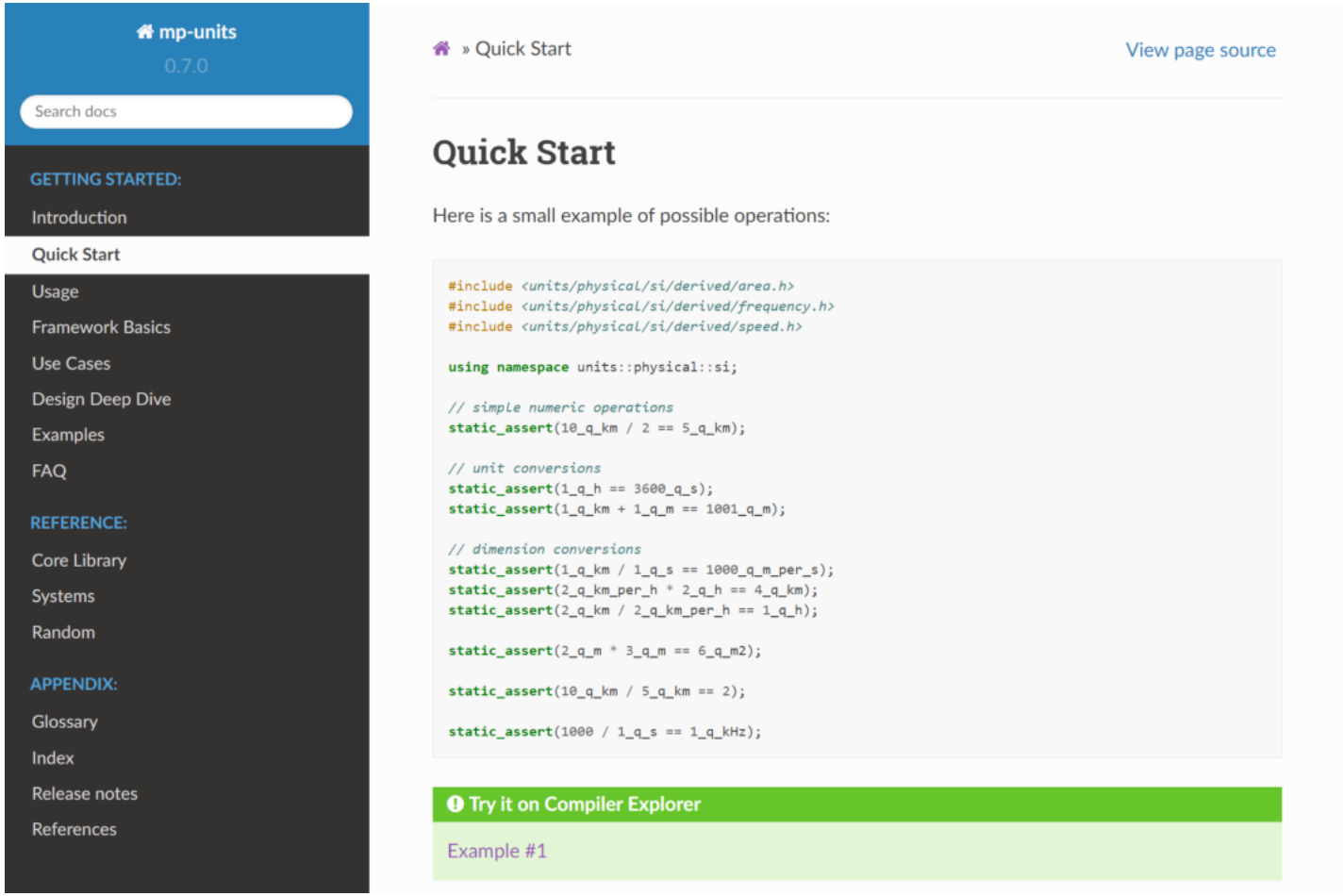
- The prefix **_q_** **will be replaced** with **q_** if the library will end up in the C++ Standard Library
- **q_** prefix needed to
  - *workaround issues with colliding built-in literals*
    - for example: **F** (farad), **J** (joule), **W** (watt), **K** (kelvin), **d** (day), **l** or **L** (litre), **erg**, **ergps**
  - *not collide with* **std::chrono::literals**

User Defined Literals (UDLs)

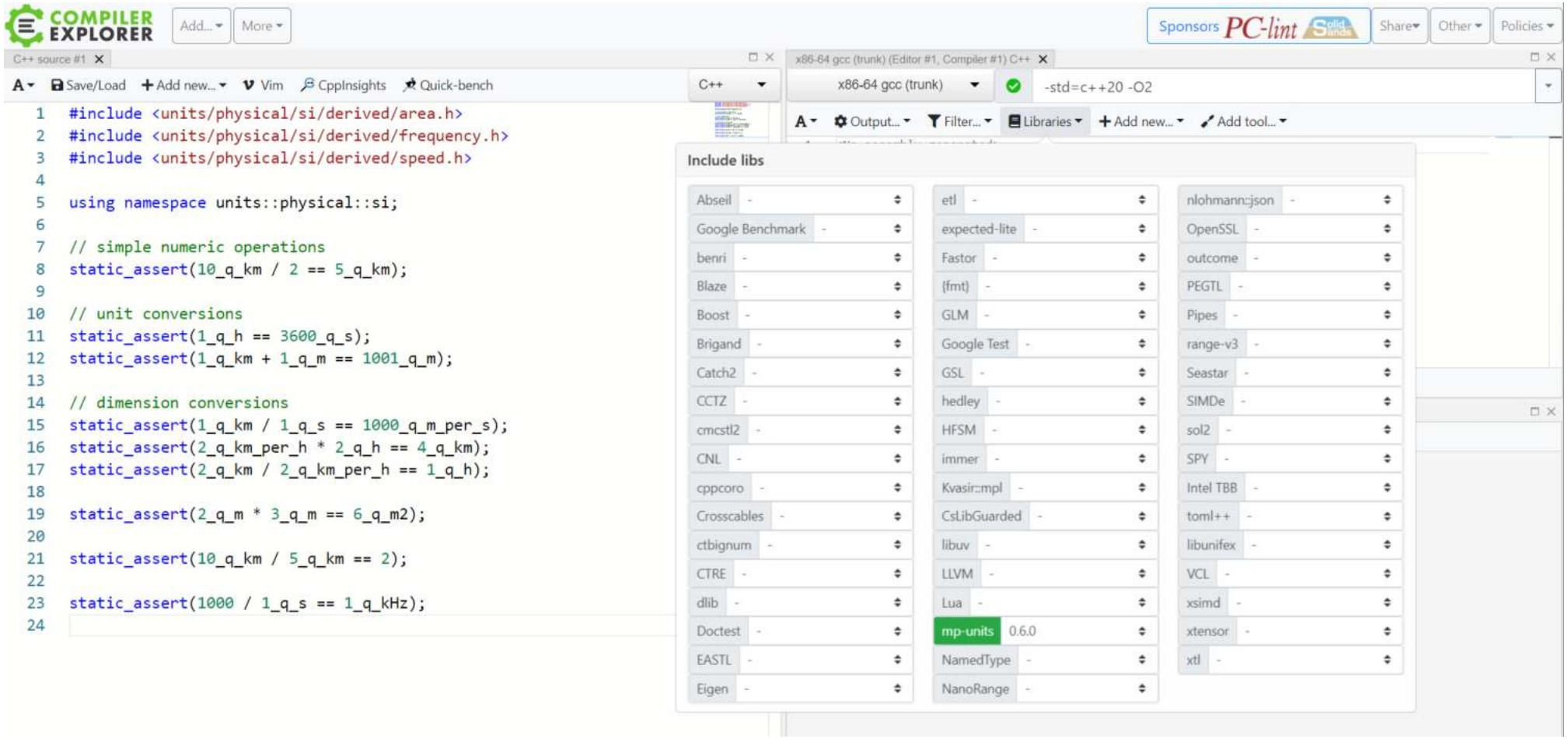
- The prefix **_q_** **will be replaced** with **q_** if the library will end up in the C++ Standard Library
- **q_** prefix needed to
  - *workaround issues with colliding built-in literals*
    - for example: **F** (farad), **J** (joule), **W** (watt), **K** (kelvin), **d** (day), **l** or **L** (litre), **erg**, **ergps**
  - *not collide with* **std::chrono::literals**

