train IT
Quantities and Units Library
P3045
Mateusz Pusz
Croydon, 2026

Plan for today

WE ARE NOT HERE TO

- discuss or challenge the design tonight
- make any decisions

Plan for today

WE ARE NOT HERE TO

- discuss or challenge the design tonight
- make any decisions

WE ARE HERE TO

- walk through what is being proposed
- ask questions to understand the library's frameworks

Plan for today

WE ARE NOT HERE TO

- discuss or challenge the design tonight
- make any decisions

WE ARE HERE TO

- walk through what is being proposed
- ask questions to understand the library's frameworks

The goal is to build **shared awareness** of:

- the **domain** (metrology is surprisingly deep)
- the **industry need** (real bugs, real cost)
- the **challenges** (why this is hard to get right)
- the **solution** (what P3045 actually proposes)

Plan for today

WE ARE NOT HERE TO

- discuss or challenge the design tonight
- make any decisions

WE ARE HERE TO

- walk through what is being proposed
- ask questions to understand the library's frameworks

The goal is to build **shared awareness** of:

- the **domain** (metrology is surprisingly deep)
- the **industry need** (real bugs, real cost)
- the **challenges** (why this is hard to get right)
- the **solution** (what P3045 actually proposes)

Think of it as a guided tour, not a review session. Grab a drink, settle in, and enjoy the ride. 🍷

Before we dive in...

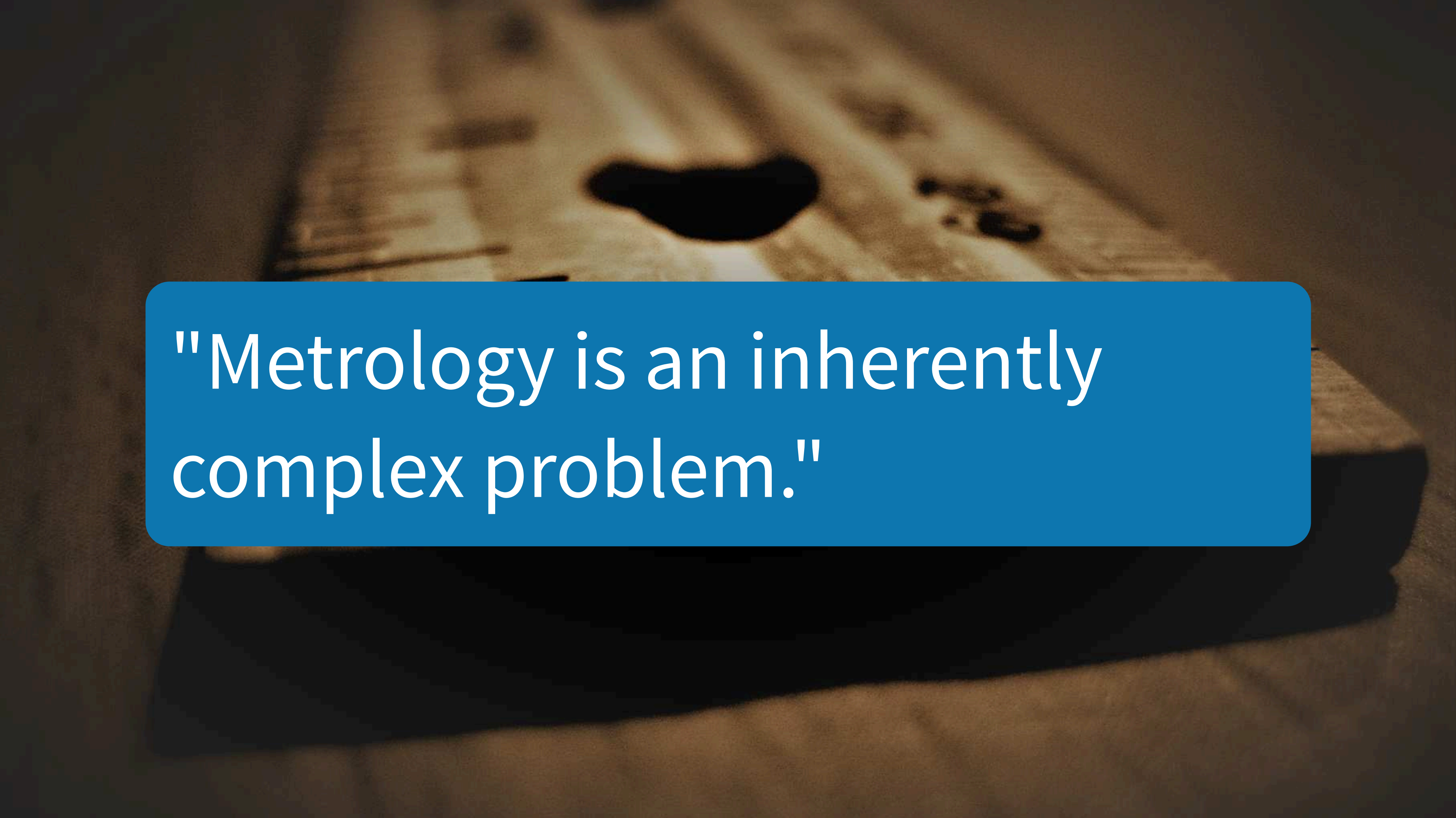
Before we dive in...

Most C++ developers think units are straightforward:
"It's just meters and seconds — how hard can it be?"

Before we dive in...

Most C++ developers think units are straightforward:
"It's just meters and seconds — how hard can it be?"

Let's fix that assumption right now.


"Metrology is an inherently complex problem."

"Metrology is an inherently complex problem."

Say it with me — out loud! 🎤

"Metrology is an inherently complex problem."

