<epam>

Unleashing the Power of C++ Templates with `mp-units`

LESSONS LEARNED AND A NEW LIBRARY DESIGN

Mateusz Pusz
June 29, 2023

LESSONS LEARNED

Issues with the previous library design

1. The Downcasting Facility issues
2. Issues not addressable with the current framework
3. Why UDLs are not a good solution?
4. Verbose to define (and standardize) systems

The Downcasting Facility

An innovative powerful feature that was a cornerstone of the V1 design.

The Downcasting Facility

```cpp
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

The Downcasting Facility

```cpp
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```cpp
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility

```cpp
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```cpp
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

```
70 km/h
```

The Downcasting Facility

```cpp
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```cpp
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

```
(gdb) ptype s
type = class units::quantity<units::isq::si::dim_speed, units::isq::si::kilometre_per_hour, int>
[with D = units::isq::si::dim_speed, U = units::isq::si::kilometre_per_hour, Rep = int] {
...
```

The Downcasting Facility

- Introduces a 1-to-1 relationship between
  - long specialization of a generic class template (**`derived_dimension`** or **`scaled_unit`**)
  - nicely named user type

The Downcasting Facility

- Introduces a 1-to-1 relationship between
  - long specialization of a generic class template (**derived_dimension** or **scaled_unit**)
  - nicely named user type

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

struct kilometre_per_hour : derived_scaled_unit<kilometre_per_hour,
                                                dim_speed, kilometre, hour> {};
```

The Downcasting Facility

- Introduces a 1-to-1 relationship between
  - long specialization of a generic class template (**derived_dimension** or **scaled_unit**)
  - nicely named user type

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

struct kilometre_per_hour : derived_scaled_unit<kilometre_per_hour,
                                                dim_speed, kilometre, hour> {};
```

CRTP required to pass a nicely named child class type to the downcasting facility framework.

The Downcasting Facility: Issues

The Downcasting Facility: Issues

There are N-to-1 relationships in the physical units domain.

The Downcasting Facility: Issues

There are N-to-1 relationships in the physical units domain.

SAME DIMENSIONS

- Energy = $M\,L^2T^{-2}$; Torque = $M\,L^2T^{-2}$ (according to the SI)
- Frequency = T^{-1}; Activity of radionuclides = T^{-1}; Modulation rate = T^{-1}

The Downcasting Facility: Issues

There are N-to-1 relationships in the physical units domain.

SAME DIMENSIONS

- Energy = $M\,L^2T^{-2}$; Torque = $M\,L^2T^{-2}$ (according to the SI)
- Frequency = T^{-1}; Activity of radionuclides = T^{-1}; Modulation rate = T^{-1}

SAME UNITS

- **Hz** = 1/s; **Bq** = 1/s; **Bd** = 1/s; 1/s is a standalone derived unit on its own

The Downcasting Facility: Issues

There are N-to-1 relationships in the physical units domain.

SAME DIMENSIONS

- Energy = $M\,L^2T^{-2}$; Torque = $M\,L^2T^{-2}$ (according to the SI)
- Frequency = T^{-1}; Activity of radionuclides = T^{-1}; Modulation rate = T^{-1}

SAME UNITS

- **Hz** = 1/s; **Bq** = 1/s; **Bd** = 1/s; 1/s is a standalone derived unit on its own

Compile-time errors when both definitions are found.

The Downcasting Facility: Issues

- Hacks as workarounds

```
struct baud : alias_unit<si::hertz, "Bd"> {};
struct kilobaud : prefixed_alias_unit<si::kilohertz, si::kilo, baud> {};
// ...

using dim_modulation_rate = si::dim_frequency;
```

The Downcasting Facility: Issues

- Hacks as workarounds

```
struct baud : alias_unit<si::hertz, "Bd"> {};
struct kilobaud : prefixed_alias_unit<si::kilohertz, si::kilo, baud> {};
// ...

using dim_modulation_rate = si::dim_frequency;
```

- **`si::dim_frequency`** observable in the compilation errors and debugger
- Prints **Hz** on temporary results (if not explicitly converted to **baud**)
- Not possible to print **`1/s`** when this unit is actually expected

The Downcasting Facility: Issues

```
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/si/speed.h>

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t);
```

```
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility: Issues

```
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/dimensions/speed.h>  // include what you use

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t);
```

```
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility: Issues

```
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>
#include <units/isq/dimensions/speed.h>  // include what you use

using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t);
```

```
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

```
70 [5/18] m/s
```

```
units::quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, 1l, 1l>,
units::exponent<units::isq::si::dim_time, -1l, 1l> >, units::scaled_unit<units::magnitude<
units::base_power<long>{2l, units::ratio{-1l, 1l}}, units::base_power<long>{3l,
units::ratio{-2l, 1l}}, units::base_power<long>{5l, units::ratio{1l, 1l}}>{},
units::unknown_coherent_unit<units::exponent<units::isq::si::dim_length, 1l, 1l>,
units::exponent<units::isq::si::dim_time, -1l, 1l> > >, int>
```

The Downcasting Facility: Issues

avg_speed.h

```
#include <units/isq/dimensions/speed.h>
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>

constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t);
```

The Downcasting Facility: Issues

avg_speed.h

```
#include <units/isq/dimensions/speed.h>
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>

constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t);
```

a.cpp

```
#include <units/isq/si/speed.h>
#include "avg_speed.h"

using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility: Issues

avg_speed.h

```
#include <units/isq/dimensions/speed.h>
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>

constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t);
```

a.cpp

```
#include <units/isq/si/speed.h>
#include "avg_speed.h"

using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

b.cpp

```
#include "avg_speed.h"

using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility: Issues

avg_speed.h

```
#include <units/isq/dimensions/speed.h>
#include <units/isq/si/length.h>
#include <units/isq/si/time.h>

constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t);
```

a.cpp

```
#include <units/isq/si/speed.h>
#include "avg_speed.h"

using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

b.cpp

```
#include "avg_speed.h"

using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

ODR violation!

The Downcasting Facility: Issues

- Users reported issues with the **inability to define multiple copies** of custom scaled units

```
namespace device_1_specific {
  struct device_1_specific_unit : units::named_scaled_unit<...> {};
}

namespace device_2_specific {
  struct device_2_specific_unit : units::named_scaled_unit<...> {};
}
```

The Downcasting Facility: Issues

- Users reported issues with the **inability to define multiple copies** of custom scaled units

```
namespace device_1_specific {
  struct device_1_specific_unit : units::named_scaled_unit<...> {};
}

namespace device_2_specific {
  struct device_2_specific_unit : units::named_scaled_unit<...> {};
}
```

- The facility can recover the type identifier but **cannot recover variable names**, which limits NTTPs usage

Issues not addressable with the current framework

HARD TO UNDERSTAND UNKNOWN QUANTITY TYPES

```
auto bad = 1 / si::length<si::kilometre>(50);
```

```
units::quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, -1, 1> >,
units::scaled_unit<units::magnitude<units::base_power<long int>{2, units::ratio{-3, 1, 0}},
units::base_power<long int>{5, units::ratio{-3, 1, 0}}>(), units::unknown_coherent_unit>, double>
```

Issues not addressable with the current framework

HARD TO UNDERSTAND UNKNOWN QUANTITY TYPES

```
auto bad = 1 / si::length<si::kilometre>(50);
```

```
units::quantity<units::unknown_dimension<units::exponent<units::isq::si::dim_length, -1, 1> >,
units::scaled_unit<units::magnitude<units::base_power<long int>{2, units::ratio{-3, 1, 0}},
units::base_power<long int>{5, units::ratio{-3, 1, 0}}>(), units::unknown_coherent_unit>, double>
```

Initially types were shorter and easier to understand. Library extensions addressing some corner cases like radians to degrees conversion made types longer and harder to understand.

Issues not addressable with the current framework

DIMENSIONS ALWAYS CLOSELY CONNECTED TO COHERENT UNITS

```
namespace units::isq {

template<typename Child, Unit U, DimensionOfT<dim_length> L, DimensionOfT<dim_time> T>
struct dim_speed : derived_dimension<Child, U, exponent<L, 1>, exponent<T, -1>> {};

template<typename T>
concept Speed = QuantityOfT<T, dim_speed>;

}
```

Issues not addressable with the current framework

DIMENSIONS ALWAYS CLOSELY CONNECTED TO COHERENT UNITS

```
namespace units::isq {

template<typename Child, Unit U, DimensionOfT<dim_length> L, DimensionOfT<dim_time> T>
struct dim_speed : derived_dimension<Child, U, exponent<L, 1>, exponent<T, -1>> {};

template<typename T>
concept Speed = QuantityOfT<T, dim_speed>;

}
```

```
namespace units::isq::si {

struct metre_per_second : derived_unit<metre_per_second> {};
struct dim_speed : isq::dim_speed<dim_speed, metre_per_second, dim_length, dim_time> {};

}
```

Issues not addressable with the current framework

DIMENSIONS ALWAYS CLOSELY CONNECTED TO COHERENT UNITS

```
namespace units::isq {

template<typename Child, Unit U, DimensionOfT<dim_length> L, DimensionOfT<dim_time> T>
struct dim_speed : derived_dimension<Child, U, exponent<L, 1>, exponent<T, -1>> {};

template<typename T>
concept Speed = QuantityOfT<T, dim_speed>;

}
```

```
namespace units::isq::si {

struct metre_per_second : derived_unit<metre_per_second> {};
struct dim_speed : isq::dim_speed<dim_speed, metre_per_second, dim_length, dim_time> {};

}
```

- Problems with *dimension-specific concepts*
  - **`QuantityOf<isq::si::dim_speed>`** and **`QuantityOfT<isq::dim_speed>`** (P1985 will solve it)

Issues not addressable with the current framework

DIMENSIONS ALWAYS CLOSELY CONNECTED TO COHERENT UNITS

```
namespace units::isq {

template<typename Child, Unit U, DimensionOfT<dim_length> L, DimensionOfT<dim_time> T>
struct dim_speed : derived_dimension<Child, U, exponent<L, 1>, exponent<T, -1>> {};

template<typename T>
concept Speed = QuantityOfT<T, dim_speed>;

}
```

```
namespace units::isq::si {

struct metre_per_second : derived_unit<metre_per_second> {};
struct dim_speed : isq::dim_speed<dim_speed, metre_per_second, dim_length, dim_time> {};

}
```

- Problems with *dimension-specific concepts*
  - **`QuantityOf<isq::si::dim_speed>`** and **`QuantityOfT<isq::dim_speed>`** (P1985 will solve it)
- Standalone *dimensional analysis* not possible

Issues not addressable with the current framework

NOT COMPOSABLE UNITS

- Coherent unit of an angular momentum

```
struct kilogram_metre_sq_per_second : derived_unit<kilogram_metre_sq_per_second> {};
```

Issues not addressable with the current framework

NOT COMPOSABLE UNITS

- Coherent unit of an angular momentum

```
struct kilogram_metre_sq_per_second : derived_unit<kilogram_metre_sq_per_second> {};
```

Which scaled versions of such units to predefine in a library?

Issues not addressable with the current framework

NO WAY TO CORRECTLY ISOLATE QUANTITIES OF DIFFERENT KINDS

```
using namespace units::isq::iec80000::references;
using namespace units::isq::si::references;

std::cout << "1 Hz + 1 Bd = " << 1 * Hz + 1 * Bd << '\n';
```

Issues not addressable with the current framework

NO WAY TO CORRECTLY ISOLATE QUANTITIES OF DIFFERENT KINDS

```
using namespace units::isq::iec80000::references;
using namespace units::isq::si::references;

std::cout << "1 Hz + 1 Bd = " << 1 * Hz + 1 * Bd << '\n';
```

```
1 Hz + 1 Bd = 2 Hz
```

Issues not addressable with the current framework

NOT COMPOSABLE QUANTITY KINDS

```
struct length_kind : units::kind<length_kind, units::isq::si::dim_length> {};
struct width_kind : units::kind<width_kind, units::isq::si::dim_length> {};
```

Issues not addressable with the current framework

NOT COMPOSABLE QUANTITY KINDS

```
struct length_kind : units::kind<length_kind, units::isq::si::dim_length> {};
struct width_kind : units::kind<width_kind, units::isq::si::dim_length> {};
```

```
using length = units::quantity_kind<length_kind, units::isq::si::metre>;
using width = units::quantity_kind<width_kind, units::isq::si::metre>;
```

Issues not addressable with the current framework

NOT COMPOSABLE QUANTITY KINDS

```
struct length_kind : units::kind<length_kind, units::isq::si::dim_length> {};
struct width_kind : units::kind<width_kind, units::isq::si::dim_length> {};
```

```
using length = units::quantity_kind<length_kind, units::isq::si::metre>;
using width = units::quantity_kind<width_kind, units::isq::si::metre>;
```

```
using horizontal_area = ???
```

Why UDLs are not a good solution?