Ongoing research to provide an alternative way to create quantities... (mpusz/units#48)

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



mp-units on the Compiler Explorer (godbolt.org/z/cs19oj)



mp-units in Conan

CONAN CENTER (OFFICIAL RELEASES)

```
[requires]
mp-units/0.6.0

[generators]
cmake
```

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

mp-units in Conan

CONAN CENTER (OFFICIAL RELEASES)

```
[requires]
mp-units/0.6.0

[generators]
cmake
```

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

LIVE AT HEAD

```
[requires]
mp-units/0.7.0@mpusz/testing

[generators]
cmake
```

```
conan remote add conan-mpusz https://api.bintray.com/conan/mpusz/conan-mpusz
```

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

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)

```
#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)

```
#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)

```
#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;
}
```

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

```
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/Y1rjvY)

```cpp
#include <units/physical/si/derived/speed.h>
#include <units/physical/si/international/derived/speed.h>

using namespace units::physical;

constexpr si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d / t;
}
```

SI coherent units: mp-units (godbolt.org/z/Y1rjvY)

```cpp
#include <units/physical/si/derived/speed.h>
#include <units/physical/si/international/derived/speed.h>

using namespace units::physical;

constexpr si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d / t;
}
```

```cpp
auto toy_example_1(si::length<si::kilometre> d, si::time<si::hour> t)
{
  return quantity_cast<si::kilometre_per_hour>(avg_speed(d, t));
}
```

```cpp
auto toy_example_2(si::length<si::international::mile> d, si::time<si::hour> t)
{
  return quantity_cast<si::international::mile_per_hour>(avg_speed(d, t));
}
```

SI coherent units: mp-units (godbolt.org/z/Y1rjvY)

```cpp
#include <units/physical/si/derived/speed.h>
#include <units/physical/si/international/derived/speed.h>

using namespace units::physical;

constexpr si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d / t;
}
```

```cpp
auto toy_example_1(si::length<si::kilometre> d, si::time<si::hour> t)
{
  return quantity_cast<si::kilometre_per_hour>(avg_speed(d, t));
}
```

```cpp
auto toy_example_2(si::length<si::international::mile> d, si::time<si::hour> t)
{
  return quantity_cast<si::international::mile_per_hour>(avg_speed(d, t));
}
```

```cpp
using namespace units::physical::si::literals;
using namespace units::physical::si::international::literals;
std::cout << toy_example_1(220_q_km, 2_q_h) << "\n";  // prints "110 km/h"
std::cout << toy_example_2(140_q_mi, 2_q_h) << "\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/YGnvPE)

```
using namespace units::physical;
```

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

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)

```
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/x7q1Kb)

```cpp
constexpr si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d / t;
}

auto toy_example_1(si::length<si::kilometre> d, si::time<si::hour> t)
{
  return quantity_cast<si::kilometre_per_hour>(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/s5WEYo)

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

auto toy_example_1(si::length<si::kilometre> d, si::time<si::hour> t)
{
  return 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 si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d * t;
}
```

```
<source>: In function 'constexpr units::physical::si::speed<units::physical::si::metre_per_second> avg_speed(
  units::physical::si::length<units::physical::si::metre>, units::physical::si::time<units::physical::si::second>)':

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

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

Compiler returned: 1
```

SI coherent units: mp-units

```
constexpr si::speed<si::metre_per_second> avg_speed(si::length<si::metre> d, si::time<si::second> t)
{
  return d / t;
}
```

```
<source>: In function 'constexpr units::physical::si::speed<units::physical::si::metre_per_second> avg_speed(
  units::physical::si::length<units::physical::si::metre>, units::physical::si::time<units::physical::si::second>)':

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

    8 |   return d * t;
      |          ~~^~~
      |            |
      |            quantity<units::unknown_dimension<units::exponent<units::physical::si::dim_length, 1, 1>,
                         units::exponent<units::physical::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

```
#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/physical/si/derived/acceleration.h>

using namespace units::physical;
using namespace units::physical::si::literals;

si::acceleration<si::metre_per_second_sq> a = 100_q_m / 10_q_s;
```

Dimension mismatch: mp-units

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

using namespace units::physical;
using namespace units::physical::si::literals;

si::acceleration<si::metre_per_second_sq> a = 100_q_m / 10_q_s;
```

```
<source>:6:55: error: conversion from 'quantity<units::physical::si::dim_speed,units::physical::si::metre_per_second,long int>'
 to non-scalar type 'quantity<units::physical::si::dim_acceleration,units::physical::si::metre_per_second_sq,double>' requested

    6 | si::acceleration<si::metre_per_second_sq> a = 100_q_m / 10_q_s;
      |                                               ~~~~~~~~^~~~~~~~

Compiler returned: 1
```

Dimension mismatch: mp-units

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

using namespace units::physical;
using namespace units::physical::si::literals;

si::acceleration<si::metre_per_second_sq> a = 100_q_m / 10_q_s;
```

```
<source>:6:55: error: conversion from 'quantity<units::physical::si::dim_speed,units::physical::si::metre_per_second,long int>'
 to non-scalar type 'quantity<units::physical::si::dim_acceleration,units::physical::si::metre_per_second_sq,double>' requested

    6 | si::acceleration<si::metre_per_second_sq> a = 100_q_m / 10_q_s;
      |                                               ~~~~~~~~^~~~~~~~

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(220_q_km, 2_q_h);
```

Debugging: mp-units

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

```
avg_speed(220_q_km, 2_q_h);
```

```
Breakpoint 2, avg_speed<units::quantity<units::physical::si::dim_length, units::physical::si::kilometre, long>,
units::quantity<units::physical::si::dim_time, units::physical::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) { /* ... */ }
```

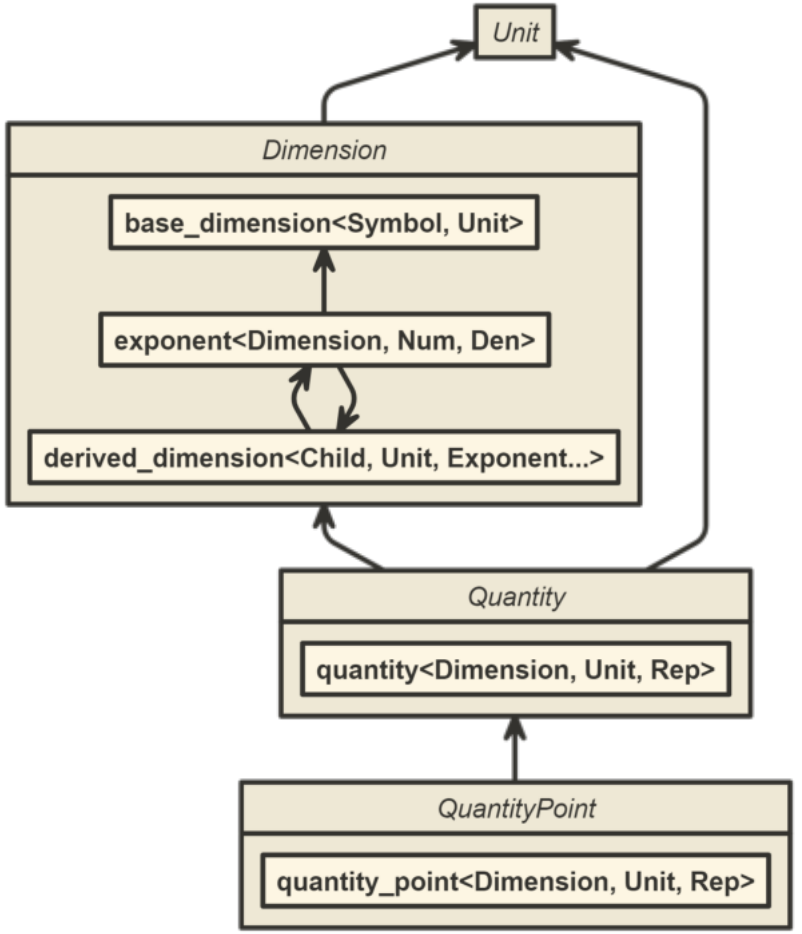
```
auto foo = [](auto&& t) { /* ... */ };
```