One more time — with conviction! 🔊

Metrology is an inherently complex problem

Metrology is an inherently complex problem

This will be a **recurring theme** throughout this talk. Every time you hear it, remember this moment. 🎤

Metrology is an inherently complex problem

This will be a **recurring theme** throughout this talk. Every time you hear it, remember this moment. 🎤

- The domain is defined by **international standards** (ISO 80000)
- It involves **multiple interacting concepts**: dimensions, quantity types, quantity kinds, units, systems of quantities, systems of units, the affine space
- Getting it right is **essential for safety-critical software**
- Getting it wrong is **surprisingly easy**

Agenda

1. The case for standardization

Agenda

1. The case for standardization
2. Real-world pain points

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries
4. Quick domain overview

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries
4. Quick domain overview
5. Six levels of safety

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries
4. Quick domain overview
5. Six levels of safety
6. Scope of the Core Library Framework

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries
4. Quick domain overview
5. Six levels of safety
6. Scope of the Core Library Framework
7. Teachability

Agenda

1. The case for standardization
2. Real-world pain points
3. The evolution of units libraries
4. Quick domain overview
5. Six levels of safety
6. Scope of the Core Library Framework
7. Teachability
8. Summary

THE CASE FOR STANDARDIZATION

Vocabulary types belong in the Standard

Vocabulary types are types that are used **at API boundaries** between independently developed libraries and components.

Vocabulary types belong in the Standard

Vocabulary types are types that are used **at API boundaries** between independently developed libraries and components.

- `std::string`, `std::vector`, `std::optional`, `std::expected`, `std::chrono::duration` — all are vocabulary types

Vocabulary types belong in the Standard

Vocabulary types are types that are used **at API boundaries** between independently developed libraries and components.

- `std::string`, `std::vector`, `std::optional`, `std::expected`, `std::chrono::duration` — all are vocabulary types
- A `quantity` type (like `quantity<m/s>`) is a vocabulary type for any API dealing with physical quantities

Vocabulary types belong in the Standard

Vocabulary types are types that are used **at API boundaries** between independently developed libraries and components.

- `std::string`, `std::vector`, `std::optional`, `std::expected`, `std::chrono::duration` — all are vocabulary types
- A `quantity` type (like `quantity<m/s>`) is a vocabulary type for any API dealing with physical quantities
- Without a standard vocabulary type, every library *invents its own*
  - users end up converting between incompatible representations
  - errors at the boundaries between libraries

Who sees the issue?

```
#include <TGeoVolume.h>  // ROOT geometry
#include <G4Tubs.hh>     // Geant4 geometry

double radius = rootVolume->GetRmax();  // returns 10.0

G4Tubs g4Tube("Tube", 0, radius, halfLength, 0, 2 * M_PI);
```

Different frameworks, different base units

UNIT	GEANT4 VALUE	ROOT VALUE
mm	1.0	0.1
cm	10.0	1.0
m	1000.0	100.0

Different frameworks, different base units

UNIT	GEANT4 VALUE	ROOT VALUE
mm	1.0	0.1
cm	10.0	1.0
m	1000.0	100.0

- ROOT's `GetRmax()` returned `10.0` meaning 10 cm
- Geant4 interprets `10.0` as 10 mm — the tube is 10× too small!

Different frameworks, different base units

UNIT	GEANT4 VALUE	ROOT VALUE
mm	1.0	0.1
cm	10.0	1.0
m	1000.0	100.0

- ROOT's **`GetRmax()`** returned **`10.0`** meaning 10 cm
- Geant4 interprets **`10.0`** as 10 mm — the tube is 10× too small!

No compiler warning, no runtime exception.

Who sees the issue?

"OK, I need to scale from ROOT cm to Geant4 mm. I should multiply..."

Who sees the issue?

"OK, I need to scale from ROOT cm to Geant4 mm. I should multiply..."

```
#include <TGeoVolume.h>        // ROOT geometry
#include <G4Tubs.hh>           // Geant4 geometry
#include <CLHEP/Units/SystemOfUnits.h>

double radius = rootVolume->GetRmax();  // 10.0 (meaning 10 cm)

G4Tubs g4Tube("Tube", 0, radius * CLHEP::mm, halfLength, 0, 2 * M_PI);
```

Who sees the issue?

"OK, I need to scale from ROOT cm to Geant4 mm. I should multiply..."

```
#include <TGeoVolume.h>        // ROOT geometry
#include <G4Tubs.hh>           // Geant4 geometry
#include <CLHEP/Units/SystemOfUnits.h>

double radius = rootVolume->GetRmax();  // 10.0 (meaning 10 cm)

G4Tubs g4Tube("Tube", 0, radius * CLHEP::mm, halfLength, 0, 2 * M_PI);
```

- **`CLHEP::mm`** is **`1.0`** (the value of 1 mm in Geant4's mm-based system)
- **`10.0 * 1.0 = 10.0`** — the tube radius is now **10 mm** ❌

Who sees the issue?

"OK, I need to scale from ROOT cm to Geant4 mm. I should multiply..."

```
#include <TGeoVolume.h>        // ROOT geometry
#include <G4Tubs.hh>           // Geant4 geometry
#include <CLHEP/Units/SystemOfUnits.h>

double radius = rootVolume->GetRmax();  // 10.0 (meaning 10 cm)

G4Tubs g4Tube("Tube", 0, radius * CLHEP::mm, halfLength, 0, 2 * M_PI);
```

Still <u>10× too small!</u> The developer picked the wrong CLHEP constant.

The "Ultimate" Solution

The correct scaling factor for ROOT → Geant4 is **`CLHEP::cm`**:

```
G4Tubs g4Tube("Tube", 0, radius * CLHEP::cm, halfLength, 0, 2 * M_PI);  // 100 mm ✅
```

Metrology is an inherently complex problem

Metrology is an inherently complex problem

What about the other direction? Geant4 $\longrightarrow$ ROOT would need **division** by `CLHEP::cm`.

Metrology is an inherently complex problem

What about the other direction? Geant4 $\longrightarrow$ ROOT would need division by `CLHEP::cm`.

- Which constant?
- Multiply or divide?
- It depends on the **direction** of conversion.

Metrology is an inherently complex problem

What about the other direction? Geant4 $\longrightarrow$ ROOT would need division by `CLHEP::cm`.