- Work only with **literals (compile-time known values)**

Why UDLs are not a good solution?

- Work only with **literals (compile-time known values)**
  - How many compile-time constants do you have in the code base beside unit tests?

Why UDLs are not a good solution?

- Work only with **literals (compile-time known values)**
  - How many compile-time constants do you have in the code base beside unit tests?
- What is the **representation type** of **`d1`** and **`d2`**?

```
using namespace std::chrono_literals;
auto d1 = 42s;
auto d2 = 42.s;
```

Why UDLs are not a good solution?

- Work only with **literals (compile-time known values)**
  - How many compile-time constants do you have in the code base beside unit tests?
- What is the **representation type** of **`d1`** and **`d2`**?

```
using namespace std::chrono_literals;
auto d1 = 42s;
auto d2 = 42.s;
```

```
static_assert(std::is_same_v<decltype(d1)::rep, std::int64_t>);
static_assert(std::is_same_v<decltype(d2)::rep, long double>);
```

Why UDLs are not a good solution?

- Work only with literals (compile-time known values)
  - How many compile-time constants do you have in the code base beside unit tests?
- What is the representation type of **`d1`** and **`d2`**?

```
using namespace std::chrono_literals;
auto d1 = 42s;
auto d2 = 42.s;
```

```
static_assert(std::is_same_v<decltype(d1)::rep, std::int64_t>);
static_assert(std::is_same_v<decltype(d2)::rep, long double>);
```

- Many literal identifiers (**`F`**, **`J`**, **`W`**, **`K`**, **`d`**, **`l`**, **`L`**, ...) are already reserved by compilers
  - often for non-standard extensions
  - we had to introduce **`_q_`** prefix to avoid collisions

Why UDLs are not a good solution?

- If both SI and CGS systems define **`_q_s`** UDL for a second unit, then **how to disambiguate** them in a common source code?

Why UDLs are not a good solution?

- If both SI and CGS systems define **_q_s** UDL for a second unit, then **how to disambiguate** them in a common source code?
- **Don't compose**

```
auto q = 42_q_kg_m2_per_s;
```

Why UDLs are not a good solution?

- If both SI and CGS systems define **_q_s** UDL for a second unit, then **how to disambiguate** them in a common source code?
- **Don't compose**

```
auto q = 42_q_kg_m2_per_s;
```

- Require **two definitions per each unit**

```
constexpr auto operator"" _q_kg_m2_per_s(unsigned long long l)
{
  gsl_ExpectsAudit(std::in_range<std::int64_t>(l));
  return angular_momentum<kilogram_metre_sq_per_second, std::int64_t>(static_cast<std::int64_t>(l));
}

constexpr auto operator"" _q_kg_m2_per_s(long double l)
{
  return angular_momentum<kilogram_metre_sq_per_second, long double>(l);
}
```

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

SOLUTION #2: UNIT ALIASES

```
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

```
auto d = km(123.);
auto v = km_per_h(70);
```

SOLUTION #3: QUANTITY REFERENCES

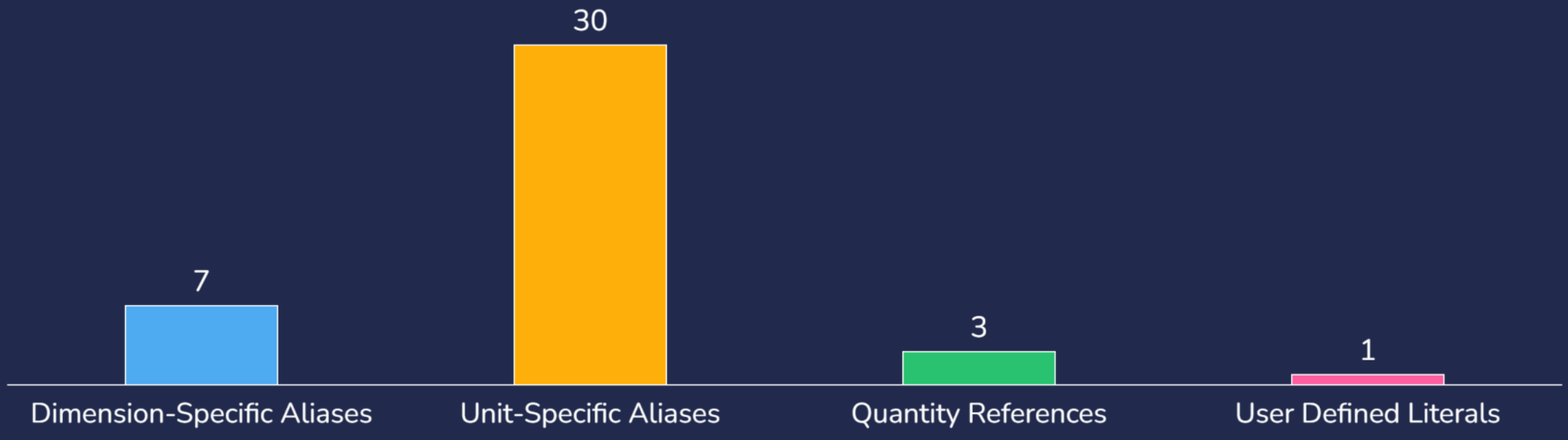
```
auto d = 123. * km;
auto v = 70 * (km / h);
```

SOLUTION #4: UDLS

```
auto d = 123.q_km;
auto v = 70q_km_per_h;
```

AhaSlides

Choose the best solution



Submission locked Unlock

Menu

40 0/200

Verbose to define (and standardize) systems

Even though the library definitions are terse comparing to other products on the market that use macros to provide multiple definitions per one entity, there is still some room for improvement.

Verbose to define (and standardize) systems

```
namespace units::isq::si {

struct kilogram_metre_sq_per_second : derived_unit<kilogram_metre_sq_per_second> {};
struct dim_angular_momentum :
    isq::dim_angular_momentum<dim_angular_momentum, kilogram_metre_sq_per_second, dim_length, dim_momentum> {};

// ...

inline namespace literals {

constexpr auto operator"" _q_kg_m2_per_s(unsigned long long l) {
  gsl_ExpectsAudit(std::in_range<std::int64_t>(l));
  return angular_momentum<kilogram_metre_sq_per_second, std::int64_t>(static_cast<std::int64_t>(l));
}
constexpr auto operator"" _q_kg_m2_per_s(long double l) {
  return angular_momentum<kilogram_metre_sq_per_second, long double>(l);
}

}  // namespace literals
}  // namespace units::isq::si

namespace units::aliases::isq::si::inline angular_momentum {

template<Representation Rep = double>
using kg_m2_per_s = units::isq::si::angular_momentum<units::isq::si::kilogram_metre_sq_per_second, Rep>;

}  // namespace units::aliases::isq::si::inline angular_momentum
```

MP-UNITS V2

(WIP)

Why V2 if V1 was not released?

- The new design is **not a gradual evolutionary change**
  - it *feels like a totally different product*
- A **huge breaking change** for the users
  - abstractions
  - interfaces
  - namespace name
  - header files content and layout

```cpp
#include <mp-units/format.h>
#include <mp-units/systems/isq/isq.h>
#include <mp-units/systems/si/si.h>
#include <format>

using namespace mp_units;
using namespace mp_units::si::unit_symbols;

quantity<isq::speed[m / s]> avg_speed(quantity<si::metre> d,
                                      quantity<si::second> t)
{ return d / t; }

int main()
{
  auto speed = avg_speed(220 * km, 2 * h);
  std::println("{}", speed);  // 30.5556 m/s
}
```

Why V2 if V1 was not released?

- The new design is **not a gradual evolutionary change**
  - it *feels like a totally different product*
- A **huge breaking change** for the users
  - abstractions
  - interfaces
  - namespace name
  - header files content and layout

```cpp
#include <mp-units/format.h>
#include <mp-units/systems/isq/isq.h>
#include <mp-units/systems/si/si.h>
#include <format>

using namespace mp_units;
using namespace mp_units::si::unit_symbols;

quantity<isq::speed[m / s]> avg_speed(quantity<si::metre> d,
                                      quantity<si::second> t)
{ return d / t; }

int main()
{
  auto speed = avg_speed(220 * km, 2 * h);
  std::println("{}", speed);  // 30.5556 m/s
}
```

We could start a new project or switch directly to V2 version to emphasize the scope of changes and breakage.

Making the compilers sweat

Even though the library is based on C++20, still in 2023, many compilers are not fully conforming.

- As of today the library works only on gcc-12.2+

Making the compilers sweat

Even though the library is based on C++20, still in 2023, many compilers are not fully conforming.

- As of today the library works only on gcc-12.2+

In case you are a compiler vendor and would like to work with us to improve your compiler's implementation, please let us know and we will work together to solve the issues.

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code
- To *improve compiler errors readability* a new idiom in the library is to **use the same identifier for a type and its instance**

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code
- To *improve compiler errors readability* a new idiom in the library is to **use the same identifier for a type and its instance**

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

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code
- To *improve compiler errors readability* a new idiom in the library is to **use the same identifier for a type and its instance**

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

```
quantity<metre / second> q;
```

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code
- To *improve compiler errors readability* a new idiom in the library is to **use the same identifier for a type and its instance**

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

```
quantity<metre / second> q;
```

```
(gdb) ptype q
type = class quantity<derived_unit<metre, per<second>>(), double> [with Rep = double] {
```

New style of definitions

- **Users always work with objects** but then **observe their type representation** in the generated code
- To *improve compiler errors readability* a new idiom in the library is to **use the same identifier for a type and its instance**

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

```
quantity<metre / second> q;
```

```
(gdb) ptype q
type = class quantity<derived_unit<metre, per<second>>(), double> [with Rep = double] {
```

Expressions templates are extensively used through the library to improve the readability of resulting types.

Physical quantity libraries enforce correct code

```
speed avg_speed(length l, time t)
{
  return l / t;
}
```

```
auto res = avg_speed(120 * km, 2 * h);
```

Physical quantity libraries enforce correct code

```
speed avg_speed(length l, time t)
{
  return l / t;
}
```

```
auto res = avg_speed(120 * km, 2 * h);
```

- Strongly typed function arguments can't be reordered
- Scaling between various units automatically handled by the library engine
- Compile-time-enforced dimensional analysis helps writing correct equations

Physical quantity libraries enforce correct code

```
speed avg_speed(length l, time t)
{
  return l / t;
}
```

```
auto res = avg_speed(120 * km, 2 * h);
```

- Strongly typed function arguments **can't be reordered**
- **Scaling between various units** automatically handled by the library engine
- Compile-time-enforced **dimensional analysis** helps writing correct equations

No runtime overhead! As fast or even faster than working directly on fundamental types.

How do you like such an interface?

```
class Box {
  area base_;
  length height_;
public:
  Box(length l, length w, length h) : base_(l * w), height_(h) {}
  // ...
};
```

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

How do you like such an interface?

```
class Box {
  area base_;
  length height_;
public:
  Box(length l, length w, length h) : base_(l * w), height_(h) {}
  // ...
};
```

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

As long as quantities of different dimensions guarantee type safety in most of the libraries on the market, nearly none of them provide any help for different quantities of the same kind.

Proper systems abstractions (ISO/IEC Guide 99:2007)

Proper systems abstractions (ISO/IEC Guide 99:2007)

- **System of Quantities**
  - *set of quantities* together with a *set of* non-contradictory *equations* relating those quantities

Proper systems abstractions (ISO/IEC Guide 99:2007)

- **System of Quantities**
  - *set of quantities* together with a *set of* non-contradictory *equations* relating those quantities
- **International System of Quantities (ISQ)**
  - system of quantities based on the *seven base quantities*: length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity
  - this system of quantities is *published in the ISO 80000 and IEC 80000* series "Quantities and units"

Proper systems abstractions (ISO/IEC Guide 99:2007)

- **System of Quantities**
  - *set of quantities* together with a *set of* non-contradictory *equations* relating those quantities
- **International System of Quantities (ISQ)**
  - system of quantities based on the *seven base quantities*: length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity
  - this system of quantities is *published in the ISO 80000 and IEC 80000* series "Quantities and units"
- **International System of Units (SI)**
  - system of units, their names and symbols
  - including a series of prefixes and their names and symbols, together with rules for their use
  - *based on the International System of Quantities*
  - adopted by the General Conference on Weights and Measures (CGPM)

ISQ in mp-units

mp-units is probably the first library on the Open Source market (in any programming language) that models the ISQ with all its definitions provided in ISO 80000.

ISQ in mp-units

mp-units is probably the first library on the Open Source market (in any programming language) that models the ISQ with all its definitions provided in ISO 80000.

Please provide feedback if something looks odd or could be improved.

ISQ base dimensions

```
namespace mp_units::isq {

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

}
```

ISQ base dimensions

```
namespace mp_units::isq {

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

}
```

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

ISQ base dimensions