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

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

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

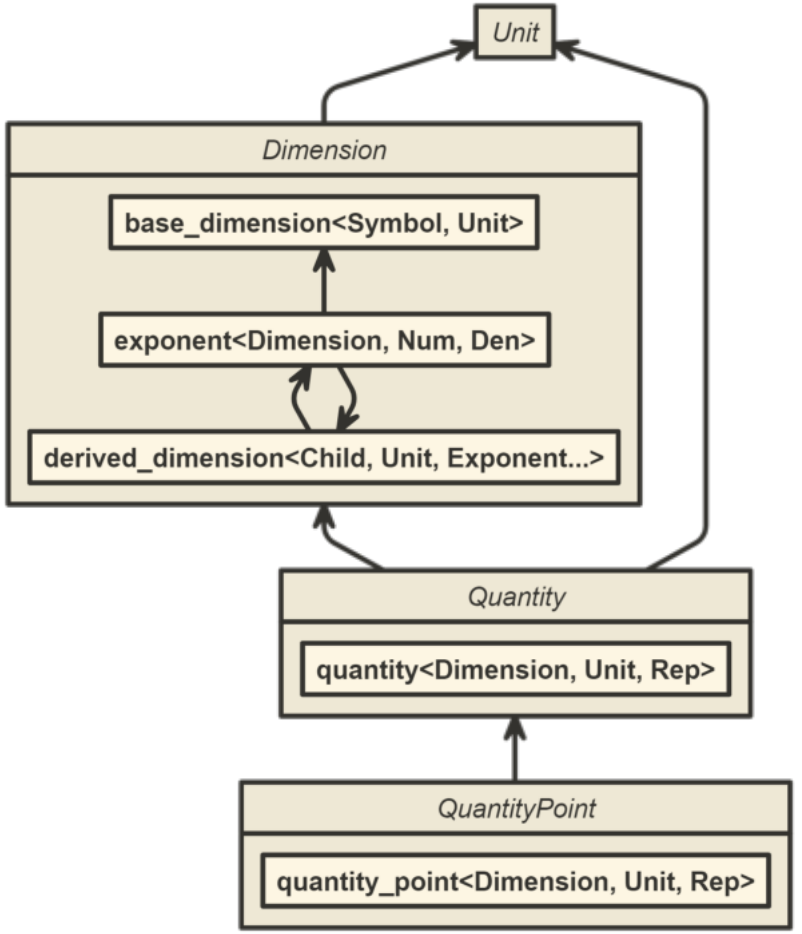
Basic Concepts



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

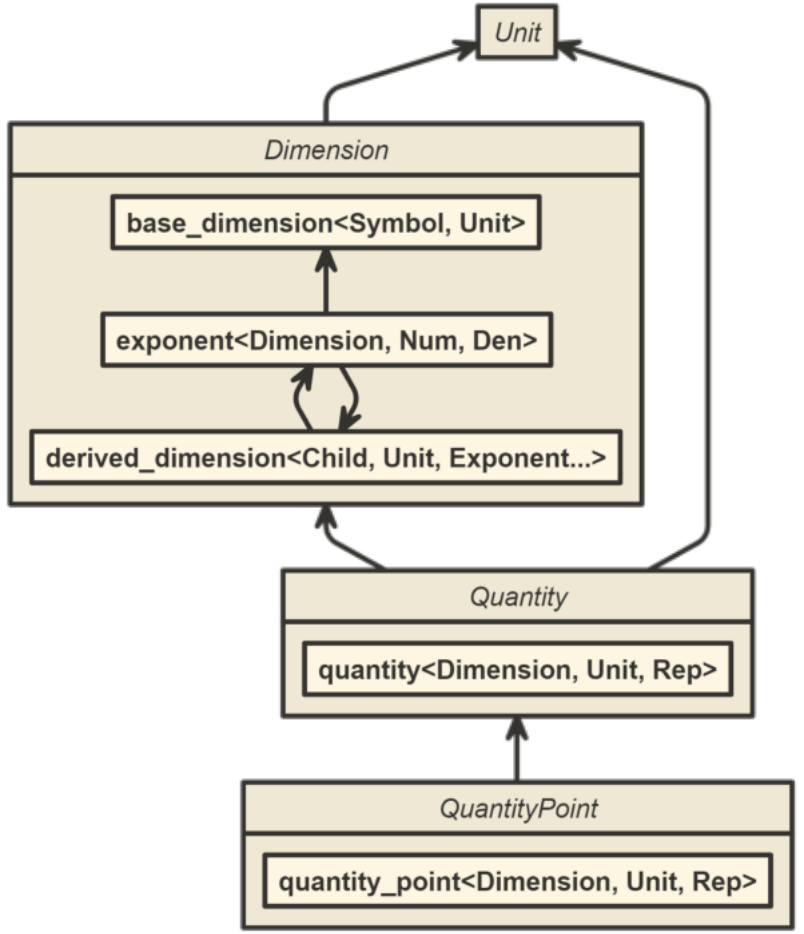


Basic Concepts

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> {};
```