- Which constant?
- Multiply or divide?
- It depends on the direction of conversion.

The developer has to think about it every single time. Across 2.5 million lines of C++.

Vocabulary types: Without standardization

```
#include <TGeoVolume.h>   // ROOT geometry
#include <G4Tubs.hh>      // Geant4 geometry
import std;

using namespace std::si::unit_symbols;

std::quantity radius = rootVolume->GetRmax() * cm;  // 10 cm — unambiguous

G4Tubs g4Tube("Tube", 0,
              radius.numerical_value_in(mm),        // 100.0 ✅
              halfLength_mm, 0, 2 * M_PI);
```

Vocabulary types: Without standardization

```
#include <TGeoVolume.h>   // ROOT geometry
#include <G4Tubs.hh>      // Geant4 geometry
import std;

using namespace std::si::unit_symbols;

std::quantity radius = rootVolume->GetRmax() * cm;  // 10 cm — unambiguous

G4Tubs g4Tube("Tube", 0,
              radius.numerical_value_in(mm),        // 100.0 ✅
              halfLength_mm, 0, 2 * M_PI);
```

- **No manual scaling factors** — the type system handles it
- **No multiply-vs-divide confusion** — just state the target unit

Vocabulary types: Without standardization

```
#include <TGeoVolume.h>   // ROOT geometry
#include <G4Tubs.hh>      // Geant4 geometry
import std;

using namespace std::si::unit_symbols;

std::quantity radius = rootVolume->GetRmax() * cm;  // 10 cm — unambiguous

G4Tubs g4Tube("Tube", 0,
              radius.numerical_value_in(mm),        // 100.0 ✅
              halfLength_mm, 0, 2 * M_PI);
```

- **No manual scaling factors** — the type system handles it
- **No multiply-vs-divide confusion** — just state the target unit

Geant4, ROOT, CLHEP each have their own **double** convention.

Vocabulary types: With standardization

```
#include <TGeoVolume.h>   // ROOT geometry
#include <G4Tubs.hh>      // Geant4 geometry
import std;

using namespace std::si::unit_symbols;

std::quantity radius = rootVolume->GetRmax();        // 10 cm

G4Tubs g4Tube("Tube", 0,
              radius,                                // scales automatically ✅
              halfLength_mm, 0, 2 * M_PI);
```

`quantity<si::milli<si::metre>>` becomes the universal vocabulary.

Certification requirements

Mission-critical and life-critical software often requires <u>certification</u>.

Certification requirements

Mission-critical and life-critical software often requires **certification**.

- DO-178C (avionics), ISO 26262 (automotive), IEC 62304 (medical), ECSS (space)

Certification requirements

Mission-critical and life-critical software often requires **certification**.

- DO-178C (avionics), ISO 26262 (automotive), IEC 62304 (medical), ECSS (space)
- These standards require **evidence of correctness** at every level

Certification requirements

Mission-critical and life-critical software often requires **certification**.

- DO-178C (avionics), ISO 26262 (automotive), IEC 62304 (medical), ECSS (space)
- These standards require **evidence of correctness** at every level
- Using well-tested, formally specified **standard library components** simplifies certification

Certification requirements

Mission-critical and life-critical software often requires certification.

- DO-178C (avionics), ISO 26262 (automotive), IEC 62304 (medical), ECSS (space)
- These standards require **evidence of correctness** at every level
- Using well-tested, formally specified **standard library components** simplifies certification
- A standardized quantities and units library would provide:
  - a **formally specified** type system for physical quantities
  - a reference implementation with **extensive test coverage**
  - **broad peer review** from the C++ community

Certification requirements

Third-party libraries require additional certification effort. A standard library component is already part of the certified toolchain.

Certification requirements

Third-party libraries require additional certification effort. A standard library component is already part of the certified toolchain.

- Compile-time safety means **fewer runtime tests needed**
- Type-safe interfaces provide **evidence of correct-by-construction** design
- Reducing human error in unit handling directly addresses **safety objectives**

`std::chrono` proves the case

`std::chrono::duration` is the closest existing analogue to what P3045 proposes — and its standardization was overwhelmingly successful.

`std::chrono` proves the case

`std::chrono::duration` is the closest existing analogue to what P3045 proposes — and its standardization was overwhelmingly successful.

- **Before C++11:** every project had its own time representation
  - milliseconds as **`int`**, microseconds as **`long long`**, seconds as **`double`**
  - endless conversion bugs at API boundaries

`std::chrono` proves the case

> **`std::chrono::duration`** is the closest existing analogue to what P3045 proposes — and its standardization was overwhelmingly successful.

- **Before C++11:** every project had its own time representation
  - milliseconds as **`int`**, microseconds as **`long long`**, seconds as **`double`**
  - endless conversion bugs at API boundaries
- **After C++11:** **`std::chrono::duration`** became the **universal vocabulary**
  - type-safe, zero-overhead, interoperable
  - adopted across the entire ecosystem

`std::chrono` proves the case

`std::chrono::duration` is the closest existing analogue to what P3045 proposes — and its standardization was overwhelmingly successful.

- **Before C++11:** every project had its own time representation
  - milliseconds as **`int`**, microseconds as **`long long`**, seconds as **`double`**
  - endless conversion bugs at API boundaries
- **After C++11:** **`std::chrono::duration`** became the **universal vocabulary**
  - type-safe, zero-overhead, interoperable
  - adopted across the entire ecosystem
- **`std::chrono`** would **not be as successful** if it had remained a third-party library

`std::chrono` proves the case

- `std::chrono` handles **one dimension** (time) with **one system of units**

`std::chrono` proves the case

- **`std::chrono`** handles **one dimension** (time) with **one system of units**
- P3045 generalizes this to **arbitrary dimensions**, **quantity hierarchies**, and **multiple systems of quantities and units**

`std::chrono` proves the case

- `std::chrono` handles **one dimension** (time) with **one system of units**
- P3045 generalizes this to **arbitrary dimensions**, **quantity hierarchies**, and **multiple systems of quantities and units**
- The design is deliberately consistent with `std::chrono`:
  - same conversion safety model (no implicit truncation)
  - same value-preserving semantics
  - `quantity` interoperates with `duration`

`std::chrono` proves the case

- **`std::chrono`** handles **one dimension** (time) with **one system of units**
- P3045 generalizes this to **arbitrary dimensions**, **quantity hierarchies**, and **multiple systems of quantities and units**
- The design is deliberately consistent with **`std::chrono`**:
  - same conversion safety model (no implicit truncation)
  - same value-preserving semantics
  - **`quantity`** interoperates with **`duration`**

If **`std::chrono::duration`** was worth standardizing for time alone, then a general quantities and units library is worth standardizing for <u>every other physical dimension</u>.