```
namespace mp_units::isq {

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

}
```

NTTPs are awesome! :-D

Introducing `quantity_spec`

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

Introducing `quantity_spec`

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

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

Introducing `quantity_spec`

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

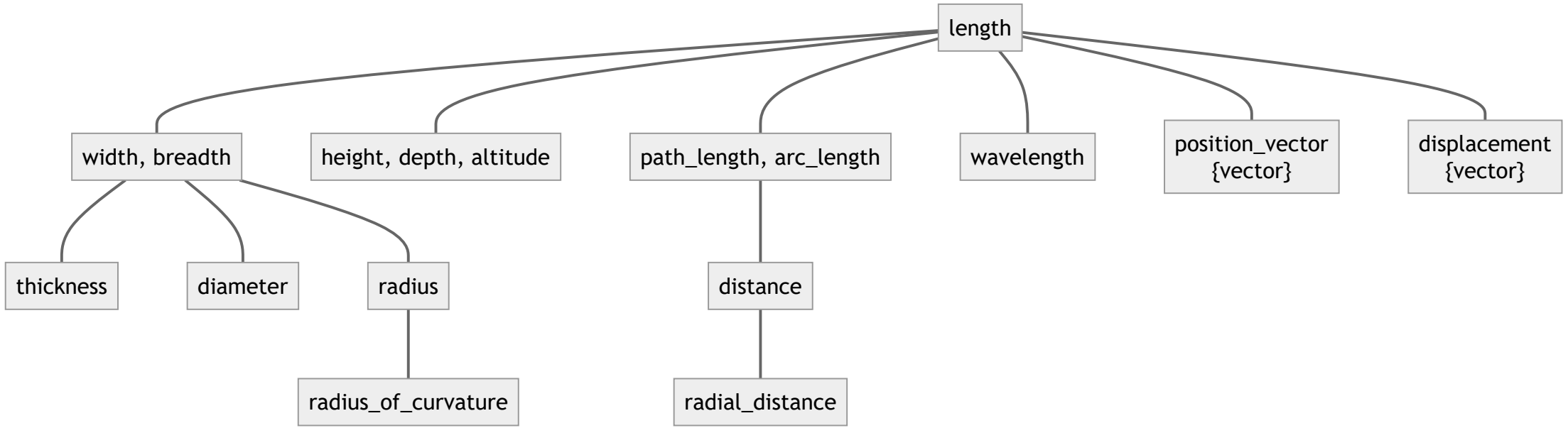
- **More than one quantity** may be defined **for the same dimension**
  - quantities of *different kinds* (i.e. frequency, modulation rate, activity, ...)
  - quantities of the *same kind* (i.e. length, width, altitude, distance, radius, wavelength, position vector, ...)
- Quantities may have **different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

Introducing `quantity_spec`

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

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

ISQ: International System of Quantities (ISO 80000)



Defining a tree of quantities of the same kind

- A quantity being the root for the tree of quantities of the same kind

```
inline constexpr struct length : quantity_spec<dim_length> {} length;
```

Defining a tree of quantities of the same kind

- A quantity being the root for the tree of quantities of the same kind

```
inline constexpr struct length : quantity_spec<dim_length> {} length;
```

- Quantity forming a leaf in a tree of quantities of the same kind

```
inline constexpr struct width : quantity_spec<length> {} width;
```

Defining a tree of quantities of the same kind

- A quantity being the root for the tree of quantities of the same kind

```
inline constexpr struct length : quantity_spec<dim_length> {} length;
```

- Quantity forming a leaf in a tree of quantities of the same kind

```
inline constexpr struct width : quantity_spec<length> {} width;
```

- Different name for the same quantity type (alias)

```
inline constexpr auto breadth = width;
```

Defining quantities of dimension length and time

```
inline constexpr struct length : quantity_spec<dim_length> {} length;
inline constexpr struct width : quantity_spec<length> {} width;
inline constexpr auto breadth = width;
inline constexpr struct height : quantity_spec<length> {} height;
inline constexpr auto depth = height;
inline constexpr auto altitude = height;
inline constexpr struct thickness : quantity_spec<width> {} thickness;
inline constexpr struct diameter : quantity_spec<width> {} diameter;
inline constexpr struct radius : quantity_spec<width> {} radius;
inline constexpr struct radius_of_curvature : quantity_spec<radius> {} radius_of_curvature;
inline constexpr struct path_length : quantity_spec<length> {} path_length;
inline constexpr auto arc_length = path_length;
inline constexpr struct distance : quantity_spec<path_length> {} distance;
inline constexpr struct radial_distance : quantity_spec<distance> {} radial_distance;
inline constexpr struct wavelength : quantity_spec<length> {} wavelength;
inline constexpr struct position_vector : quantity_spec<length, quantity_character::vector> {} position_vector;
inline constexpr struct displacement : quantity_spec<length, quantity_character::vector> {} displacement;
```

Defining quantities of dimension length and time

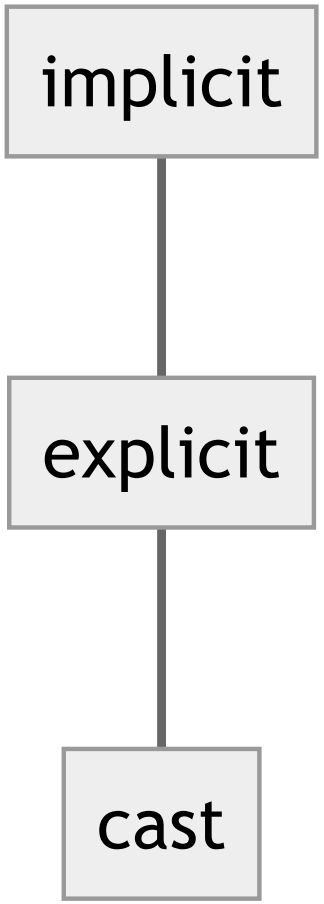
```
inline constexpr struct length : quantity_spec<dim_length> {} length;
inline constexpr struct width : quantity_spec<length> {} width;
inline constexpr auto breadth = width;
inline constexpr struct height : quantity_spec<length> {} height;
inline constexpr auto depth = height;
inline constexpr auto altitude = height;
inline constexpr struct thickness : quantity_spec<width> {} thickness;
inline constexpr struct diameter : quantity_spec<width> {} diameter;
inline constexpr struct radius : quantity_spec<width> {} radius;
inline constexpr struct radius_of_curvature : quantity_spec<radius> {} radius_of_curvature;
inline constexpr struct path_length : quantity_spec<length> {} path_length;
inline constexpr auto arc_length = path_length;
inline constexpr struct distance : quantity_spec<path_length> {} distance;
inline constexpr struct radial_distance : quantity_spec<distance> {} radial_distance;
inline constexpr struct wavelength : quantity_spec<length> {} wavelength;
inline constexpr struct position_vector : quantity_spec<length, quantity_character::vector> {} position_vector;
inline constexpr struct displacement : quantity_spec<length, quantity_character::vector> {} displacement;
```

```
inline constexpr struct time : quantity_spec<dim_time> time;
inline constexpr auto duration = time;
inline constexpr struct period_duration : quantity_spec<duration> period_duration;
inline constexpr auto period = period_duration;
inline constexpr struct time_constant : quantity_spec<duration> time_constant;
```

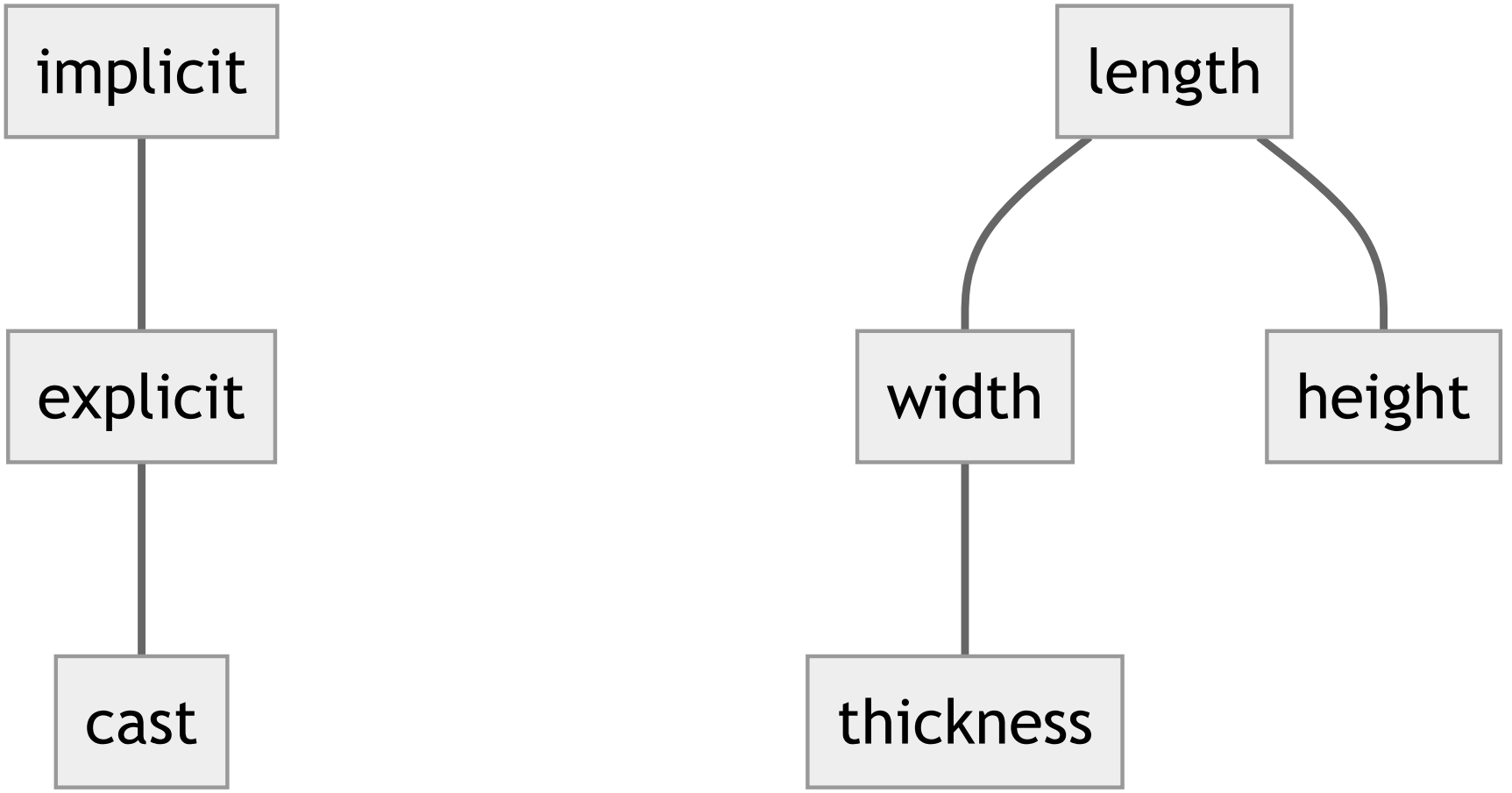
Removing abstractions

Proper implementation of the ISQ hierarchies superseded **`quantity_kind`** and **`quantity_point_kind`** class templates from the V1 framework.

Three levels of quantity conversions



Three levels of quantity conversions



Three levels of quantity conversions

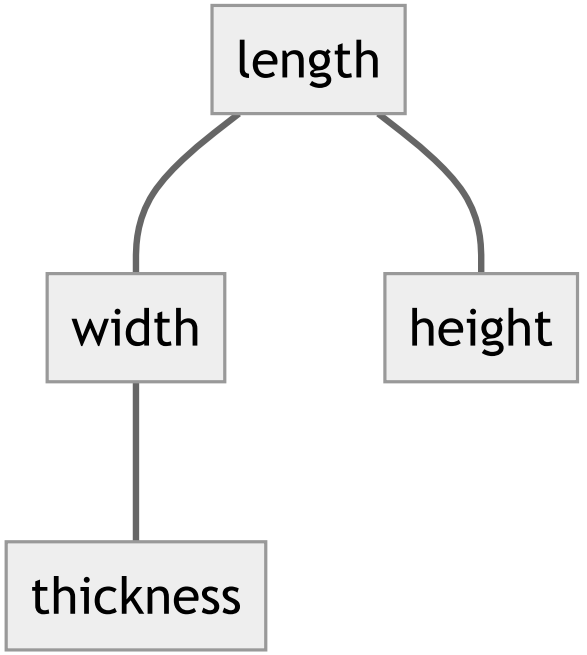
```
static_assert(isq::width != isq::length);
static_assert(isq::width != isq::height);