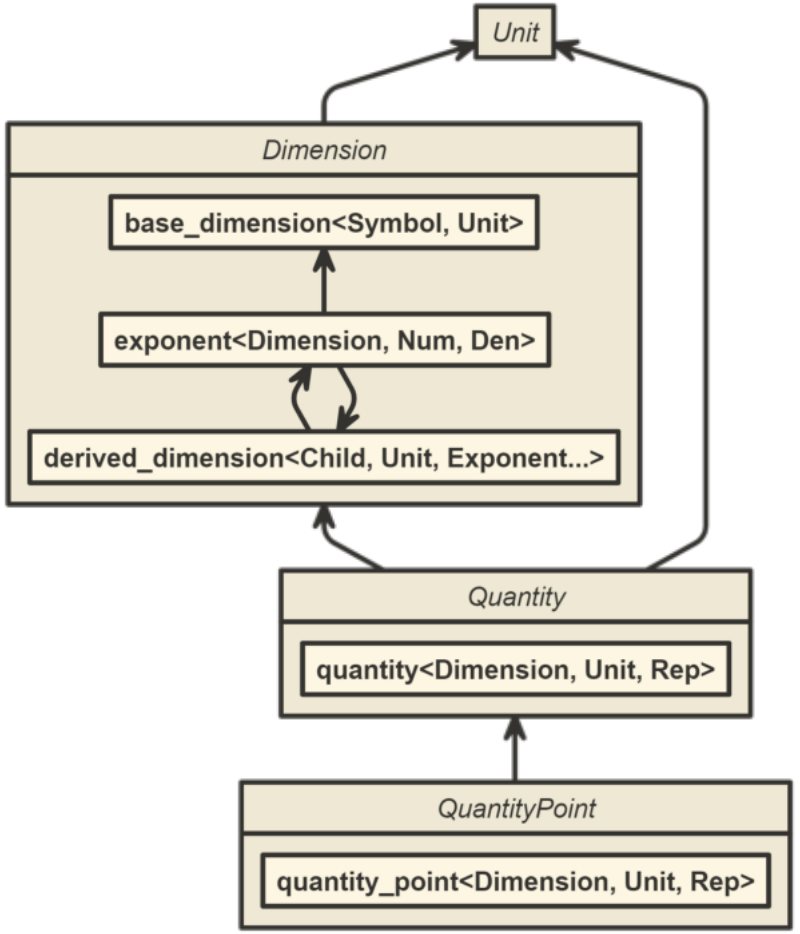
Basic Concepts

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

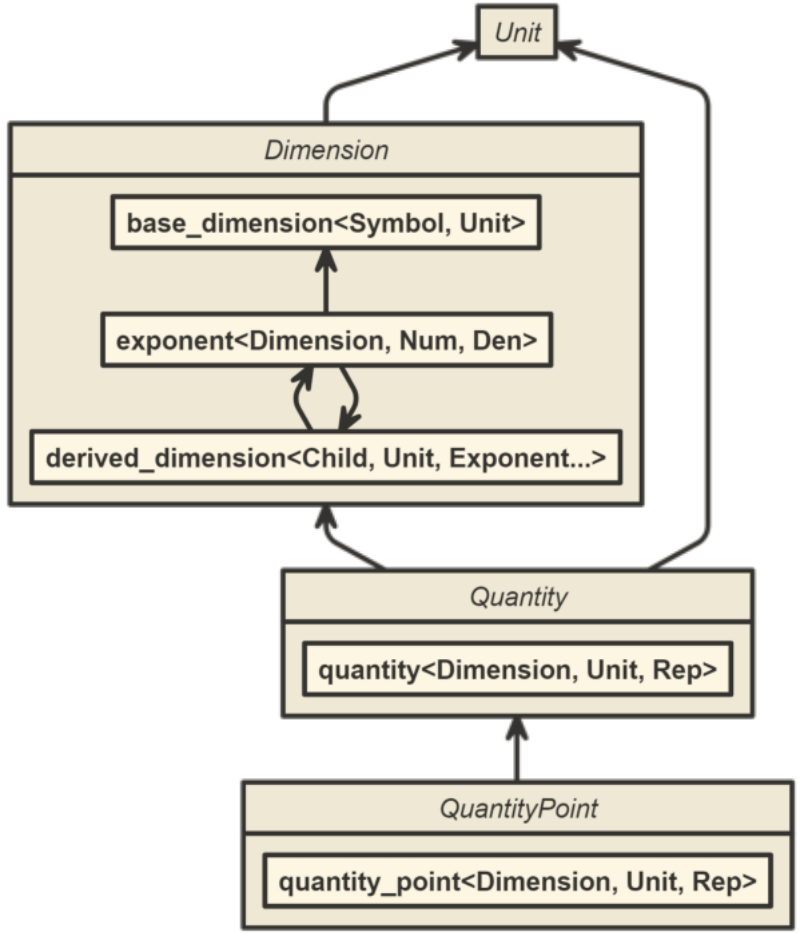
UNIT

- 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> {};
```

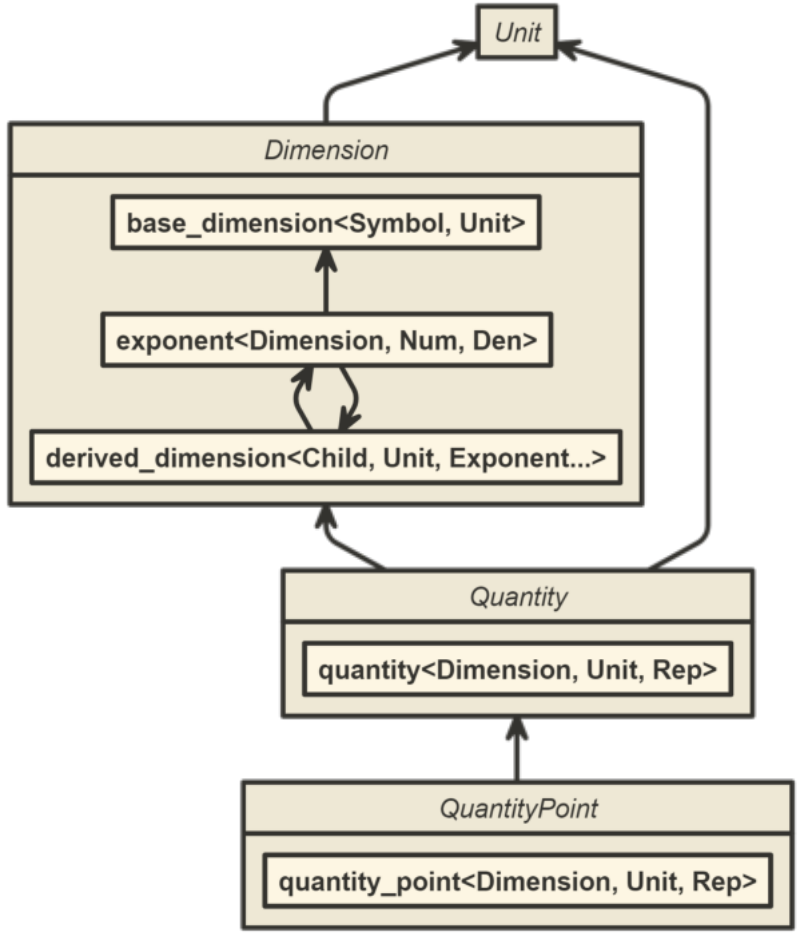
```
struct metre_per_second : unit<metre_per_second> {};
struct kilometre_per_hour : deduced_unit<
  kilometre_per_hour, dim_speed, kilometre, hour> {};
```