Broad industry impact

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

Broad industry impact

Every industry listed on the previous slide writes C++ code that handles physical quantities using raw **`double`** values with informal unit conventions.

Broad industry impact

Every industry listed on the previous slide writes C++ code that handles physical quantities using raw `double` values with informal unit conventions.

- The **potential for error** is enormous
- The **cost of errors** ranges from financial loss to loss of life
- A **standard solution** would benefit the entire C++ ecosystem

Extensibility matters

Companies that cannot use open-source libraries (certification constraints) are forced to write their own solutions.

Extensibility matters

Companies that cannot use open-source libraries (certification constraints) are forced to write their own solutions.

- Writing a correct quantities and units library is **far from trivial**
- Getting the **design right** is even harder than the C++ code itself
- Even **domain experts** struggle to formalize their knowledge into a rigorous model
- **C++ experts** lack the domain knowledge to cover all corner cases

Extensibility matters

As a result companies either:

- use **simplistic wrappers** (losing most safety guarantees)
- **give up** and keep using `double` everywhere

Extensibility matters

As a result companies either:

- use **simplistic wrappers** (losing most safety guarantees)
- **give up** and keep using `double` everywhere

A standard library means nobody has to reinvent this — and custom domain extensions can build **on top of** the standard framework rather than from scratch.

Standardizing existing practice

This is not inventing something new. We are standardizing **25+ years** of existing practice.

Standardizing existing practice

This is not inventing something new. We are standardizing **25+ years** of existing practice.

- **1998**: Walter Brown presented the "SI Library of Unit-Based Computation" at CHEP
- Since then: Boost.Units, nholthaus/units, SI library, Au, mp-units, and many more

Standardizing existing practice

This is not inventing something new. We are standardizing 25+ years of existing practice.

- 1998: Walter Brown presented the "SI Library of Unit-Based Computation" at CHEP
- Since then: Boost.Units, nholthaus/units, SI library, Au, mp-units, and many more
- The **authors of the most popular C++ libraries joined forces** on this proposal
  - mp-units, nholthaus/units, SI library, Au — together **>90% of all GitHub stars** in C++ dimensional analysis

Standardizing existing practice

This is not inventing something new. We are standardizing 25+ years of existing practice.

- **1998**: Walter Brown presented the "SI Library of Unit-Based Computation" at CHEP
- Since then: Boost.Units, nholthaus/units, SI library, Au, mp-units, and many more
- The **authors of the most popular C++ libraries joined forces** on this proposal
  - mp-units, nholthaus/units, SI library, Au — together **>90% of all GitHub stars** in C++ dimensional analysis
- The core abstractions (dimension, unit, quantity, conversions) are well-understood and **converged across implementations**

Standardizing existing practice

Star History

mpusz/mp-units
nholthaus/units
bernedom/si
aurora-opensource/au
boostorg/units
martinmoene/physunits-ct-cpp11
loligothick/mitama-dimensional
gitsbi/unlib
jansende/benri
tonypilz/units
acodcha/phq
njoy/dimensionalanalysis

GitHub Stars
1200
1000
800
600
400
200
2014
2016
2018
2020
2022
2024
2026
Date
star-history.com

WG21 wants it

In Belfast 2019, LEWG took the following polls on P1935.

WG21 wants it

In Belfast 2019, LEWG took the following polls on P1935.

POLL: We should promise more committee time to pursuing **common units** (such as SI, customary, etc) to the standard library

SF	F	N	A	SA
11	7	4	2	0

WG21 wants it

In Belfast 2019, LEWG took the following polls on P1935.

POLL: We should promise more committee time to pursuing a standard library framework for user defined units and unit systems

SF	F	N	A	SA
10	8	4	1	1

WG21 wants it

In **Belfast 2019**, LEWG took the following polls on P1935.

POLL: We should promise more committee time to pursuing a standard library framework for user defined units and unit systems

SF	F	N	A	SA
10	8	4	1	1

Strong consensus on both polls. The authors formed a working group of experts from all major C++ units libraries per the Committee's direction.

REAL-WORLD PAIN POINTS

C++ developers need help

- Many C++ **engineers write life-critical software** today
- Experience in safety-critical domains is **hard to come by**
- **Training alone will not solve** mistakes caused by confusing units or quantities
- **Code** handling physical computations is **often written by domain experts** (e.g., physicists) **not necessarily fluent in C++**

C++ safety

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

C++ safety

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

A quantities and units library is one of the most impactful safety abstractions we can provide.

The proliferation of **double**

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

This is from a real open-source glide computer (LK8000). The function mixes distances, speeds, bearings (angles), time — all as **double**.

The proliferation of magic numbers

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

The proliferation of magic numbers

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

- What are the units of **hr**, **temp**, **abs_press**?
- What is **287.06**? What is **273.15**?
- Is **temp** in Celsius or Kelvin?