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

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

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

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

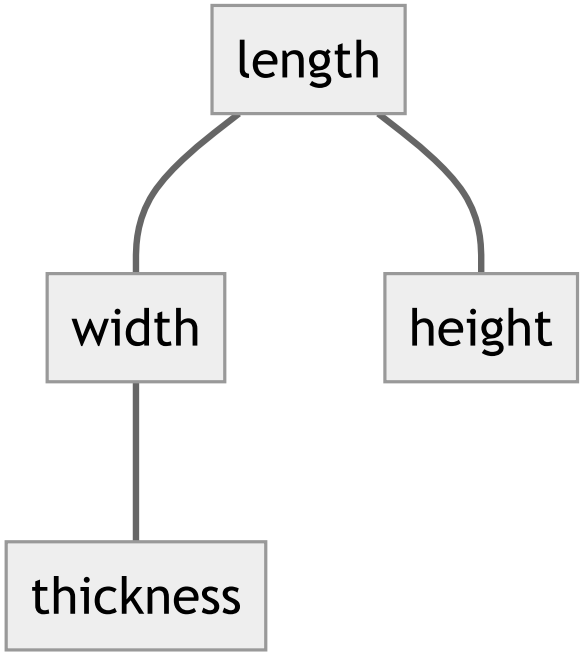


Common quantity specification

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

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

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



Derived quantities

```
inline constexpr struct area : quantity_spec<pow<2>(length)> area;
inline constexpr struct volume : quantity_spec<pow<3>(length)> volume;
inline constexpr struct speed : quantity_spec<length / time> speed;
inline constexpr struct velocity : quantity_spec<speed, position_vector / duration> velocity;
inline constexpr struct acceleration : quantity_spec<velocity / duration> acceleration;
inline constexpr struct frequency : quantity_spec<1 / period_duration> frequency;
```

NTTPs are awesome! :-D

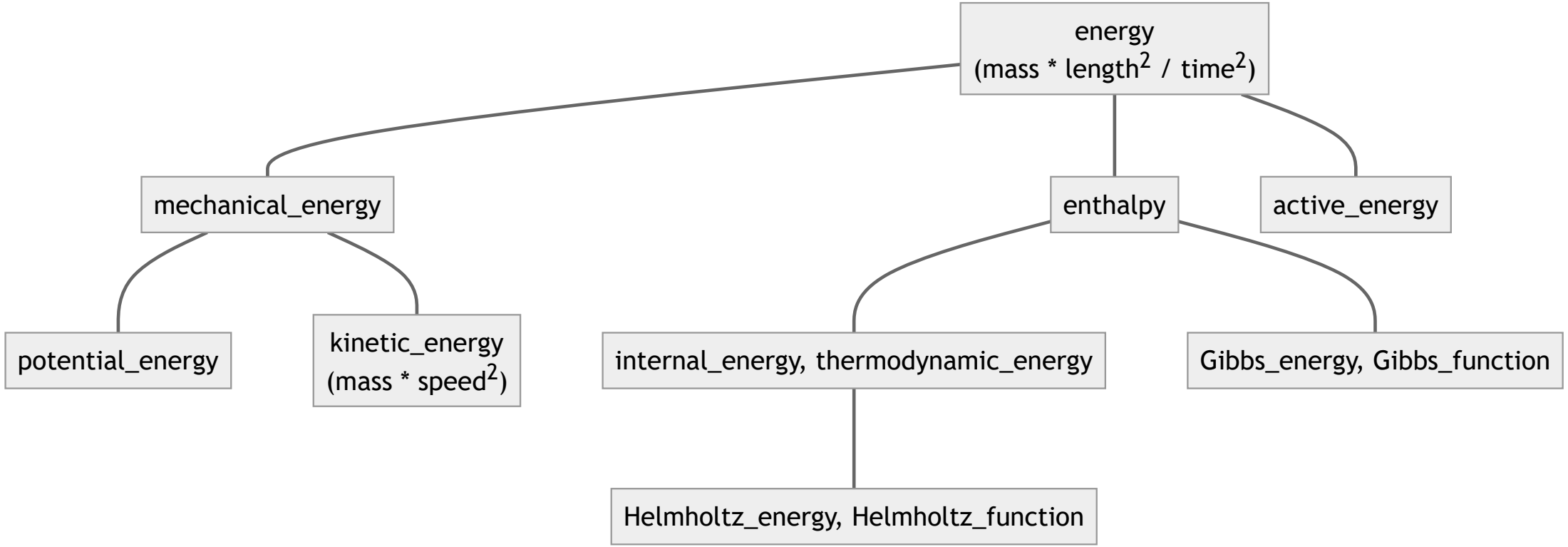
Derived quantities

```
inline constexpr struct area : quantity_spec<pow<2>(length)> area;
inline constexpr struct volume : quantity_spec<pow<3>(length)> volume;
inline constexpr struct speed : quantity_spec<length / time> speed;
inline constexpr struct velocity : quantity_spec<speed, position_vector / duration> velocity;
inline constexpr struct acceleration : quantity_spec<velocity / duration> acceleration;
inline constexpr struct frequency : quantity_spec<1 / period_duration> frequency;
```

PURE DIMENSIONAL ANALYSIS

```
static_assert(isq::area.dimension == pow<2>(isq::dim_length));
static_assert(isq::speed.dimension == isq::dim_length / isq::dim_time);
static_assert(isq::acceleration.dimension == isq::velocity.dimension / isq::dim_time);
static_assert(isq::frequency.dimension == dimension_one / isq::dim_time);
```

Derived quantity equations often do not automatically form a hierarchy tree



Defining a hierarchy of derived quantities

```
inline constexpr struct energy : quantity_spec<mass * pow<2>(length) / pow<2>(time)> {} energy;
inline constexpr struct mechanical_energy : quantity_spec<energy> {} mechanical_energy;
inline constexpr struct potential_energy : quantity_spec<mechanical_energy> {} potential_energy;
inline constexpr struct kinetic_energy : quantity_spec<mechanical_energy, mass * pow<2>(speed)> {} kinetic_energy;
```

- First template parameter **defines the hierarchy tree**

Defining a hierarchy of derived quantities

```
inline constexpr struct energy : quantity_spec<mass * pow<2>(length) / pow<2>(time)> {} energy;
inline constexpr struct mechanical_energy : quantity_spec<energy> {} mechanical_energy;
inline constexpr struct potential_energy : quantity_spec<mechanical_energy> {} potential_energy;
inline constexpr struct kinetic_energy : quantity_spec<mechanical_energy, mass * pow<2>(speed)> {} kinetic_energy;
```

- First template parameter **defines the hierarchy tree**
- Second parameter (if provided) **constraints the recipe**

Defining a hierarchy of derived quantities

```
inline constexpr struct energy : quantity_spec<mass * pow<2>(length) / pow<2>(time)> {} energy;
inline constexpr struct mechanical_energy : quantity_spec<energy> {} mechanical_energy;
inline constexpr struct potential_energy : quantity_spec<mechanical_energy> {} potential_energy;
inline constexpr struct kinetic_energy : quantity_spec<mechanical_energy, mass * pow<2>(speed)> {} kinetic_energy;
```

- First template parameter **defines the hierarchy tree**
- Second parameter (if provided) **constraints the recipe**

Concepts are awesome! :-D

Conversions of quantity equations to derived quantities

```
constexpr auto qs = isq::mass * pow<2>(isq::length) / pow<2>(isq::time);
```

Conversions of quantity equations to derived quantities

```
constexpr auto qs = isq::mass * pow<2>(isq::length) / pow<2>(isq::time);
```

- The result of a quantity equation is not the same as a derived quantity

```
static_assert(qs != isq::energy);
```

Conversions of quantity equations to derived quantities

```
constexpr auto qs = isq::mass * pow<2>(isq::length) / pow<2>(isq::time);
```

- The result of a quantity equation is **not the same** as a derived quantity

```
static_assert(qs != isq::energy);
```

- If the derived quantity **provides a recipe** that is compatible with the result of a quantity equation, then such a result is **implicitly convertible** to this derived quantity

```
static_assert(implicitly_convertible(qs, isq::energy));
static_assert(implicitly_convertible(isq::mass * pow<2>(isq::speed), isq::kinetic_energy));
```

Conversions of quantity equations to derived quantities

```
constexpr auto qs = isq::mass * pow<2>(isq::length) / pow<2>(isq::time);
```

- The result of a quantity equation is **not the same** as a derived quantity

```
static_assert(qs != isq::energy);
```

- If the derived quantity **provides a recipe** that is compatible with the result of a quantity equation, then such a result is **implicitly convertible** to this derived quantity

```
static_assert(implicitly_convertible(qs, isq::energy));
static_assert(implicitly_convertible(isq::mass * pow<2>(isq::speed), isq::kinetic_energy));
```

- Otherwise, an **explicit conversion is needed**

```
static_assert(!implicitly_convertible(qs, isq::mechanical_energy));
static_assert(explicitly_convertible(qs, isq::mechanical_energy));
```

Simple user extensions and composability

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

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

Simple user extensions and composability

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

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

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

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

Simple user extensions and composability

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

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

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

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

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

Simple user extensions and composability

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

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

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

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

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

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

What should be the result of the following equation?

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

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

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

What should be the result of the following equation?

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

Boost.Units

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

What should be the result of the following equation?

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

Boost.Units

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

2 Hz

What should be the result of the following equation?

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

Boost.Units

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

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

2 Hz

What should be the result of the following equation?

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

Boost.Units

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

2 Hz

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

2 Hz

What should be the result of the following equation?

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

Boost.Units

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

2 Hz

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

2 Hz

nholthaus/units

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

What should be the result of the following equation?

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

Boost.Units

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

```
2 Hz
```

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

```
2 Hz
```

nholthaus/units

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

```
2 s^-1
```

What should be the result of the following equation?

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

Pint

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

What should be the result of the following equation?

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

Pint

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

```
3.0 Hz
```

What should be the result of the following equation?

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

Pint

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

```
3.0 Hz
```

JSR 385

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

What should be the result of the following equation?

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

Pint

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

```
3.0 Hz
```

JSR 385

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

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

Quantity kinds (ISO 80000)

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

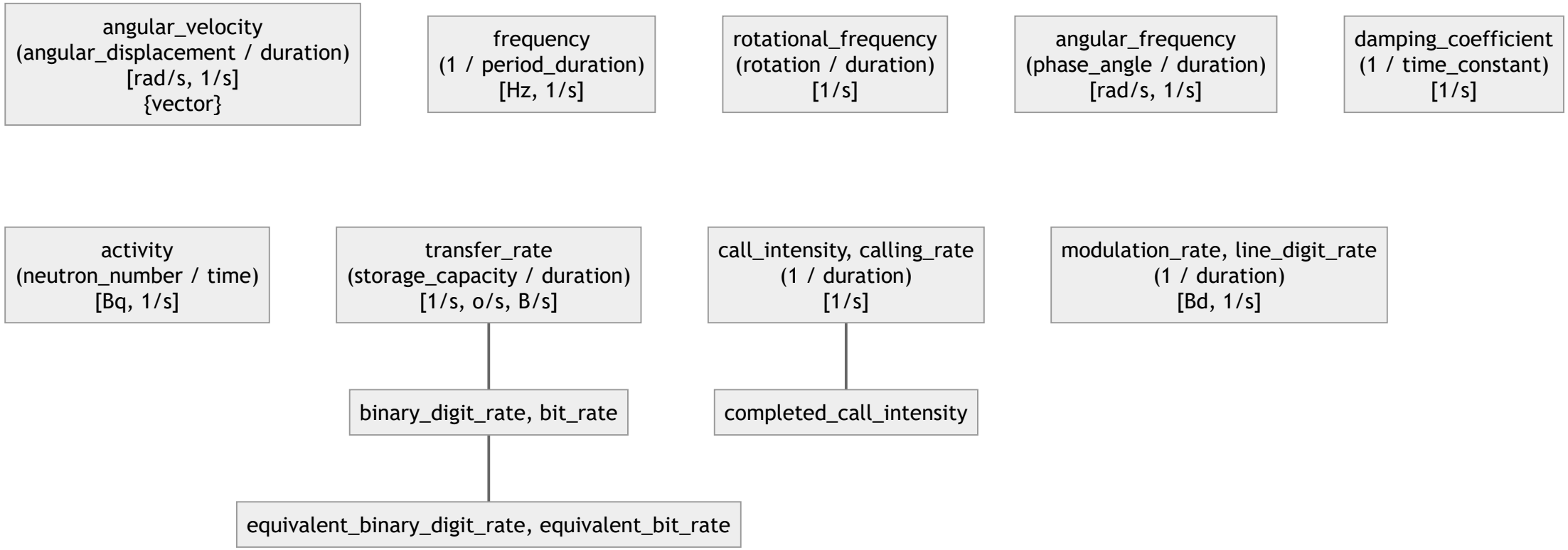
Quantity kinds (ISO 80000)

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

Quantity kinds (ISO 80000)

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

ISQ quantities of dimension T^{-1}



`kind_of<QS>` modifier

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

`kind_of<QS>` modifier

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

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

`kind_of<QS>` modifier

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

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

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

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

`kind_of<QS>` modifier

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

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

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

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

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

SI units for ISQ base quantities

```
namespace mp_units::si {

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

}
```

SI units for ISQ base quantities

```
namespace mp_units::si {

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

}
```

All SI units store the information about associated quantity kind.