Basic Concepts

DIMENSION

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

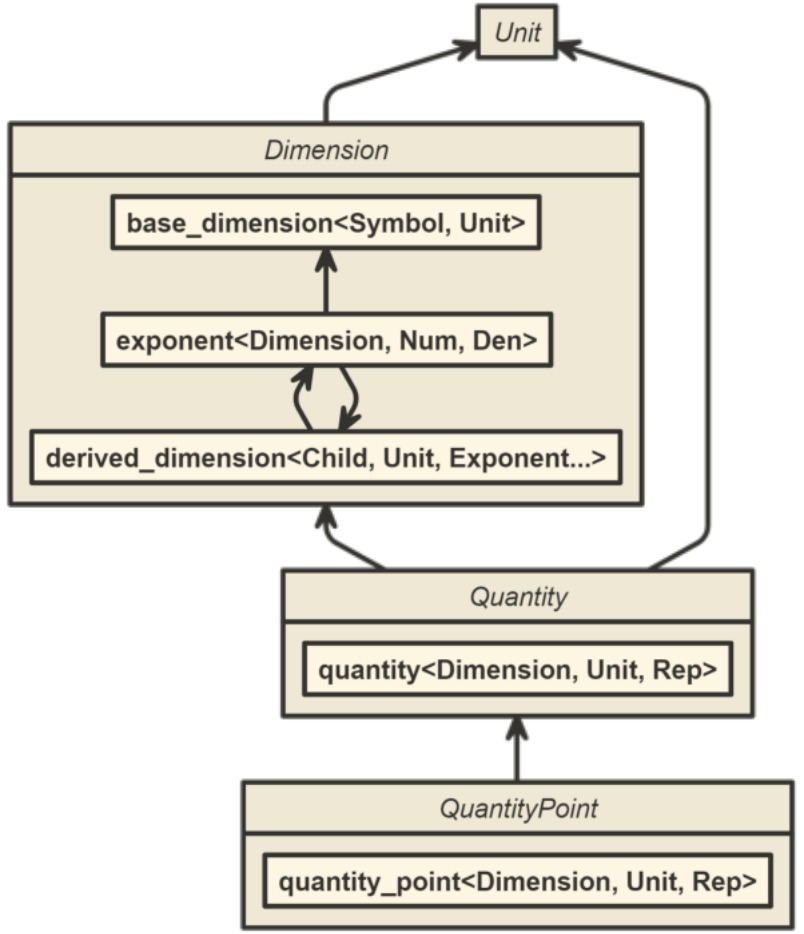


Basic Concepts

DIMENSION

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

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



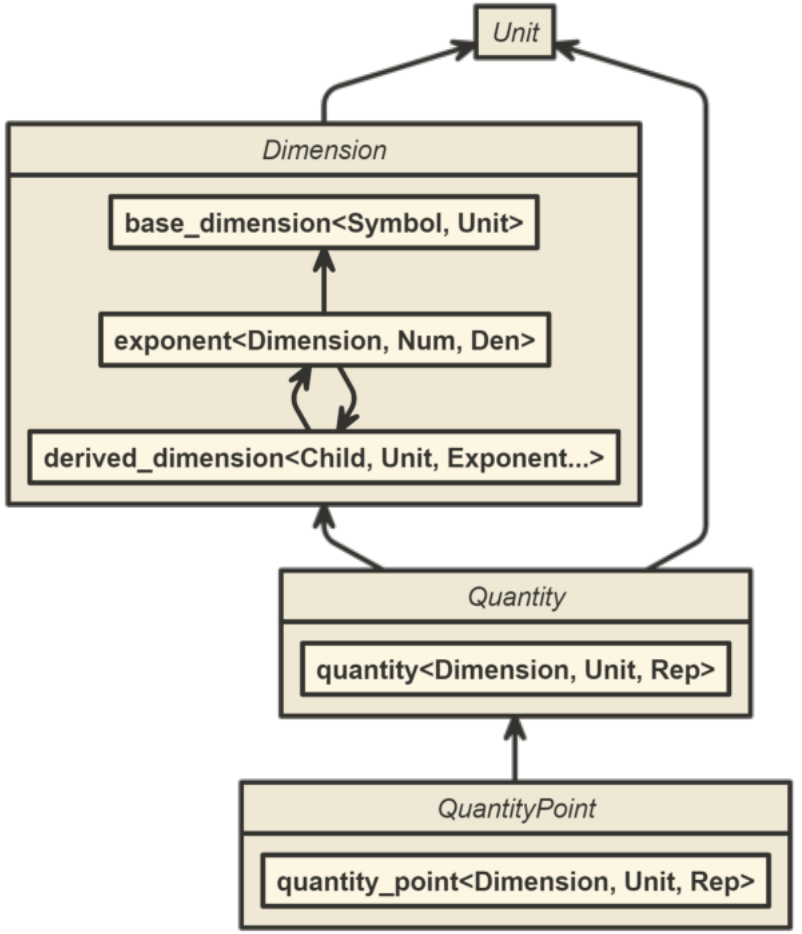
Basic Concepts

DIMENSION

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

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

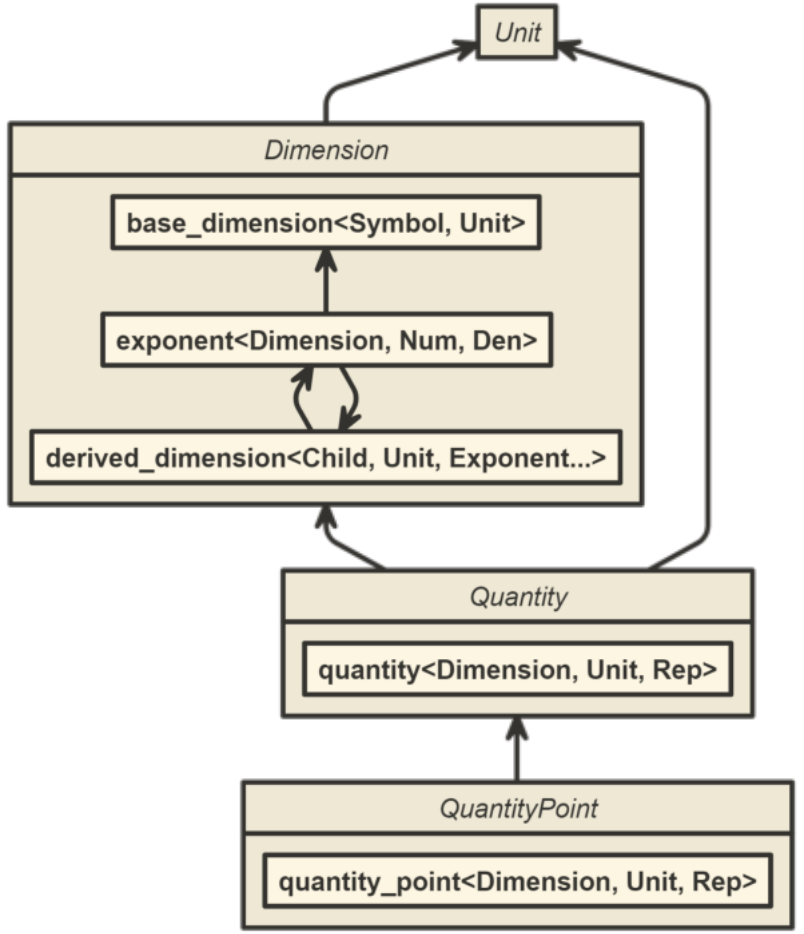
```
struct dim_speed : derived_dimension<dim_speed,
                                     metre_per_second,
                                     exponent<dim_length, 1>,
                                     exponent<dim_time, -1>> {};
```