The proliferation of conversion macros

```
#ifndef PI
static const double PI = (4*atan(1));
#endif
#define EARTH_DIAMETER          12733426.0
#define NAUTICALMILESTOMETRES   (double)1851.96
#define KNOTSTOMETRESSECONDS    (double)0.5144

#define TOKNOTS            (double)1.944
#define TOFEETPERMINUTE    (double)196.9
#define TOMPH              (double)2.237
#define TOKPH              (double)3.6
#define TONAUTICALMILES    (1.0 / 1852.0)
#define TOMILES            (1.0 / 1609.344)
#define TOKILOMETER        (0.001)
#define TOFEET             (1.0 / 0.3048)
#define TOMETER            (1.0)
```

The proliferation of conversion macros

Often more than once in the same repository:

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

The proliferation of conversion macros

Often more than once in the same repository:

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

Four different definitions of the same conversion factor in one codebase. Note the subtle precision difference in the last one (**`57.2957805f`** vs **`57.2957795131`**).

Who sees the issue?

A set of functions from the same header file.

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

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

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

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

Typical production issues

PAIN POINTS AT CERN

The challenge at CERN

- **ATLAS Athena** framework: **2.5 million lines** of C++ code
- **Geant4** (simulation): base units are **mm, ns, MeV, rad**
- **ROOT** (analysis): base units are **cm, s, GeV, degree**
- **CLHEP** (older library): yet another set of conventions
- All use `double` for everything

The challenge at CERN

- **ATLAS Athena** framework: **2.5 million lines** of C++ code
- **Geant4** (simulation): base units are **mm, ns, MeV, rad**
- **ROOT** (analysis): base units are **cm, s, GeV, degree**
- **CLHEP** (older library): yet another set of conventions
- All use **`double`** for everything

When combining results across frameworks, developers must manually track which unit convention is in effect. Every **`double`** boundary is a potential factor-of-10 (or worse) error.

Argument soup: G4Trap (Geant4)

```
G4Trap(const G4String& pName,
       G4double pDz,          // ← length
       G4double pTheta,       // ← angle
       G4double pPhi,         // ← angle
       G4double pDy1,         // ← length
       G4double pDx1,         // ← length
       G4double pDx2,         // ← length
       G4double pAlp1,        // ← angle
       G4double pDy2,         // ← length
       G4double pDx3,         // ← length
       G4double pDx4,         // ← length
       G4double pAlp2);       // ← angle
```

Argument soup: G4Trap (Geant4)

```
G4Trap(const G4String& pName,
       G4double pDz,         // ← length
       G4double pTheta,      // ← angle
       G4double pPhi,        // ← angle
       G4double pDy1,        // ← length
       G4double pDx1,        // ← length
       G4double pDx2,        // ← length
       G4double pAlp1,       // ← angle
       G4double pDy2,        // ← length
       G4double pDx3,        // ← length
       G4double pDx4,        // ← length
       G4double pAlp2);      // ← angle
```

- **12 double parameters** mixing lengths and angles freely
- Swap any two adjacent parameters of the same kind → **compiles fine, wrong geometry**

Argument soup: TGeoMedium (ROOT)

```
TGeoMedium(const char *name, Int_t numed, Int_t nmat,
            Int_t isvol,      // ← flag
            Int_t ifield,     // ← flag
            Double_t fieldm,  // ← magnetic field
            Double_t tmaxfd,  // ← angle
            Double_t stemax,  // ← length
            Double_t deemax,  // ← energy
            Double_t epsil,   // ← length
            Double_t stmin);  // ← length
```

Argument soup: TGeoMedium (ROOT)

```
TGeoMedium(const char *name, Int_t numed, Int_t nmat,
           Int_t isvol,      // ← flag
           Int_t ifield,     // ← flag
           Double_t fieldm,  // ← magnetic field
           Double_t tmaxfd,  // ← angle
           Double_t stemax,  // ← length
           Double_t deemax,  // ← energy
           Double_t epsil,   // ← length
           Double_t stmin);  // ← length
```

- **6 Double_t** mixing field strength, angle, length, and energy
- No compiler protection against swapping them

Argument soup: G4ParticleDefinition

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width,
                     G4double charge, G4int iSpin, G4int iParity,
                     G4int iConjugation, G4int iIsospin, G4int iIsospin3,
                     G4int gParity, const G4String& pType, G4int lepton,
                     G4int baryon, G4int encoding, G4bool stable,
                     G4double lifetime, G4DecayTable *decaytable,
                     G4bool shortlived = false,
                     const G4String& subType = "",
                     G4int anti_encoding = 0,
                     G4double magneticMoment = 0.0);
```

Argument soup: `G4ParticleDefinition`

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width,
                     G4double charge, G4int iSpin, G4int iParity,
                     G4int iConjugation, G4int iIsospin, G4int iIsospin3,
                     G4int gParity, const G4String& pType, G4int lepton,
                     G4int baryon, G4int encoding, G4bool stable,
                     G4double lifetime, G4DecayTable *decaytable,
                     G4bool shortlived = false,
                     const G4String& subType = "",
                     G4int anti_encoding = 0,
                     G4double magneticMoment = 0.0);
```

- 21 parameters
- **`mass`** and **`width`** are adjacent **`G4double`**s

Argument soup: `G4ParticleDefinition`

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width,
                     G4double charge, G4int iSpin, G4int iParity,
                     G4int iConjugation, G4int iIsospin, G4int iIsospin3,
                     G4int gParity, const G4String& pType, G4int lepton,
                     G4int baryon, G4int encoding, G4bool stable,
                     G4double lifetime, G4DecayTable *decaytable,
                     G4bool shortlived = false,
                     const G4String& subType = "",
                     G4int anti_encoding = 0,
                     G4double magneticMoment = 0.0);
```

- 21 parameters
- **`mass`** and **`width`** are adjacent **`G4double`**s

Many parameters above should be strongly typed — **`quantity`** for physical values, dedicated types for quantum numbers.

Who sees the issue?

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

Who sees the issue?

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

"Maybe the unit is wrong?"

Who sees the issue?

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

"Maybe the unit is wrong?"

- `mass`: invariant mass — the numerical value represents **rest energy** $E = mc^2$ in MeV
- `width`: decay width Γ — related to particle lifetime by $\Gamma = \hbar/\tau$, expressed in MeV
- Both are just **G4double in MeV** — numerically identical types

Who sees the issue?

```cpp
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```cpp
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

Who sees the issue?