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian : named_unit<"sr", square(metre) / square(metre),
                                               kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz : named_unit<"Hz", 1 / second, kind_of<isq::frequency>> {} hertz;
inline constexpr struct newton : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian : named_unit<"sr", square(metre) / square(metre),
                                               kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz : named_unit<"Hz", 1 / second, kind_of<isq::frequency>> {} hertz;
inline constexpr struct newton : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

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

SI named derived units

```
namespace mp_units::si {

inline constexpr struct radian : named_unit<"rad", metre / metre, kind_of<isq::angular_measure>> {} radian;
inline constexpr struct steradian : named_unit<"sr", square(metre) / square(metre),
                                               kind_of<isq::solid_angular_measure>> {} steradian;
inline constexpr struct hertz : named_unit<"Hz", 1 / second, kind_of<isq::frequency>> {} hertz;
inline constexpr struct newton : named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct pascal : named_unit<"Pa", newton / square(metre)> {} pascal;
inline constexpr struct joule : named_unit<"J", newton * metre> {} joule;
inline constexpr struct watt : named_unit<"W", joule / second> {} watt;
inline constexpr struct coulomb : named_unit<"C", ampere * second> {} coulomb;
// ...

}
```

Did I already mentioned that concepts and NTTPs are awesome? :-D

Class Types in 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.

Unit symbols

```
namespace mp_units::si::unit_symbols {

inline constexpr auto qm = quecto<metre>;
inline constexpr auto rm = ronto<metre>;
inline constexpr auto ym = yocto<metre>;
inline constexpr auto zm = zepto<metre>;
inline constexpr auto am = atto<metre>;
inline constexpr auto fm = femto<metre>;
inline constexpr auto pm = pico<metre>;
inline constexpr auto nm = nano<metre>;
inline constexpr auto um = micro<metre>;
inline constexpr auto mm = milli<metre>;
inline constexpr auto cm = centi<metre>;
inline constexpr auto dm = deci<metre>;
inline constexpr auto m = metre;
```

```
inline constexpr auto dam = deca<metre>;
inline constexpr auto hm = hecto<metre>;
inline constexpr auto km = kilo<metre>;
inline constexpr auto Mm = mega<metre>;
inline constexpr auto Gm = giga<metre>;
inline constexpr auto Tm = tera<metre>;
inline constexpr auto Pm = peta<metre>;
inline constexpr auto Em = exa<metre>;
inline constexpr auto Zm = zetta<metre>;
inline constexpr auto Ym = yotta<metre>;
inline constexpr auto Rm = ronna<metre>;
inline constexpr auto Qm = quetta<metre>;

// ...
}
```

Unit symbols

```
namespace mp_units::si::unit_symbols {

inline constexpr auto qm = quecto<metre>;
inline constexpr auto rm = ronto<metre>;
inline constexpr auto ym = yocto<metre>;
inline constexpr auto zm = zepto<metre>;
inline constexpr auto am = atto<metre>;
inline constexpr auto fm = femto<metre>;
inline constexpr auto pm = pico<metre>;
inline constexpr auto nm = nano<metre>;
inline constexpr auto um = micro<metre>;
inline constexpr auto mm = milli<metre>;
inline constexpr auto cm = centi<metre>;
inline constexpr auto dm = deci<metre>;
inline constexpr auto m = metre;
```

```
inline constexpr auto dam = deca<metre>;
inline constexpr auto hm = hecto<metre>;
inline constexpr auto km = kilo<metre>;
inline constexpr auto Mm = mega<metre>;
inline constexpr auto Gm = giga<metre>;
inline constexpr auto Tm = tera<metre>;
inline constexpr auto Pm = peta<metre>;
inline constexpr auto Em = exa<metre>;
inline constexpr auto Zm = zetta<metre>;
inline constexpr auto Ym = yotta<metre>;
inline constexpr auto Rm = ronna<metre>;
inline constexpr auto Qm = quetta<metre>;

// ...
}
```

Unit symbols usage is opt-in to limit collisions with user-provided identifiers.

Unit symbols

```
namespace mp_units::si::unit_symbols {

inline constexpr auto qm = quecto<metre>;
inline constexpr auto rm = ronto<metre>;
inline constexpr auto ym = yocto<metre>;
inline constexpr auto zm = zepto<metre>;
inline constexpr auto am = atto<metre>;
inline constexpr auto fm = femto<metre>;
inline constexpr auto pm = pico<metre>;
inline constexpr auto nm = nano<metre>;
inline constexpr auto um = micro<metre>;
inline constexpr auto mm = milli<metre>;
inline constexpr auto cm = centi<metre>;
inline constexpr auto dm = deci<metre>;
inline constexpr auto m = metre;
```

```
inline constexpr auto dam = deca<metre>;
inline constexpr auto hm = hecto<metre>;
inline constexpr auto km = kilo<metre>;
inline constexpr auto Mm = mega<metre>;
inline constexpr auto Gm = giga<metre>;
inline constexpr auto Tm = tera<metre>;
inline constexpr auto Pm = peta<metre>;
inline constexpr auto Em = exa<metre>;
inline constexpr auto Zm = zetta<metre>;
inline constexpr auto Ym = yotta<metre>;
inline constexpr auto Rm = ronna<metre>;
inline constexpr auto Qm = quetta<metre>;

// ...
}
```

Some SI prefixes are not possible to be implemented with **`std::ratio`**.

Quantity

Quantity - property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a **number** and a **reference**. A reference can be a **measurement unit**, ...

-- ISO/IEC Guide 99:2007

Quantity

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









  // ...
};
```

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;









  // ...
};
```

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;
  Rep number_;









  // ...
};
```

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;
  Rep number_;

  // ...
};
```

Current structural type requirements suck :-(

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;
  Rep number_;

  [[nodiscard]] constexpr const Rep& number() const& noexcept { return number_; }
  [[nodiscard]] constexpr Rep&& number() && noexcept { return std::move(number_); }
  [[nodiscard]] constexpr const Rep&& number() const&& noexcept { return std::move(number_); }




  // ...
};
```

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;
  Rep number_;

  [[nodiscard]] constexpr const Rep& number() const& noexcept { return number_; }
  [[nodiscard]] constexpr Rep&& number() && noexcept { return std::move(number_); }
  [[nodiscard]] constexpr const Rep&& number() const&& noexcept { return std::move(number_); }

  template<Unit U>
  [[nodiscard]] constexpr Rep number_in(U) const noexcept;

  // ...
};
```

Quantity

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R).character> Rep = double>
class quantity {
public:
  static constexpr Reference auto reference = R;
  Rep number_;

  [[nodiscard]] constexpr const Rep& number() const& noexcept { return number_; }
  [[nodiscard]] constexpr Rep&& number() && noexcept { return std::move(number_); }
  [[nodiscard]] constexpr const Rep&& number() const&& noexcept { return std::move(number_); }

  template<Unit U>
  [[nodiscard]] constexpr Rep number_in(U) const noexcept;

  // ...
};
```

Concepts are awesome! :-D

Creating a quantity

- By definition

```
quantity<si::metre> q1;
quantity<si::metre, int> q2;
```

```
quantity<m> q1;
quantity<m, int> q2;
```

Creating a quantity

- By definition

```
quantity<si::metre> q1;
quantity<si::metre, int> q2;
```

```
quantity<m> q1;
quantity<m, int> q2;
```

- With multiply syntax

```
auto q1 = 42. * si::metre;
auto q2 = 42 * si::metre;
```

```
auto q1 = 42. * m;
auto q2 = 42 * m;
```

Creating a quantity

- By **definition**

```
quantity<si::metre> q1;
quantity<si::metre, int> q2;
```

```
quantity<m> q1;
quantity<m, int> q2;
```

- With **multiply syntax**

```
auto q1 = 42. * si::metre;
auto q2 = 42 * si::metre;
```

```
auto q1 = 42. * m;
auto q2 = 42 * m;
```

- **Mixed**

```
quantity<si::metre> q1(42. * m);
quantity<si::metre, int> q2(42 * m);
```

```
quantity<si::metre> q1 = 42. * m;
quantity<si::metre, int> q2 = 42 * m;
```

Creating a quantity

- By definition

```
quantity<si::metre> q1;
quantity<si::metre, int> q2;
```

```
quantity<m> q1;
quantity<m, int> q2;
```

- With multiply syntax

```
auto q1 = 42. * si::metre;
auto q2 = 42 * si::metre;
```

```
auto q1 = 42. * m;
auto q2 = 42 * m;
```

- Mixed

```
quantity<si::metre> q1(42. * m);
quantity<si::metre, int> q2(42 * m);
```

```
quantity<si::metre> q1 = 42. * m;
quantity<si::metre, int> q2 = 42 * m;
```

```
quantity<si::metre> q1(42. * km);
quantity<si::metre, int> q2(42 * km);
```

```
quantity<si::metre> q1 = 42. * km;
quantity<si::metre, int> q2 = 42 * km;
```

`explicit` is not explicit enough

TODAY

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

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

`explicit` is not explicit enough

TODAY

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

TOMORROW

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

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

`explicit` is not explicit enough

In V2 a **`quantity`** class template has no publicly available constructor taking a raw value.

What about `quantity_spec`?

What about `quantity_spec`?

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

What about `quantity_spec`?

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

```
static_assert((42 * m).quantity_spec == kind_of<isq::length>);
static_assert((60 * (km / h)).quantity_spec == kind_of<isq::length / isq::time>);
```

What about `quantity_spec`?

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

```
static_assert((42 * m).quantity_spec == kind_of<isq::length>);
static_assert((60 * (km / h)).quantity_spec == kind_of<isq::length / isq::time>);
```

Quantities defined only in terms of units model a quantity of a kind associated with such unit.

What about `quantity_spec`?

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

```
static_assert((42 * m).quantity_spec == kind_of<isq::length>);
static_assert((60 * (km / h)).quantity_spec == kind_of<isq::length / isq::time>);
```

- Quantity-type-aware reference

```
static_assert((42 * isq::height[m]).quantity_spec == isq::height);
static_assert((60 * isq::speed[km / h]).quantity_spec == isq::speed);
```

What about `quantity_spec`?

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

```
static_assert((42 * m).quantity_spec == kind_of<isq::length>);
static_assert((60 * (km / h)).quantity_spec == kind_of<isq::length / isq::time>);
```

- Quantity-type-aware reference

```
static_assert((42 * isq::height[m]).quantity_spec == isq::height);
static_assert((60 * isq::speed[km / h]).quantity_spec == isq::speed);
```

- Conversion of a kind to a specific quantity type

```
static_assert(isq::height(42 * m).quantity_spec == isq::height);
static_assert(isq::speed(60 * (km / h)).quantity_spec == isq::speed);
```

Quantity reference can be more than just a unit

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

Quantity reference can be more than just a unit

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

Quantity reference can be more than just a unit

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

A user never has to type **`reference`** class template instantiation by hand.

How does it work without CRTP?

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

```
inline constexpr struct height : quantity_spec<length> {} height;
```

How does it work without CRTP?

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

```
inline constexpr struct height : quantity_spec<length> {} height;
```

```
template<NamedQuantitySpec auto Q, auto... Args>
struct quantity_spec<Q, Args...> : std::remove_const_t<decltype(Q)> {









};
```

How does it work without CRTP?