Basic Concepts

QUANTITY

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



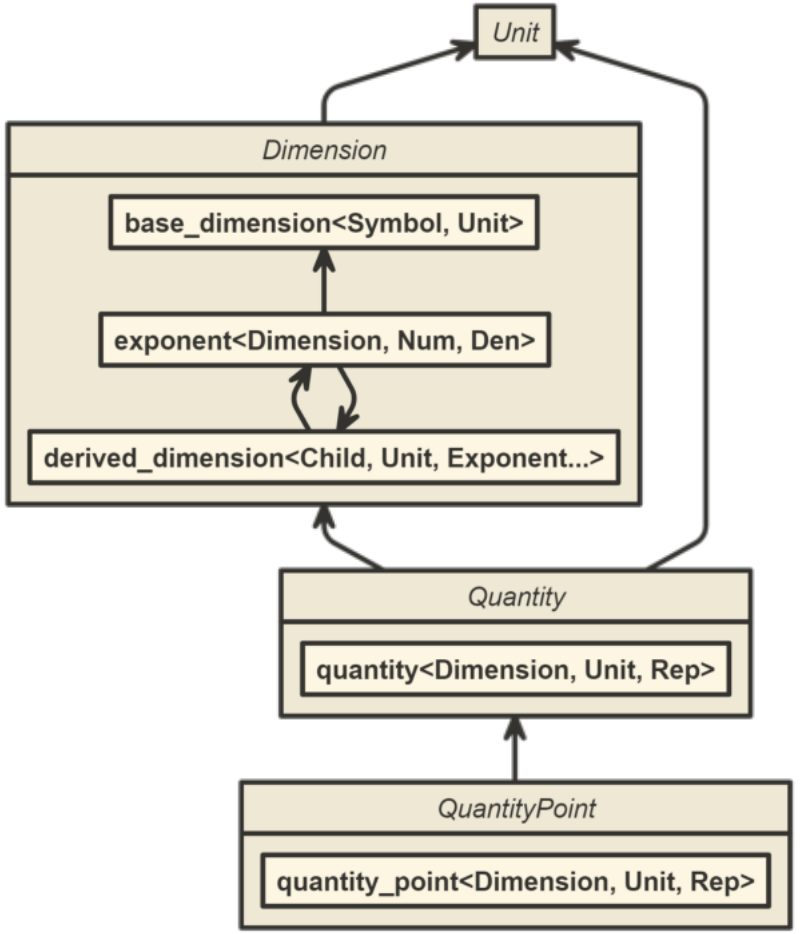
Basic Concepts

QUANTITY

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

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

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



Basic Concepts

QUANTITY

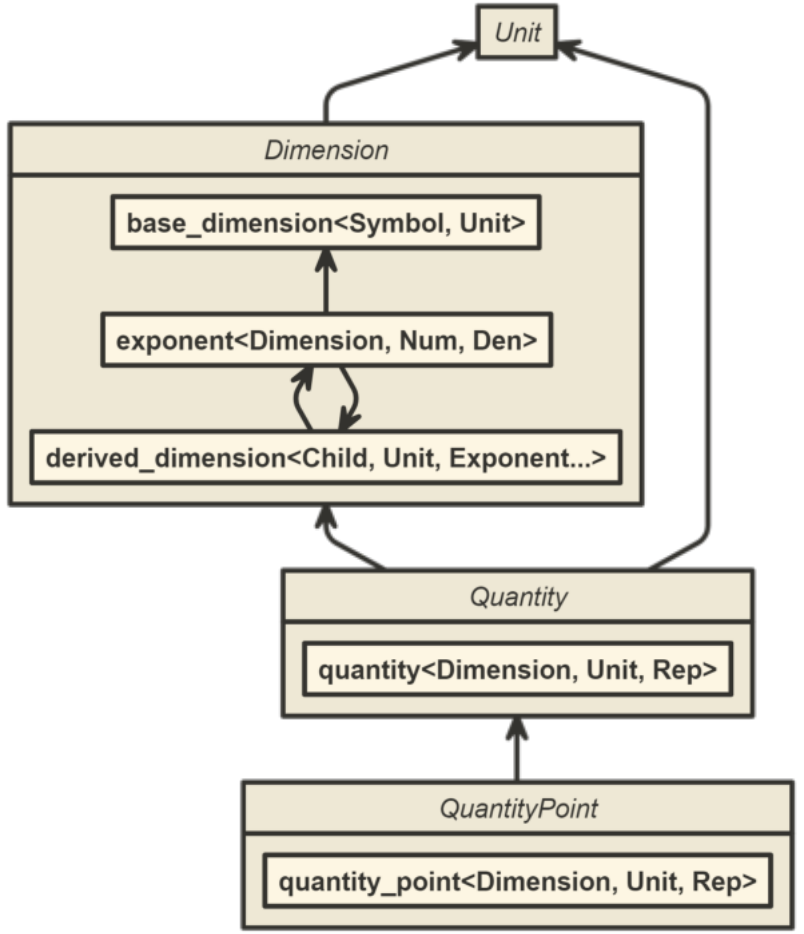
- A concrete *amount of a unit* for a *specified dimension* with a *specific representation*

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

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

QUANTITY POINT

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



Basic Concepts

QUANTITY

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

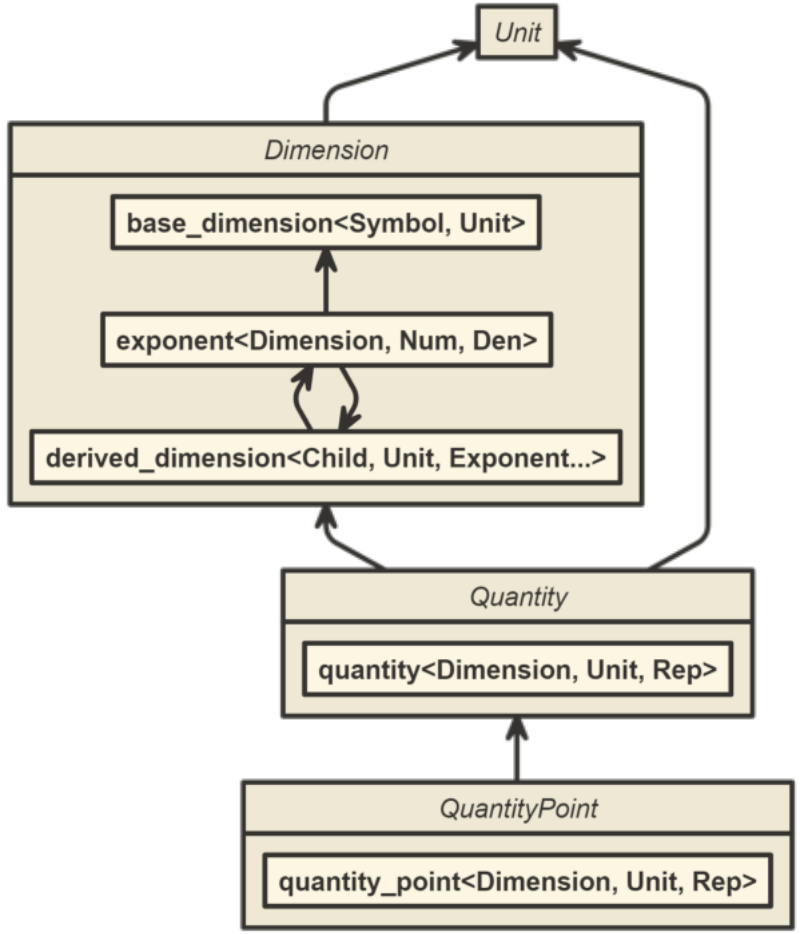
```
template<Unit U, ScalableNumber Rep = double>
using length = quantity<dim_length, U, Rep>;
```

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

QUANTITY POINT

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

```
units::quantity_point d(123_q_km);
```



Concepts example

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

```
constexpr price calc_fine(units::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(units::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(units::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(units::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(units::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(units::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';
```

Not just a syntactic sugar

Not just a syntactic sugar

- Constraining *function template return types*

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

Not just a syntactic sugar

- Constraining *function template return types*

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

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

```
const units::Speed auto speed = avg_speed(220_q_km, 2_q_h);
```

Not just a syntactic sugar

- Constraining *function template return types*

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

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

```
const units::Speed auto speed = avg_speed(220_q_km, 2_q_h);
```

- 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 **`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 **`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*

What if a unit was a constant value rather than a type?

```
constexpr auto metre = 1._q_m;
constexpr auto kilo = 1000;
constexpr auto kilometre = kilo * metre;

constexpr auto second = 1._q_s;
constexpr auto hour = 3600 * second;
```

What if a unit was a constant value rather than a type?

```
constexpr auto metre = 1._q_m;
constexpr auto kilo = 1000;
constexpr auto kilometre = kilo * metre;

constexpr auto second = 1._q_s;
constexpr auto hour = 3600 * second;
```

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

What if a unit was a constant value rather than a type?

```
constexpr auto metre = 1._q_m;
constexpr auto kilo = 1000;
constexpr auto kilometre = kilo * metre;

constexpr auto second = 1._q_s;
constexpr auto hour = 3600 * second;
```

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

```
std::cout << avg_speed(220 * kilometre, 2 * hour) << "\n";  // prints "30.5556 m/s"
```

What if a unit was a constant value rather than a type?

```
constexpr auto metre = 1._q_m;
constexpr auto kilo = 1000;
constexpr auto kilometre = kilo * metre;

constexpr auto second = 1._q_s;
constexpr auto hour = 3600 * second;
```

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

```
std::cout << avg_speed(220 * kilometre, 2 * hour) << "\n";  // prints "30.5556 m/s"
```

Not needed additional conversions result in a worse runtime performance and loss of precision.