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

- **0.321 MeV mass** is lighter than an electron (0.511 MeV) — **impossible for a baryon**
- **938 MeV width** means near-instantaneous decay — **physically nonsensical**

Who sees the issue?

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);
```

- **0.321 MeV mass** is lighter than an electron (0.511 MeV) — **impossible for a baryon**
- **938 MeV width** means near-instantaneous decay — **physically nonsensical**

Swapping `mass` ↔ `width` compiles perfectly. Caught only in physics validation (if at all).

Potential HEP bug: parameter swapping

```
G4ParticleDefinition(const G4String& aName, G4double mass, G4double width, ...);
```

```
G4ParticleDefinition("MyParticle", 0.321 * MeV, 938.272 * MeV, /* ... */);  // Bug
```

```
G4ParticleDefinition("MyParticle", 938.272 * MeV, 0.321 * MeV, /* ... */);  // Correct
```

The "Magic 1.0" problem — at scale

- We saw the Geant4 ⟶ ROOT conversion trap earlier
- In ATLAS Athena's **2.5 million lines of code**, this pattern appears **everywhere**
- Every framework boundary is a potential factor-of-10 error

The "Magic 1.0" problem — at scale

- We saw the Geant4 → ROOT conversion trap earlier
- In ATLAS Athena's **2.5 million lines of code**, this pattern appears **everywhere**
- Every framework boundary is a potential factor-of-10 error

FRAMEWORK PAIR	COMMON MISTAKE	EFFECT
Geant4 → ROOT	missing `/ CLHEP::cm`	10× geometry
ROOT → Geant4	missing `* CLHEP::cm`	0.1× geometry
CLHEP → ROOT	wrong CODATA vintage	subtle drift

The "Magic 1.0" problem — at scale

- We saw the Geant4 → ROOT conversion trap earlier
- In ATLAS Athena's **2.5 million lines of code**, this pattern appears **everywhere**
- Every framework boundary is a potential factor-of-10 error

FRAMEWORK PAIR	COMMON MISTAKE	EFFECT
Geant4 → ROOT	missing `/ CLHEP::cm`	10× geometry
ROOT → Geant4	missing `* CLHEP::cm`	0.1× geometry
CLHEP → ROOT	wrong CODATA vintage	subtle drift

This is a **silent 10× error** in detector geometry. No compiler warning, no runtime exception.

CODATA drift problem

CONSTANT	CLHEP (2014)	GEANT4 (2018)	ROOT (2022)
electron mass (MeV/c^2)	0.5109989461	0.5109989500	0.51099895069

CODATA drift problem

CONSTANT	CLHEP (2014)	GEANT4 (2018)	ROOT (2022)
electron mass (MeV/c^2)	0.5109989461	0.5109989500	0.51099895069

- Three frameworks use **different CODATA vintages**
- Mixing constants from different versions introduces **subtle inconsistencies**
- Results may differ in the **8th–10th significant digit**
- For precision measurements this **matters**

CODATA drift problem

CONSTANT	CLHEP (2014)	GEANT4 (2018)	ROOT (2022)
electron mass (MeV/c^2)	0.5109989461	0.5109989500	0.51099895069

- Three frameworks use **different CODATA vintages**
- Mixing constants from different versions introduces **subtle inconsistencies**
- Results may differ in the **8th–10th significant digit**
- For precision measurements this **matters**

A type-safe library can enforce consistent CODATA version selection at compile time.

Metrology is an inherently complex problem

Metrology is an inherently complex problem

The problems shown are not theoretical. They exist in production code at CERN, in aviation software, in navigation systems.

Metrology is an inherently complex problem

The problems shown are not theoretical. They exist in production code at CERN, in aviation software, in navigation systems.

- The proliferation of `double` is **universal**
- Conversion errors are **pervasive**
- Domain complexity is **real and unavoidable**
- **No amount of code review or training** can eliminate these classes of bugs
- Only the **type system** can provide systematic protection

THE EVOLUTION OF UNITS LIBRARIES

Why do we need units libraries?

If we never made unit errors, a units library would just add compile time. But we are human and we make errors — and the cost of undetected unit bugs can be catastrophic.

Why do we need units libraries?

If we never made unit errors, a units library would just add compile time. But we are human and we make errors — and the cost of undetected unit bugs can be catastrophic.

The goal of every units library is to catch quantities-related errors at compile time, so developers can move faster with confidence and never get conversions wrong.

Two things matter most

1. **Error messages** must be understandable — even for non-expert users
2. **Syntax** must stay out of the way — too much boilerplate defeats the purpose

Two things matter most

1. **Error messages** must be understandable — even for non-expert users
2. **Syntax** must stay out of the way — too much boilerplate defeats the purpose

Nobody will use a library that prevents bugs but is too painful to type.
Nobody will use a library whose error messages they cannot understand.

How it all started?

Before 1998, libraries like CLHEP used simple `double` with unit multipliers as constants. No dimensional types — everything was just `double`.

How it all started?

Before 1998, libraries like CLHEP used simple **`double`** with unit multipliers as constants. No dimensional types — everything was just **`double`**.

And they still do...

How it all started?

```cpp
// Modern HEP frameworks (Geant4, ROOT, CLHEP) — still just doubles today:

// From Geant4 SystemOfUnits.h:
static constexpr double millimeter  = 1.;
static constexpr double meter       = 1000. * millimeter;
static constexpr double kilometer   = 1000. * meter;
static constexpr double nanosecond  = 1.;
static constexpr double second      = 1.e+9 * nanosecond;

// From ROOT TMath.h:
inline Double_t Pi()       { return 3.14159265358979323846; }
inline Double_t DegToRad() { return Pi() / 180.0; }

// Usage in production code:
G4double distance = 5.0 * kilometer;    // Just a double: 5000.0
G4double time = 2.0;                    // Just a double: 2.0 (ns? s?)
G4double speed = distance / time;       // Just a double: 2500.0 (mm/ns? m/s?)
```

How it all started?