```
static_assert(is_of_type<42 * m, quantity<si::metre, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

```
inline constexpr struct height : quantity_spec<length> {} height;
```

```
template<NamedQuantitySpec auto Q, auto... Args>
struct quantity_spec<Q, Args...> : std::remove_const_t<decltype(Q)> {
  template<typename Self, AssociatedUnit U>
  [[nodiscard]] consteval Reference auto operator[](this Self self, U u)
    requires (implicitly_convertible(self, get_associated_quantity(u)))
  {
    return reference<self, u>{};
  }



};
```

How does it work without CRTP?

```
static_assert(is_of_type<42 * m, quantity<si::metre{}, int>>);
static_assert(is_of_type<42 * isq::height[m], quantity<reference<isq::height, si::metre>{}, int>>);
static_assert(is_of_type<isq::height(42 * m), quantity<reference<isq::height, si::metre>{}, int>>);
```

```
static_assert(is_same_v<quantity<isq::height[m]>, quantity<reference<isq::height, si::metre>{}>>);
```

```
inline constexpr struct height : quantity_spec<length> {} height;
```

```
template<NamedQuantitySpec auto Q, auto... Args>
struct quantity_spec<Q, Args...> : std::remove_const_t<decltype(Q)> {
  template<typename Self, AssociatedUnit U>
  [[nodiscard]] consteval Reference auto operator[](this Self self, U u)
    requires (implicitly_convertible(self, get_associated_quantity(u)))
  {
    return reference<self, u>{};
  }

  template<typename Self, typename Q>
  [[nodiscard]] constexpr Quantity auto operator()(this Self self, Q&& q) const
    requires Quantity<std::remove_cvref_t<Q>> &&
             (explicitly_convertible(std::remove_cvref_t<Q>::quantity_spec, self))
  {
    return std::forward<Q>(q).number() * reference<self, std::remove_cvref_t<Q>::unit>{};
  }
};
```

Expression Templates improve readability of generated types

```
constexpr auto q1 = 220 * km / (2 * h);
static_assert(is_of_type<q1, quantity<derived_unit<si::kilo_<si::metre>, per<si::hour>>(), int>>);
```

Expression Templates improve readability of generated types

```
constexpr auto q1 = 220 * km / (2 * h);
static_assert(is_of_type<q1, quantity<derived_unit<si::kilo_<si::metre>, per<si::hour>>(), int>>);
```

```
constexpr auto q2 = isq::distance(220 * si::kilo<si::metre>) / isq::time(2 * si::hour);
static_assert(is_of_type<q2, quantity<reference<derived_quantity_spec<isq::distance, per<isq::time>>,
                                                derived_unit<si::kilo_<si::metre>, per<si::hour>>>(), int>>);
```

Expression Templates improve readability of generated types

```
constexpr auto q1 = 220 * km / (2 * h);
static_assert(is_of_type<q1, quantity<derived_unit<si::kilo_<si::metre>, per<si::hour>>(), int>>);
```

```
constexpr auto q2 = isq::distance(220 * si::kilo<si::metre>) / isq::time(2 * si::hour);
static_assert(is_of_type<q2, quantity<reference<derived_quantity_spec<isq::distance, per<isq::time>>,
                                                derived_unit<si::kilo_<si::metre>, per<si::hour>>>(), int>>);
```

Thanks to expression templates usage the type resulting from such an expression is easy to understand.

Yet another inconsistency in the language

```
constexpr auto q1 = 220 * km / (2 * h);
static_assert(is_of_type<q1, quantity<derived_unit<si::kilo_<si::metre>, per<si::hour>>(), int>>);
```

```
constexpr auto q2 = isq::distance(220 * si::kilo<si::metre>) / isq::time(2 * si::hour);
static_assert(is_of_type<q2, quantity<reference<derived_quantity_spec<isq::distance, per<isq::time>>,
                                                derived_unit<si::kilo_<si::metre>, per<si::hour>>>(), int>>);
```

- C++ standard allows providing the same identifier for a class type and its instance which is used heavily in the library
- However
  - it is not allowed to do it for class templates and its instances
  - variable template can't share the same identifier as a class template

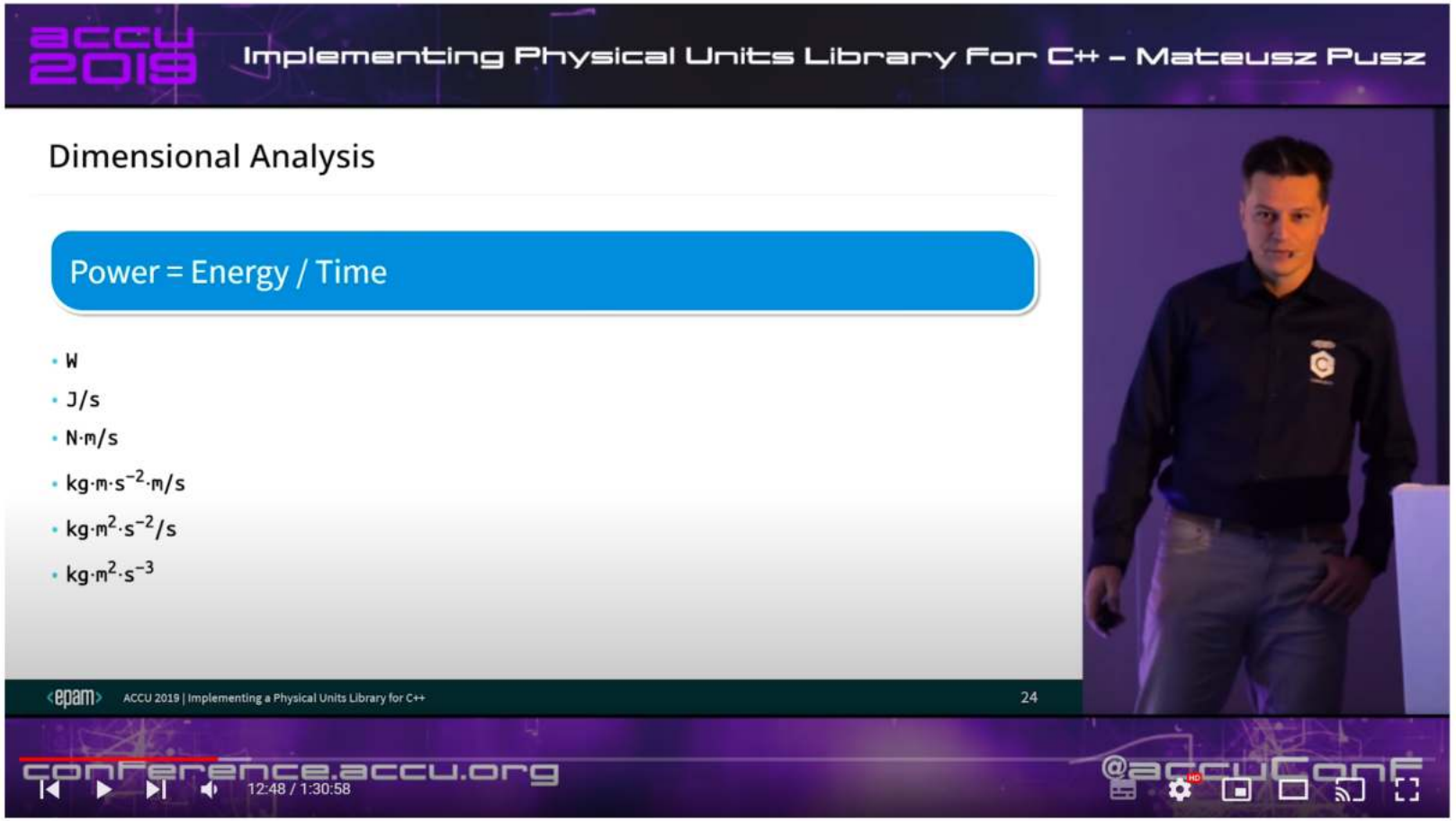
Yet another inconsistency in the language

```
constexpr auto q1 = 220 * km / (2 * h);
static_assert(is_of_type<q1, quantity<derived_unit<si::kilo_<si::metre>, per<si::hour>>(), int>>);
```

```
constexpr auto q2 = isq::distance(220 * si::kilo<si::metre>) / isq::time(2 * si::hour);
static_assert(is_of_type<q2, quantity<reference<derived_quantity_spec<isq::distance, per<isq::time>>,
                                                derived_unit<si::kilo_<si::metre>, per<si::hour>>>(), int>>);
```

- C++ standard allows providing the same identifier for a class type and its instance which is used heavily in the library
- However
  - it is not allowed to do it for class templates and its instances
  - variable template can't share the same identifier as a class template
- This limitation is also problematic for P2601 "Make redundant empty angle brackets optional"

ACCU 2019



Implementing Physical Units Library for C++ - Mateusz Pusz [ACCU 2019]

Dimensional Analysis

```
static_assert(isq::power.dimension == isq::energy.dimension / isq::time.dimension);
```

Dimensional Analysis

```
static_assert(isq::power.dimension == isq::energy.dimension / isq::time.dimension);
```

```
quantity<isq::power[W]> q1 = isq::energy(84 * J) / (2 * s);
auto q2 = q1[J / s];
auto q3 = q1[N * m / s];
auto q4 = q1[kg * m2 / s3];
```

Dimensional Analysis

```
static_assert(isq::power.dimension == isq::energy.dimension / isq::time.dimension);
```

```
quantity<isq::power[W]> q1 = isq::energy(84 * J) / (2 * s);
auto q2 = q1[J / s];
auto q3 = q1[N * m / s];
auto q4 = q1[kg * m2 / s3];
```

```
std::println("{}", q1);
std::println("{}", q2);
std::println("{}", q3);
std::println("{}", q4);
```

```
42 W
42 J/s
42 N m/s
42 kg m²/s³
```

Dimensional Analysis

```
static_assert(isq::power.dimension == isq::energy.dimension / isq::time.dimension);
```

```
quantity<isq::power[W]> q1 = isq::energy(84 * J) / (2 * s);
auto q2 = q1[J / s];
auto q3 = q1[N * m / s];
auto q4 = q1[kg * m2 / s3];
```

```
std::println("{}", q1);
std::println("{}", q2);
std::println("{}", q3);
std::println("{}", q4);
```

```
42 W
42 J/s
42 N m/s
42 kg m²/s³
```

```
std::println("{:%dq}", q1);
std::println("{:%dq}", q2);
std::println("{:%dq}", q3);
std::println("{:%dnq}", q4);
```

```
W
J/s
N·m/s
kg·m²·s⁻³
```

TIP: Parameterized behavior

```cpp
struct unit_symbol_formatting {
  text_encoding encoding = text_encoding::default_encoding;
  unit_symbol_solidus solidus = unit_symbol_solidus::default_denominator;
  unit_symbol_separator separator = unit_symbol_separator::default_separator;
};

template<typename CharT = char, Unit U>
[[nodiscard]] constexpr std::basic_string<CharT> unit_symbol(U u, unit_symbol_formatting fmt = unit_symbol_formatting{})
```

TIP: Parameterized behavior

```cpp
struct unit_symbol_formatting {
  text_encoding encoding = text_encoding::default_encoding;
  unit_symbol_solidus solidus = unit_symbol_solidus::default_denominator;
  unit_symbol_separator separator = unit_symbol_separator::default_separator;
};

template<typename CharT = char, Unit U>
[[nodiscard]] constexpr std::basic_string<CharT> unit_symbol(U u, unit_symbol_formatting fmt = unit_symbol_formatting{})
```

```cpp
static_assert(unit_symbol(kilogram * metre / square(second)) == "kg m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.separator = half_high_dot}) == "kg·m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii}) == "kg m/s^2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = always}) == "kg m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii, .solidus = always}) == "kg m/s^2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = never}) == "kg m s⁻²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii, .solidus = never}) == "kg m s^-2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = never, .separator = half_high_dot}) == "kg·m·s⁻²");
```

TIP: Parametrized behavior

```
struct unit_symbol_formatting {
  text_encoding encoding = text_encoding::default_encoding;
  unit_symbol_solidus solidus = unit_symbol_solidus::default_denominator;
  unit_symbol_separator separator = unit_symbol_separator::default_separator;
};

template<typename CharT = char, Unit U>
[[nodiscard]] constexpr std::basic_string<CharT> unit_symbol(U u, unit_symbol_formatting fmt = unit_symbol_formatting{})
```

```
static_assert(unit_symbol(kilogram * metre / square(second)) == "kg m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.separator = half_high_dot}) == "kg·m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii}) == "kg m/s^2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = always}) == "kg m/s²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii, .solidus = always}) == "kg m/s^2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = never}) == "kg m s⁻²");
static_assert(unit_symbol(kilogram * metre / square(second), {.encoding = ascii, .solidus = never}) == "kg m s^-2");
static_assert(unit_symbol(kilogram * metre / square(second), {.solidus = never, .separator = half_high_dot}) == "kg·m·s⁻²");
```

Starting from C++20 **`std::string`** and **`std::vector`** are **`constexpr`**-friendly.

Quantity arithmetics

Mutually comparable quantities are called **quantities of the same kind**. Two or more quantities cannot be **added or subtracted** unless they belong to the same category of mutually comparable quantities.