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.

basic_fixed_string

```cpp
template<typename CharT, std::size_t N>
struct basic_fixed_string {
  CharT data_[N + 1] = {};  // has to be public to be a structural type

  using iterator = CharT*;
  using const_iterator = const CharT*;

  constexpr basic_fixed_string(CharT ch) noexcept;

  constexpr basic_fixed_string(const CharT (&txt)[N + 1]) noexcept;

  [[nodiscard]] constexpr std::size_t size() const noexcept;
  [[nodiscard]] constexpr const CharT* c_str() const noexcept;
  [[nodiscard]] constexpr const CharT& operator[](std::size_t index) const noexcept;
  [[nodiscard]] constexpr CharT operator[](std::size_t index) noexcept;

  [[nodiscard]] constexpr iterator begin() noexcept;
  [[nodiscard]] constexpr const_iterator begin() const noexcept;
  [[nodiscard]] constexpr iterator end() noexcept;
  [[nodiscard]] constexpr const_iterator end() const noexcept;

  template<std::size_t N2>
  [[nodiscard]] constexpr friend basic_fixed_string<CharT, N + N2>
    operator+(const basic_fixed_string& lhs, const basic_fixed_string<CharT, N2>& rhs) noexcept;

  // ...
};
```

basic_fixed_string

```cpp
template<typename CharT, std::size_t N>
struct basic_fixed_string {
  // ...

  [[nodiscard]] constexpr bool operator==(const basic_fixed_string& other) const
  {
    return std::ranges::equal(*this, other);
  }

  template<std::size_t N2>
  [[nodiscard]] friend constexpr bool operator==(const basic_fixed_string&, const basic_fixed_string<CharT, N2>&)
  { return false; }

  template<std::size_t N2>
  [[nodiscard]] friend constexpr auto operator<=>(const basic_fixed_string& lhs, const basic_fixed_string<CharT, N2>& rhs)
  {
    return std::lexicographical_compare_three_way(lhs.begin(), lhs.end(), rhs.begin(), rhs.end());
  }

  template<class Traits>
  friend std::basic_ostream<CharT, Traits>& operator<<(std::basic_ostream<CharT, Traits>& os,
                                                       const basic_fixed_string& txt);
};
```

Text Output

OUTPUT STREAMS

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

Text Output

OUTPUT STREAMS

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

FMT::FORMAT

```
fmt::print("{}", 123_q_km);        // 123 km
fmt::print("{:%Q}", 123_q_km);     // 123
fmt::print("{:%q}", 123_q_km);     // km
fmt::print("{:%Q%q}", 123_q_km);   // 123km
```

Unicode and ASCII-only

```
fmt::print("{}", 10_q_R);                  // 10 Ω
fmt::print("{:%Q %Aq}", 10_q_R);           // 10 ohm
fmt::print("{}", 125_q_us);                // 125 µs
fmt::print("{:%Q %Aq}", 125_q_us);         // 125 us
fmt::print("{}", 9.8_q_m_per_s2);          // 9.8 m/s²
fmt::print("{:%Q %Aq}", 9.8_q_m_per_s2);   // 9.8 m/s^2
```

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

Linear Algebra vs. Quantities (P1385)

LINEAR ALGEBRA OF QUANTITIES

```
fs_vector<si::length<si::metre>, 3> v = { 1_q_m, 2_q_m, 3_q_m };
fs_vector<si::length<si::metre>, 3> u = { 3_q_m, 2_q_m, 1_q_m };
fs_vector<si::length<si::kilometre>, 3> t = { 3_q_km, 2_q_km, 1_q_km };
```

Linear Algebra vs. Quantities (P1385)

LINEAR ALGEBRA OF QUANTITIES

```
fs_vector<si::length<si::metre>, 3> v = { 1_q_m, 2_q_m, 3_q_m };
fs_vector<si::length<si::metre>, 3> u = { 3_q_m, 2_q_m, 1_q_m };
fs_vector<si::length<si::kilometre>, 3> t = { 3_q_km, 2_q_km, 1_q_km };
```

```
std::cout << "v + u     = " << v + u << "\n";
std::cout << "v + t     = " << v + t << "\n";
std::cout << "t_in_m    = " << fs_vector<si::length<si::metre>, 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

```
fs_vector<si::length<si::metre>, 3> v = { 1_q_m, 2_q_m, 3_q_m };
fs_vector<si::length<si::metre>, 3> u = { 3_q_m, 2_q_m, 1_q_m };
fs_vector<si::length<si::kilometre>, 3> t = { 3_q_km, 2_q_km, 1_q_km };
```

```
std::cout << "v + u     = " << v + u << "\n";
std::cout << "v + t     = " << v + t << "\n";
std::cout << "t_in_m    = " << fs_vector<si::length<si::metre>, 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

```
si::length<si::metre, fs_vector<int, 3>> v(fs_vector<int, 3>{ 1, 2, 3 });
si::length<si::metre, fs_vector<int, 3>> u(fs_vector<int, 3>{ 3, 2, 1 });
si::length<si::kilometre, fs_vector<int, 3>> t(fs_vector<int, 3>{ 3, 2, 1 });
```

Linear Algebra vs. Quantities (P1385)

QUANTITIES OF LINEAR ALGEBRA TYPES

```
si::length<si::metre, fs_vector<int, 3>> v(fs_vector<int, 3>{ 1, 2, 3 });
si::length<si::metre, fs_vector<int, 3>> u(fs_vector<int, 3>{ 3, 2, 1 });
si::length<si::kilometre, fs_vector<int, 3>> t(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-9.3 support dropped in mp-units/0.6.0
- gcc-10+
- Visual Studio 16.7+ (with a few exceptions)

Compilers support

- gcc-9.3 support dropped in mp-units/0.6.0
- gcc-10+
- Visual Studio 16.7+ (with a few exceptions)

MSVC SYNTAX

- gcc

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

- Visual Studio 16.7

```
template<Length L, Time T>
Speed auto avg_speed(L d, T t)
{
  return d / t;
}
```

Compilers support

- gcc-9.3 support dropped in mp-units/0.6.0
- gcc-10+
- Visual Studio 16.7+ (with a few exceptions)

MSVC SYNTAX

- gcc

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

- Visual Studio 16.7