```cpp
// Modern HEP frameworks (Geant4, ROOT, CLHEP) — still just doubles today:

// From Geant4 SystemOfUnits.h:
static constexpr double millimeter  = 1.;
static constexpr double meter       = 1000. * millimeter;
static constexpr double kilometer   = 1000. * meter;
static constexpr double nanosecond  = 1.;
static constexpr double second      = 1.e+9 * nanosecond;

// From ROOT TMath.h:
inline Double_t Pi()       { return 3.14159265358979323846; }
inline Double_t DegToRad() { return Pi() / 180.0; }

// Usage in production code:
G4double distance = 5.0 * kilometer;    // Just a double: 5000.0
G4double time = 2.0;                    // Just a double: 2.0 (ns? s?)
G4double speed = distance / time;       // Just a double: 2500.0 (mm/ns? m/s?)
```

- **No compile-time safety**: `distance + time` compiles (adds 5000.0 + 2.0)
- **Lost dimensionality**: after operations, everything is just a `double`
- **Framework confusion**: Geant4 uses mm/ns, ROOT uses cm/s — mixing them is silent error

Walter Brown's SI Library (1998)

FERMILAB-Conf-98/328

Introduction to the SI Library of Unit-Based Computation

Walter E. Brown

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

Walter Brown's SI Library (1998)

FERMILAB-Conf-98/328

Introduction to the SI Library of Unit-Based Computation

Walter E. Brown

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

Walter Brown presented "SI Library of Unit-Based Computation" at CHEP '98 — the first systematic approach to compile-time dimensional analysis in C++.

Walter Brown's SI Library (1998)

```
Length d2( 2.5 );                   // 2.5 meters; same as 2.5 * m
Length d3( 1.2 * cm );              // 1.2 centimeters; recorded as .012 * m
Length d6 = d2 + d3;                // ✓ all dimensions match

Length len( 2*cm );
Width  wid( 3*cm );                 // Width is a synonym for Length
Area   a( len * wid );              // ✓ Length × Width = Area

// Bad instantiation attempts; all detected at compile-time:
// Length d7( d2 * d3 );            // ❌ oops: an Area can't initialize a Length
// Length d8 = 3.5;                 // ❌ oops: 3.5 is not a Length
```

Walter Brown's SI Library (1998)

```
Length d2( 2.5 );              // 2.5 meters; same as 2.5 * m
Length d3( 1.2 * cm );         // 1.2 centimeters; recorded as .012 * m
Length d6 = d2 + d3;           // ✓ all dimensions match

Length len( 2*cm );
Width  wid( 3*cm );            // Width is a synonym for Length
Area   a( len * wid );         // ✓ Length × Width = Area

// Bad instantiation attempts; all detected at compile-time:
// Length d7( d2 * d3 );       // ❌ oops: an Area can't initialize a Length
// Length d8 = 3.5;            // ❌ oops: 3.5 is not a Length
```

First library to provide dimensional type safety with zero runtime overhead.

Boost.Units: Pioneering compile-time dimensional analysis

Boost.Units first released nearly 20 years ago was a **pioneering** library — the first widely-adopted generic C++ framework providing **compile-time** dimensional analysis, value truncation prevention, and derived quantity equations at **zero overhead**.

Boost.Units: Pioneering compile-time dimensional analysis

Boost.Units first released nearly 20 years ago was a **pioneering** library — the first widely-adopted generic C++ framework providing **compile-time** dimensional analysis, value truncation prevention, and derived quantity equations at **zero overhead**.

A groundbreaking achievement built within **C++03** constraints.

The main goal: generate errors (Boost.Units)

```
// Define kilometer unit
using kilometer_base_unit =
  bu::make_scaled_unit<bu::si::length, bu::scale<10, bu::static_rational<3>>>::type;
using length_kilometer = kilometer_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(kilometers, length_kilometer);

// Define hour unit
using time_hour = bu::metric::hour_base_unit::unit_type;
BOOST_UNITS_STATIC_CONSTANT(hours, time_hour);

// Define km/h unit
using kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
```

The main goal: generate errors (Boost.Units)

```cpp
bu::quantity<bu::si::time> time_to_goal(bu::quantity<bu::si::length> distance,
                                        bu::quantity<kilometers_per_hour> speed)
{
  return bu::quantity<bu::si::time>(distance / speed);
}
```

```cpp
bu::quantity<bu::si::length> distance(110.0 * bu::si::meters);
bu::quantity<kilometers_per_hour> speed(90.0 * kilometers / hours);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

The main goal: generate errors (Boost.Units)

```cpp
bu::quantity<bu::si::time> time_to_goal(bu::quantity<bu::si::length> distance,
                                        bu::quantity<kilometers_per_hour> speed)
{
  return bu::quantity<bu::si::time>(distance / speed);
}
```

```cpp
bu::quantity<bu::si::length> distance(110.0 * bu::si::meters);
bu::quantity<kilometers_per_hour> speed(90.0 * kilometers / hours);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

```
<source>: In function 'int main()':
<source>:33:28: error: could not convert 'speed' from 'quantity<unit<list<[...],boost::units::list<boost::units::dim<boost::units::time_base_dim
   33 |   auto time = time_to_goal(speed, distance);
      |                            ^~~~~
      |                            |
      |                            quantity<unit<list<[...],boost::units::list<boost::units::dim<boost::units::time_base_dimension, boost::units
```

The main goal: generate errors (Boost.Units)

```cpp
bu::quantity<bu::si::time> time_to_goal(bu::quantity<bu::si::length> distance,
                                        bu::quantity<kilometers_per_hour> speed)
{
  return bu::quantity<bu::si::time>(distance / speed);
}
```