-- ISO 80000

Quantity arithmetics

COMPARISONS

```
static_assert(isq::width(1 * m) == isq::height(1 * m));
```

Quantity arithmetics

COMPARISONS

```
static_assert(isq::width(1 * m) == isq::height(1 * m));
```

ADDITION AND SUBTRACTION

```
static_assert(isq::width(1 * m) + isq::height(1 * m) == 2 * m);
static_assert((isq::width(1 * m) + isq::height(1 * m)).quantity_spec == isq::length);
```

Quantity value conversions

VALUE-PRESERVING

```
auto q1 = 5 * km;
std::println("{}", q1[m]);
```

Quantity value conversions

VALUE-PRESERVING

```
auto q1 = 5 * km;
std::println("{}", q1[m]);
```

```
auto q1 = 5 * km;
quantity<si::metre, int> q2 = q1;
```

Quantity value conversions

VALUE-PRESERVING

```
auto q1 = 5 * km;
std::println("{}", q1[m]);
```

```
auto q1 = 5 * km;
quantity<si::metre, int> q2 = q1;
```

TRUNCATING CONVERSIONS

```
auto q1 = 2.5 * m;
auto q2 = value_cast<int>(q1);
```

Quantity value conversions

VALUE-PRESERVING

```
auto q1 = 5 * km;
std::println("{}", q1[m]);
```

```
auto q1 = 5 * km;
quantity<si::metre, int> q2 = q1;
```

TRUNCATING CONVERSIONS

```
auto q1 = 2.5 * m;
auto q2 = value_cast<int>(q1);
```

```
auto q1 = 1'250 * m;
auto q2 = value_cast<km>(q1);
```

Quantity type conversions

IMPLICIT

```
void foo(quantity<isq::length[m]>);
```

```
quantity<isq::height[m]> q = 42 * m;
foo(q);
```

Quantity type conversions

IMPLICIT

```
void foo(quantity<isq::length[m]>);
```

```
quantity<isq::height[m]> q = 42 * m;
foo(q);
```

EXPLICIT

```
void foo(quantity<isq::height[m]>);
```

```
quantity<isq::length[m]> q = 42 * m;
foo(isq::height(q));
```

Quantity type conversions

IMPLICIT

```
void foo(quantity<isq::length[m]>);
```

```
quantity<isq::height[m]> q = 42 * m;
foo(q);
```

EXPLICIT

```
void foo(quantity<isq::height[m]>);
```

```
quantity<isq::length[m]> q = 42 * m;
foo(isq::height(q));
```

CAST

```
void foo(quantity<isq::height[m]>);
```

```
quantity<isq::width[m]> q = 42 * m;
foo(quantity_cast<isq::height>(q));
```

Initializing compound types

```
using state = kalman::state<quantity<isq::position_vector[m]>,
                            quantity<isq::velocity[m / s]>,
                            quantity<isq::acceleration[m / s2]>>;

const state initial = {30 * km, 50 * (m / s), 0 * (m / s2)};
const quantity<isq::position_vector[m], int> measurements[] = {
    30'160 * m, 30'365 * m, 30'890 * m, 31'050 * m, 31'785 * m,
    32'215 * m, 33'130 * m, 34'510 * m, 36'010 * m, 37'265 * m
};
```

Initializing compound types

```
using state = kalman::state<quantity<isq::position_vector[m]>,
                            quantity<isq::velocity[m / s]>,
                            quantity<isq::acceleration[m / s2]>>;

const state initial = {30 * km, 50 * (m / s), 0 * (m / s2)};
const quantity<isq::position_vector[m], int> measurements[] = {
    30'160 * m, 30'365 * m, 30'890 * m, 31'050 * m, 31'785 * m,
    32'215 * m, 33'130 * m, 34'510 * m, 36'010 * m, 37'265 * m
};
```

Thanks to the implicit conversion from the quantity kind we do not have to repeat **`isq::position_vector`** in the initialization of every array element while still being unit-safe.

Explicit quantity type casts with `quantity_cast`

```
template<typename T>
distance spherical_distance(position<T> from, position<T> to)
{
  constexpr auto earth_radius = 6'371 * isq::radius[si::kilo<si::metre>];

  // the haversine formula
  using isq::sin, isq::cos, isq::asin;
  const auto sin_lat = sin((to.lat - from.lat) / 2);
  const auto sin_lon = sin((to.lon - from.lon) / 2);
  const auto central_angle = 2 * asin(sqrt(sin_lat * sin_lat +
                             cos(from.lat) * cos(to.lat) * sin_lon * sin_lon));

  return quantity_cast<isq::distance>(earth_radius * central_angle);
}
```

Explicit quantity type casts with `quantity_cast`

```
template<typename T>
distance spherical_distance(position<T> from, position<T> to)
{
  constexpr auto earth_radius = 6'371 * isq::radius[si::kilo<si::metre>];

  // the haversine formula
  using isq::sin, isq::cos, isq::asin;
  const auto sin_lat = sin((to.lat - from.lat) / 2);
  const auto sin_lon = sin((to.lon - from.lon) / 2);
  const auto central_angle = 2 * asin(sqrt(sin_lat * sin_lat +
                             cos(from.lat) * cos(to.lat) * sin_lon * sin_lon));

  return quantity_cast<isq::distance>(earth_radius * central_angle);
}
```

```
template<typename T = double>
using latitude = quantity<isq::angular_measure[si::degree], ranged_representation<T, -90, 90>>;
template<typename T = double>
using longitude = quantity<isq::angular_measure[si::degree], ranged_representation<T, -180, 180>>;
```

[basic.lookup.argdep]

- For each argument type **T** in the function call, there is a **set of zero or more associated namespaces** and a set of zero or more associated entities (other than namespaces) **to be considered**

[basic.lookup.argdep]

- For each argument type **T** in the function call, there is a **set of zero or more associated namespaces** and a set of zero or more associated entities (other than namespaces) **to be considered**
- The sets of namespaces and entities are **determined in the following way**
  - if **T** is a **class template specialization**, its associated namespaces and entities also include
    - the *namespaces* and entities *associated with the types of the template arguments* provided for **template type parameters** (excluding template template parameters)
    - the templates used as template template arguments
    - the *namespaces of which any template template arguments are members*
    - the classes of which any member templates used as template template arguments are members

[basic.lookup.argdep]

- For each argument type **T** in the function call, there is a **set of zero or more associated namespaces** and a set of zero or more associated entities (other than namespaces) **to be considered**
- The sets of namespaces and entities are **determined in the following way**
  - if **T** is a **class template specialization**, its associated namespaces and entities also include
    - the *namespaces* and entities *associated with the types of the template arguments* provided for **template type parameters** (excluding template template parameters)
    - the templates used as template template arguments
    - the *namespaces of which any template template arguments are members*
    - the classes of which any member templates used as template template arguments are members

[Note 1 : Non-type template arguments do not contribute to the set of associated namespaces. —end note]

Strong angular system

Angles in the SI: a detailed proposal for solving the problem

Paul Quincey
National Physical Laboratory[1], Hampton Road, Teddington, TW11 0LW, United Kingdom.
paulgquincey@gmail.com

Published in Metrologia 2021

Abstract

A recent Letter [1] proposed changing the dimensionless status of the radian and steradian within the SI, while allowing the continued use of the convention to set the angle 1 radian equal to the number 1 within equations, providing this is done explicitly. This would bring the advantages of a physics-based, consistent, and logically-robust unit system, with unambiguous units for all physical quantities, for the first time, while any upheaval to familiar equations and routine practice would be minimised. More details of this proposal are given here. The only notable changes for typical end-users would be: improved units for the quantities torque, angular momentum and moment of inertia; a statement of the convention accompanying some familiar equations; and the use of different symbols for $\hbar$ the action and $\hbar$ the angular momentum, a small step forward for quantum physics. Some features of the proposal are already established practice for quantities involving the steradian such as radiant intensity and radiance.

Strong angular system

```
namespace mp_units::angular {

inline constexpr struct dim_angle : base_dimension<"A"> {} dim_angle;

inline constexpr struct angle : quantity_spec<dim_angle> {} angle;
inline constexpr struct solid_angle : quantity_spec<pow<2>(angle)> {} solid_angle;

inline constexpr struct radian : named_unit<"rad", kind_of<angle>> {} radian;
inline constexpr struct revolution : named_unit<"rev", mag<2> * mag_pi * radian> {} revolution;
inline constexpr struct degree : named_unit<basic_symbol_text{"°", "deg"}, mag<ratio{1, 360}> * revolution> {} degree;
inline constexpr struct gradian : named_unit<basic_symbol_text{"ᵍ", "grad"}, mag<ratio{1, 400}> * revolution> {} gradian;
inline constexpr struct steradian : named_unit<"sr", square(radian)> {} steradian;

}
```

Strong angular system

```
namespace mp_units::angular {

inline constexpr struct dim_angle : base_dimension<"A"> {} dim_angle;

inline constexpr struct angle : quantity_spec<dim_angle> {} angle;
inline constexpr struct solid_angle : quantity_spec<pow<2>(angle)> {} solid_angle;

inline constexpr struct radian : named_unit<"rad", kind_of<angle>> {} radian;
inline constexpr struct revolution : named_unit<"rev", mag<2> * mag_pi * radian> {} revolution;
inline constexpr struct degree : named_unit<basic_symbol_text{"°", "deg"}, mag<ratio{1, 360}> * revolution> {} degree;
inline constexpr struct gradian : named_unit<basic_symbol_text{"ᵍ", "grad"}, mag<ratio{1, 400}> * revolution> {} gradian;
inline constexpr struct steradian : named_unit<"sr", square(radian)> {} steradian;

}
```

```
namespace mp_units::isq_angle {

using namespace isq;

inline constexpr struct cotes_angle_constant : quantity_spec<angular::angle> {} cotes_angle_constant;
inline constexpr struct angular_measure :
          quantity_spec<angular::angle, cotes_angle_constant * arc_length / radius> {} angular_measure;
inline constexpr struct solid_angular_measure :
          quantity_spec<pow<2>(cotes_angle_constant) * area / pow<2>(radius)> {} solid_angular_measure;
// ...
}
```

Two sets of overloads

```
namespace isq {

template<ReferenceOf<angular_measure> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<one, Rep> sin(const quantity<R, Rep>& q) noexcept;

template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<si::radian, Rep> asin(const quantity<R, Rep>& q) noexcept;

}
```

```
namespace angular {

template<ReferenceOf<angle> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<one, Rep> sin(const quantity<R, Rep>& q) noexcept;

template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<radian, Rep> asin(const quantity<R, Rep>& q) noexcept;

}
```

Two sets of overloads

```
namespace isq {

template<ReferenceOf<angular_measure> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<one, Rep> sin(const quantity<R, Rep>& q) noexcept;

template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<si::radian, Rep> asin(const quantity<R, Rep>& q) noexcept;

}
```

```
namespace angular {

template<ReferenceOf<angle> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<one, Rep> sin(const quantity<R, Rep>& q) noexcept;

template<ReferenceOf<dimensionless> auto R, typename Rep>
  requires treat_as_floating_point<Rep>
[[nodiscard]] inline quantity<radian, Rep> asin(const quantity<R, Rep>& q) noexcept;

}
```

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

```
quantity<si::metre / si::second> q;
foo(q);
```

- Nice syntax and usability
- ADL doesn't use namespaces of units for lookup

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

```
quantity<si::metre / si::second> q;
foo(q);
```

- Nice syntax and usability
- ADL doesn't use namespaces of units for lookup

```
quantity<decltype(si::metre / si::second)> q;
foo(q);
```

- Really inconvenient
- ADL works as expected

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

```
quantity<si::metre / si::second> q;
foo(q);
```

- Nice syntax and usability
- ADL doesn't use namespaces of units for lookup

```
quantity<decltype(si::metre / si::second)> q;
foo(q);
```

- Really inconvenient
- ADL works as expected

Why so often we can't have nice things?

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

A BREAKING CHANGE

```
namespace my {

static constexpr struct X {} x;
enum class E { e1, e2 };

}
```

```
template<auto>
class MyClass {};

foo(MyClass<my::E::e1>{});
foo(MyClass<&my::x>{});
```

ADL rules are incomplete

ADL rules are incomplete! It was quite fine before C++20, but now we have new exciting use cases for NTTPs.

A BREAKING CHANGE

```
namespace my {

static constexpr struct X {} x;
enum class E { e1, e2 };

}
```

```
template<auto>
class MyClass {};

foo(MyClass<my::E::e1>{});
foo(MyClass<&my::x>{});
```

I whish we had epochs in the language...