```
template<Length L, Time T>
Speed auto avg_speed(L d, T t)
{
  return d / t;
}
```

We hope that clang will catch up soon.

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
- **C++23 might be challenging**
  - we need more field experience and feedback
  - schedule for C++23 is tight
  - 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
- **C++23 might be challenging**
  - we need more field experience and feedback
  - schedule for C++23 is tight
  - 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

AUTHORS OF OTHER LIBRARIES

- Implementation experience
- Production feedback

Please try and tell us about your experience or requirements

COMPANIES

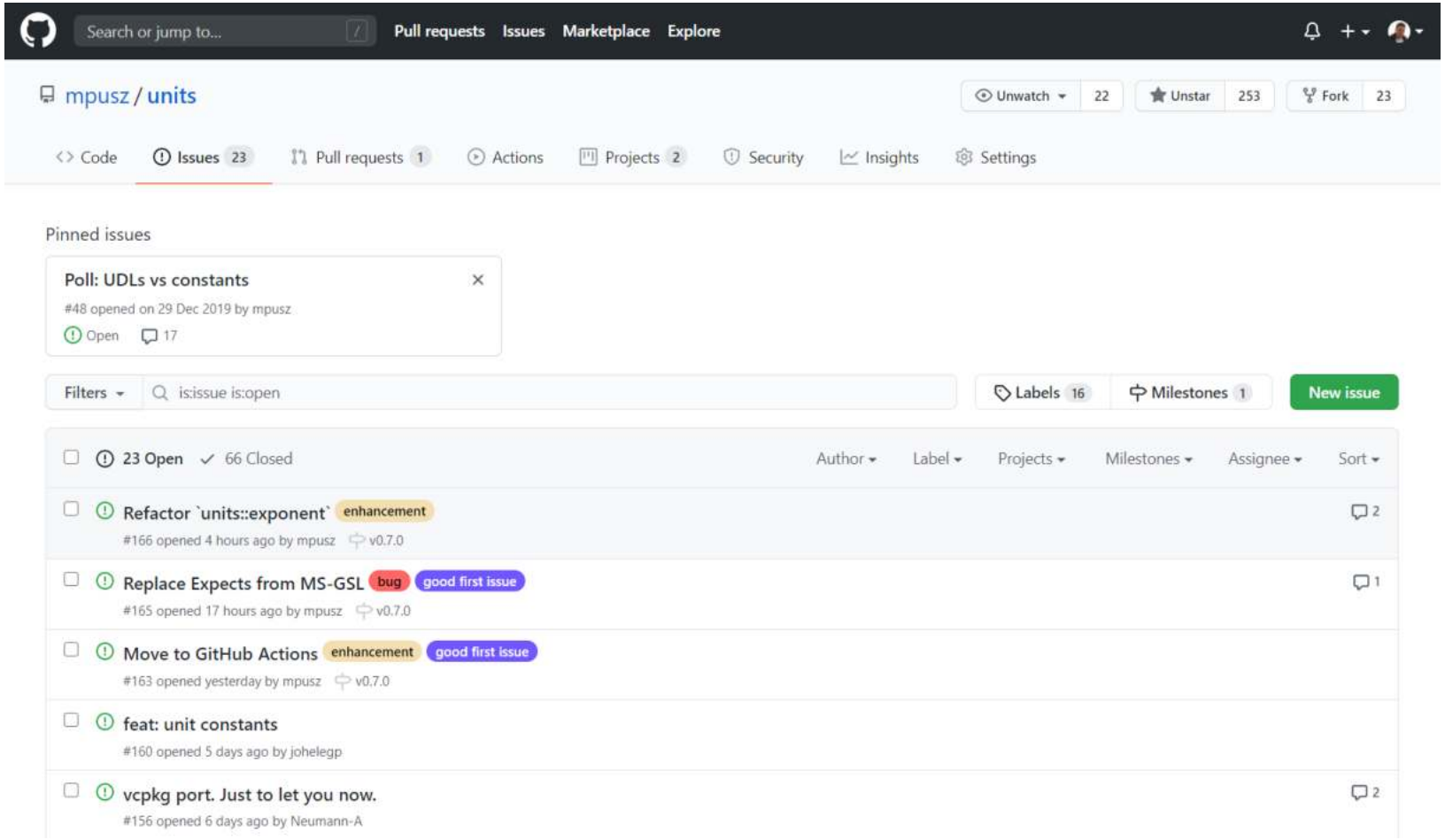
- Production/POC use
- Requirements

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)
- Oliver Schönrock (@oschonrock)
- Michael Ford (@mikeford1)
- Jan A. Sende (@jansende)
- @i-ky
- Others...

Thank You! It is not just me...

CONTRIBUTORS

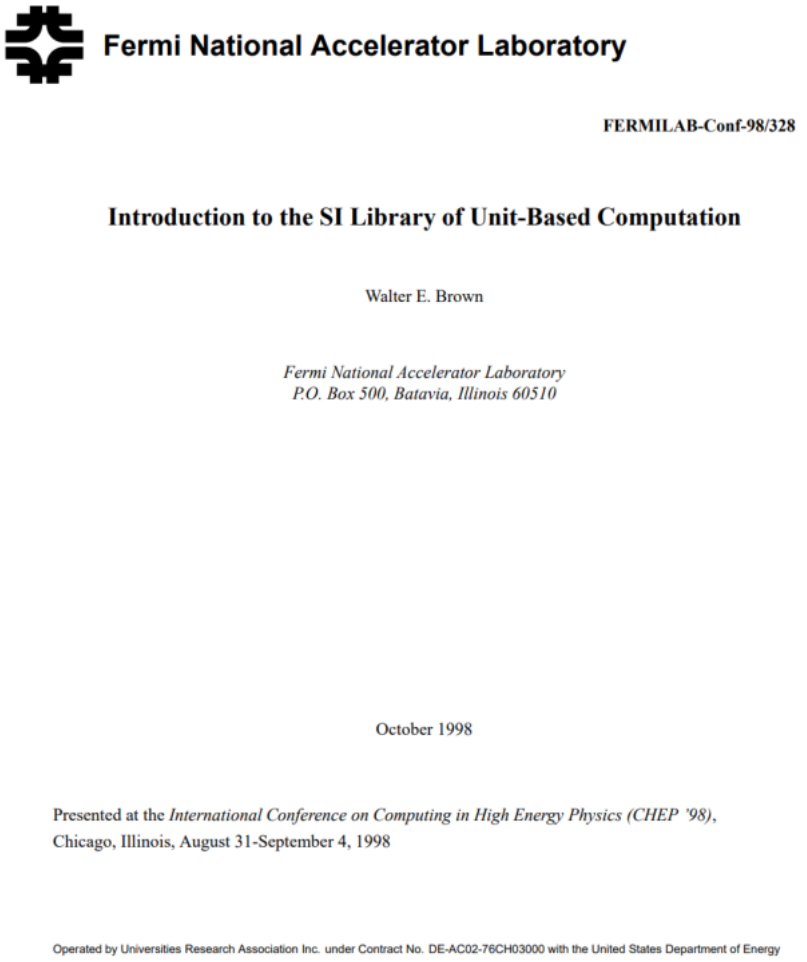
- Johel Ernesto Guerrero Peña (@johelegp)
- Riccardo Brugo (@rbrugo)
- Ramzi Sabra (@yasamoka)
- Andy Little (@kwikius)
- Oliver Schönrock (@oschonrock)
- Michael Ford (@mikeford1)
- Jan A. Sende (@jansende)
- @i-ky
- Others...

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

Join the C++ Concepts CppCon2020 Class

C++ Concepts: Constraining C++ Templates in C++20 and Before

C++ Concepts: Constraining C++ Templates in C++20 and Before is a two-day online training course with programming exercises taught by Mateusz Pusz. It is offered online from 11AM to 3PM Eastern Time (EDT), Monday September 21st and Tuesday September 22nd, 2020 (after the conference).

Course Description

C++ Concepts is one of the most significant and long-awaited features of C++20. They improve template interfaces by explicitly stating the compile-time contract between the user and the architect of the code. Concepts limit the number of compilation errors and make them much more user-friendly when they occur.

The workshop will describe this C++20 feature, its similarities and differences to Concepts TS (provided with gcc-7), and will present ways to benefit from a significant part of the functionality in current production C++ projects that can use only "legacy" C++11 features.

The training targets C++ developers and code architects that want to improve their skills in implementing general-purpose tools.

-2
DAYS REMAINING

Attendee quote

"Very interesting and knowledge-filled conference. You go there, meet really interesting people, and hear about things that you didn't know about C++. In the week I was there, I learned a significant amount of new content that I could immediately apply in my job."


CAUTION
Programming
is addictive
(and too much fun)