Strong interfaces

```
class Box {
  quantity<horizontal_area[m2]> base_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    base_(l * w), height_(h)
  {}
  // ...
};
```

Strong interfaces

```
class Box {
  quantity<horizontal_area[m2]> base_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    base_(l * w), height_(h)
  {}
  // ...
};
```

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

Strong interfaces

```
class Box {
  quantity<horizontal_area[m2]> base_;
  quantity<isq::height[m]> height_;
public:
  Box(quantity<horizontal_length[m]> l, quantity<isq::width[m]> w, quantity<isq::height[m]> h):
    base_(l * w), height_(h)
  {}
  // ...
};
```

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

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

Generic interfaces

```
QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto l,
                                      QuantityOf<isq::time> auto d)
{
  return isq::speed(l / d);
}
```

Generic interfaces

```
QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto l,
                                      QuantityOf<isq::time> auto d)
{
  return isq::speed(l / d);
}
```

```
auto s1 = avg_speed(220 * km, 2 * h);
auto s2 = avg_speed(140 * mi, 2 * h);
auto s3 = avg_speed(isq::altitude(2. * km), isq::duration(10 * min));
```

Generic interfaces

```
QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto l,
                                      QuantityOf<isq::time> auto d)
{
  return isq::speed(l / d);
}
```

```
auto s1 = avg_speed(220 * km, 2 * h);
auto s2 = avg_speed(140 * mi, 2 * h);
auto s3 = avg_speed(isq::altitude(2. * km), isq::duration(10 * min));
```

```
std::println("{}", s1);
std::println("{}", s2);
std::println("{}", s3);
```

Generic interfaces

```
QuantityOf<isq::speed> auto avg_speed(QuantityOf<isq::length> auto l,
                                      QuantityOf<isq::time> auto d)
{
  return isq::speed(l / d);
}
```

```
auto s1 = avg_speed(220 * km, 2 * h);
auto s2 = avg_speed(140 * mi, 2 * h);
auto s3 = avg_speed(isq::altitude(2. * km), isq::duration(10 * min));
```

```
std::println("{}", s1);
std::println("{}", s2);
std::println("{}", s3);
```

```
110 km/h
70 mi/h
0.2 km/min
```

Custom representation types

```cpp
template<typename T>
class measurement {
public:
  using value_type = T;

  measurement() = default;

  constexpr explicit measurement(value_type val, const value_type& err = {});

  [[nodiscard]] constexpr const value_type& value() const { return value_; }
  [[nodiscard]] constexpr const value_type& uncertainty() const { return uncertainty_; }

  [[nodiscard]] constexpr value_type relative_uncertainty() const { return uncertainty() / value(); }
  [[nodiscard]] constexpr value_type lower_bound() const { return value() - uncertainty(); }
  [[nodiscard]] constexpr value_type upper_bound() const { return value() + uncertainty(); }

  // ...

private:
  value_type value_{};
  value_type uncertainty_{};
};
```

Custom representation types

```
const auto length = measurement{123., 1.} * m;
std::cout << "10 * " << length << " = " << 10 * length << '\n';
```

```
10 * 123 ± 1 m = 1230 ± 10 m
```

Custom representation types

```
const auto length = measurement{123., 1.} * m;
std::cout << "10 * " << length << " = " << 10 * length << '\n';
```

```
10 * 123 ± 1 m = 1230 ± 10 m
```

```
const auto a = isq::acceleration(measurement{9.8, 0.1} * (m / s2));
const auto t = measurement{1.2, 0.1} * s;

const QuantityOf<isq::velocity> auto v = a * t;
std::cout << a << " * " << t << " = " << v << " = " << v[km / h] << '\n';
```

```
9.8 ± 0.1 m/s² * 1.2 ± 0.1 s = 11.76 ± 0.98732 m/s = 42.336 ± 3.55435 km/h
```

What should the result be?

```
auto res = la_vector<int, 3>{1, 2, 3} * m;
```

What should the result be?

```
auto res = la_vector<int, 3>{1, 2, 3} * m;
```

LA LIBRARY

- Checks if **`int`** can be multiplied with **`m`**
- Tries to produce

  `la_vector<quantity<si::metre, int>, 3>`

What should the result be?

```
auto res = la_vector<int, 3>{1, 2, 3} * m;
```

LA LIBRARY

- Checks if **`int`** can be multiplied with **`m`**
- Tries to produce

  `la_vector<quantity<si::metre, int>, 3>`

MP-UNITS

- Treats **`la_vector<int, 3>`** as **`Rep`**
- Tries to produce

  `quantity<si::metre, la_vector<int, 3>>`

What should the result be?

```
auto res = la_vector<int, 3>{1, 2, 3} * m;
```

- **Factory function** to use in case of such a collision

```
auto res = make_quantity<si::metre>(la_vector<int, 3>{1, 2, 3});
```

What should the result be?

```
auto res = la_vector<int, 3>{1, 2, 3} * m;
```

- **Factory function** to use in case of such a collision

```
auto res = make_quantity<si::metre>(la_vector<int, 3>{1, 2, 3});
```

- When reasonable, a **dedicated overload** of an **`operator*`** can be provided

```
template<typename Rep, size_t N, Reference R>
[[nodiscard]] quantity<R{}, la_vector<Rep, N>> operator*(const la_vector<Rep, N>& lhs, R)
{
  return make_quantity<R{}>(lhs);
}
```

Custom representation types

```
template<typename T>
class measurement {
public:
  using value_type = T;

  measurement() = default;

  constexpr explicit measurement(value_type val, const value_type& err = {}) : value_(std::move(val))
  {
    using namespace std;
    uncertainty_ = abs(err);
  }

  // ...

private:
  value_type value_{};
  value_type uncertainty_{};
};
```

Using declaration can't be provided in a constructor initializer list.

Proper math for wrapping types

```
template<Quantity Q>
[[nodiscard]] inline Q abs(const Q& q) noexcept
  requires requires { abs(q.number()); } || requires { std::abs(q.number()); }
{
  using std::abs;
  return make_quantity<Q::reference>(abs(q.number()));
}
```

Should we have CPOs for mathematical functions?

Many people work now on libraries providing custom "smart" numeric representation types/wrappers.

CPOs for math functions could increase interop between various numeric-related types.

The affine space

```
inline constexpr struct mean_sea_level : absolute_point_origin<isq::height> {} mean_sea_level;
```

The affine space

```
inline constexpr struct mean_sea_level : absolute_point_origin<isq::height> {} mean_sea_level;
```

```
constexpr quantity_point<isq::height[m], mean_sea_level> ZRH_ground_level = 432 * m;
constexpr quantity_point<isq::height[m], mean_sea_level> GDN_ground_level = 149 * m;
```

The affine space

```
inline constexpr struct mean_sea_level : absolute_point_origin<isq::height> {} mean_sea_level;
```

```
constexpr quantity_point<isq::height[m], mean_sea_level> ZRH_ground_level = 432 * m;
constexpr quantity_point<isq::height[m], mean_sea_level> GDN_ground_level = 149 * m;
```

```
constexpr quantity_point<isq::height[m], ZRH_ground_level> ZRH_afe = 200 * m;
constexpr quantity_point<isq::height[m], GDN_ground_level> GDN_afe = 450 * m;
```

The affine space

```
inline constexpr struct mean_sea_level : absolute_point_origin<isq::height> {} mean_sea_level;
```

```
constexpr quantity_point<isq::height[m], mean_sea_level> ZRH_ground_level = 432 * m;
constexpr quantity_point<isq::height[m], mean_sea_level> GDN_ground_level = 149 * m;
```

```
constexpr quantity_point<isq::height[m], ZRH_ground_level> ZRH_afe = 200 * m;
constexpr quantity_point<isq::height[m], GDN_ground_level> GDN_afe = 450 * m;
```

NTTPs are awesome! :-D

The affine space

```
inline constexpr struct mean_sea_level : absolute_point_origin<isq::height> {} mean_sea_level;
```

```
constexpr quantity_point<isq::height[m], mean_sea_level> ZRH_ground_level = 432 * m;
constexpr quantity_point<isq::height[m], mean_sea_level> GDN_ground_level = 149 * m;
```

```
constexpr quantity_point<isq::height[m], ZRH_ground_level> ZRH_afe = 200 * m;
constexpr quantity_point<isq::height[m], GDN_ground_level> GDN_afe = 450 * m;
```

```
static_assert(ZRH_afe.relative() == 200 * m);
static_assert(GDN_afe.relative() == 450 * m);

static_assert(ZRH_afe.absolute() == 632 * m);
static_assert(GDN_afe.absolute() == 599 * m);

static_assert(GDN_afe - ZRH_afe == -33 * m);
static_assert(GDN_ground_level - ZRH_ground_level == -283 * m);
static_assert(ZRH_afe - GDN_ground_level == 483 * m);
```

Faster than lightspeed constants

```
inline constexpr struct standard_gravity :
  named_unit<basic_symbol_text{"g₀", "g_0"}, mag<ratio{980'665, 100'000}> * metre / square(second)> {} standard_gravity;
```

```
inline constexpr auto g = 1 * si::standard_gravity;
```

Faster than lightspeed constants

```
inline constexpr struct standard_gravity :
  named_unit<basic_symbol_text{"g₀", "g_0"}, mag<ratio{980'665, 100'000}> * metre / square(second)> {} standard_gravity;
```

```
inline constexpr auto g = 1 * si::standard_gravity;
```

```
class StorageTank {
public:
  [[nodiscard]] constexpr QuantityOf<isq::weight> auto filled_weight() const {
    const QuantityOf<isq::mass> auto mass = density_ * isq::volume(base_ * height_);
    return isq::weight(mass * g);
  }
  [[nodiscard]] constexpr quantity<isq::height[m]> fill_level(const quantity<isq::mass[kg]>& measured_mass) const {
    return height_ * measured_mass * g / filled_weight();
  }
  // ...
};
```

Faster than lightspeed constants

```
inline constexpr struct standard_gravity :
  named_unit<basic_symbol_text{"g₀", "g_0"}, mag<ratio{980'665, 100'000}> * metre / square(second)> {} standard_gravity;
```

```
inline constexpr auto g = 1 * si::standard_gravity;
```

```
class StorageTank {
public:
  [[nodiscard]] constexpr QuantityOf<isq::weight> auto filled_weight() const {
    const QuantityOf<isq::mass> auto mass = density_ * isq::volume(base_ * height_);
    return isq::weight(mass * g);
  }
  [[nodiscard]] constexpr quantity<isq::height[m]> fill_level(const quantity<isq::mass[kg]>& measured_mass) const {
    return height_ * measured_mass * g / filled_weight();
  }
  // ...
};
```

```
const auto filled_weight = tank.filled_weight();
const auto fill_level = tank.fill_level(measured_mass);
std::println("fill weight at {} = {} ({})", fill_time, filled_weight, filled_weight[N]);
std::println("fill height at {} = {} ({} full)", fill_time, fill_level, fill_percent);
```

Faster than lightspeed constants

```
inline constexpr struct standard_gravity :
  named_unit<basic_symbol_text{"g₀", "g_0"}, mag<ratio{980'665, 100'000}> * metre / square(second)> {} standard_gravity;
```

```
inline constexpr auto g = 1 * si::standard_gravity;
```

```
class StorageTank {
public:
  [[nodiscard]] constexpr QuantityOf<isq::weight> auto filled_weight() const {
    const QuantityOf<isq::mass> auto mass = density_ * isq::volume(base_ * height_);
    return isq::weight(mass * g);
  }
  [[nodiscard]] constexpr quantity<isq::height[m]> fill_level(const quantity<isq::mass[kg]>& measured_mass) const {
    return height_ * measured_mass * g / filled_weight();
  }
  // ...
};
```

```
const auto filled_weight = tank.filled_weight();
const auto fill_level = tank.fill_level(measured_mass);
std::println("fill weight at {} = {} ({})", fill_time, filled_weight, filled_weight[N]);
std::println("fill height at {} = {} ({} full)", fill_time, fill_level, fill_percent);
```

```
fill weight at 200 s = 100 g₀ kg (980.665 N)
fill height at 200 s = 0.04 m (20 % full)
```

SUMMARY

Simpler

- No more downcasting facility and CRTP idiom
- User-facing abstractions no longer needed
  - **`exponent`**
  - **`prefix`**
  - **`quantity_kind`** and **`quantity_point_kind`**
- One **`named_unit`** and **`quantity_spec`** to rule them all
- Much terser systems definitions
- ISQ dimensions are *not templates* anymore

Better

- **Composable units**
- Consistent *NTTP usage* improves readability and usability
- *Expression templates* improve readability of types
- *Pure dimensionless analysis*
- Robust *quantity creation* helpers
- ***`quantity_spec`*** to improve quantity definitions
- Support for *all ISQ quantities*
- Proper *quantity kinds* support
- Faster than lightspeed *constants*
- Much more ...

C++ IS AWESOME!!!

Every new C++ language version adds new exciting features that enable more user-friendly abstractions.

- Concepts and constraints
- NTTPs
- deducing **`this`**

C++ IS AWESOME!!!

Every new C++ language version adds new exciting features that enable more user-friendly abstractions.

- Concepts and constraints
- NTTPs
- deducing **`this`**

In C++ we can do amazing things at compile-time without a need to pay for them at runtime.

There is still place for improvement / My Wish List

- "Artificial" limitations of structural types force bad class design
  - we should not publicly expose data members of our classes

There is still place for improvement / My Wish List

- "Artificial" limitations of structural types force bad class design
  - we should not publicly expose data members of our classes
- ADL rules were not updated when we weakened the restrictions for NTTPs
  - hard to provide/use customization points

There is still place for improvement / My Wish List

- "Artificial" limitations of structural types force bad class design
  - we should not publicly expose data members of our classes
- ADL rules were not updated when we weakened the restrictions for NTTPs
  - hard to provide/use customization points
- Math/numeric functions are hard to use for non-fundamental types
  - we will see lots of "smart" numeric types in the near future

There is still place for improvement / My Wish List

- "Artificial" limitations of structural types force bad class design
  - we should not publicly expose data members of our classes
- ADL rules were not updated when we weakened the restrictions for NTTPs
  - hard to provide/use customization points
- Math/numeric functions are hard to use for non-fundamental types
  - we will see lots of "smart" numeric types in the near future
- Consider allowing class template and its object to have the same identifier
  - consistency with non-template classes
  - enables P2601


CAUTION
Programming
is addictive
(and too much fun)