```cpp
bu::quantity<bu::si::length> distance(110.0 * bu::si::meters);
bu::quantity<kilometers_per_hour> speed(90.0 * kilometers / hours);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

```
<source>: In function 'int main()':
<source>:33:28: error: could not convert 'speed' from 'quantity<unit<list<[...],boost::units::list<boost::units::dim<
boost::units::time_base_dimension, boost::units::static_rational<-1> >, boost::units::dimensionless_type>>,
boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list<
boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >,
boost::units::list<boost::units::heterogeneous_system_dim<boost::units::scaled_base_unit<boost::units::si::second_base_unit,
boost::units::scale<60, boost::units::static_rational<2> > >, boost::units::static_rational<-1> >, boost::units::dimensionless_type> >,
boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1> >,
boost::units::list<boost::units::dim<boost::units::time_base_dimension, boost::units::static_rational<-1> >, boost::units::dimensionless_type> >,
boost::units::list<boost::units::scale_list_dim<boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::dimensionless_type> > >,
[...]>>' to 'quantity<unit<list<[...],boost::units::dimensionless_type>,boost::units::homogeneous_system<boost::units::list<
boost::units::si::meter_base_unit, boost::units::list<boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit,
boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::list<boost::units::si::second_base_unit,
boost::units::list<boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit,
boost::units::list<boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit,
boost::units::list<boost::units::angle::radian_base_unit, boost::units::list<boost::units::angle::steradian_base_unit,
boost::units::dimensionless_type> > > > > > > > >,[...]>>'
   33 |   auto time = time_to_goal(speed, distance);
```

The main goal: generate errors (Boost.Units)

```cpp
bu::quantity<bu::si::time> time_to_goal(bu::quantity<bu::si::length> distance,
                                        bu::quantity<kilometers_per_hour> speed)
{
  return bu::quantity<bu::si::time>(distance / speed);
}
```

```cpp
bu::quantity<bu::si::length> distance(110.0 * bu::si::meters);
bu::quantity<kilometers_per_hour> speed(90.0 * kilometers / hours);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

```
<source>: In function 'int main()':
<source>:33:28: error: could not convert 'speed' from 'quantity<unit<list<[...],boost::units::list<boost::units::dim<boost::units::time_base_dimension,
boost::units::static_rational<-1> >, boost::units::dimensionless_type>>,boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list<
boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >,boost::units::list<boost::units::heterogeneous_system_dim<
boost::units::scaled_base_unit<boost::units::si::second_base_unit, boost::units::scale<60, boost::units::static_rational<2> > >, boost::units::static_rational<-1> >,
boost::units::dimensionless_type> >, boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1> >,
boost::units::list<boost::units::dim<boost::units::time_base_dimension, boost::units::static_rational<-1> >, boost::units::dimensionless_type> >,
boost::units::list<boost::units::scale_list_dim<boost::units::scale<10, boost::units::static_rational<3> > >, boost::units::dimensionless_type> > >,
[...]>>' to 'quantity<unit<list<[...],boost::units::dimensionless_type>,boost::units::homogeneous_system<boost::units::list<boost::units::si::meter_base_unit,
boost::units::list<boost::units::scaled_base_unit<boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit, boost::units::list<boost::units::si::kelvin_base_unit,
boost::units::list<boost::units::si::mole_base_unit, boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > >,[...]>>'
   33 |   auto time = time_to_goal(speed, distance);
      |                            ^~~~~
      |                            |
      |                            quantity<unit<list<[...],boost::units::list<boost::units::dim<boost::units::time_base_dimension, boost::units::static_rational<-1> >,
                                   boost::units::dimensionless_type>>, boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list<
                                   boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >, boost::units::list<
                                   boost::units::heterogeneous_system_dim<boost::units::scaled_base_unit<boost::units::si::second_base_unit, boost::units::scale<
                                   60, boost::units::static_rational<2> > >, boost::units::static_rational<-1> >, boost::units::dimensionless_type> >,
                                   boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1> >,
                                   boost::units::list<boost::units::dim<boost::units::time_base_dimension, boost::units::static_rational<-1> >,
                                   boost::units::dimensionless_type> >, boost::units::list<boost::units::scale_list_dim<boost::units::scale<10,
                                   boost::units::static_rational<3> > >, boost::units::dimensionless_type> > >,[...]>>
```

Boost.Units — what changed?

The verbose errors and boilerplate stem from **both C++03 constraints and design choices** that made sense at the time. It took the community **20 years** to learn what works well and what doesn't.

Boost.Units — what changed?

The verbose errors and boilerplate stem from **both C++03 constraints and design choices** that made sense at the time. It took the community **20 years** to learn what works well and what doesn't.

With today's knowledge, we could probably design a better library even within C++98 constraints. But we didn't have that knowledge in 2008.

nholthaus/units (2015): Learning from Boost.Units

~10 years after Boost.Units, nholthaus/units by Nick Holthaus (now P3045 co-author) improved **scaled and derived unit handling** using C++14.

The main goal: generate errors (nholthaus/units)

```
time::second_t time_to_goal(length::meter_t distance,
                            velocity::kilometers_per_hour_t speed)
{
  return time::second_t(distance / speed);
}
```

```
length::meter_t distance = 110.0_m;
velocity::kilometers_per_hour_t speed(90.0);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

The main goal: generate errors (nholthaus/units)

```
time::second_t time_to_goal(length::meter_t distance,
                            velocity::kilometers_per_hour_t speed)
{
  return time::second_t(distance / speed);
}
```

```
length::meter_t distance = 110.0_m;
velocity::kilometers_per_hour_t speed(90.0);
auto duration = time_to_goal(speed, distance);  // Wrong order
```

```
error: static assertion failed due to requirement 'units::traits::is_convertible_unit<units::unit<std::ratio<5, 18>, units::base_unit<std::ratio<1, 1>,
  std::ratio<0, 1>, std::ratio<-1, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>>,
  std::ratio<0, 1>, std::ratio<0, 1>>, units::unit<std::ratio<1, 1>, units::base_unit<std::ratio<1, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>,
  std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<0, 1>>, std::ratio<0, 1>, std::ratio<0, 1>>>::value':
  Units are not compatible.
 1652 |                 static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
      |                               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in instantiation of function template specialization 'units::convert<units::unit<std::ratio<5, 18>, units::base_unit<std::ratio<1>, std::ratio<0>,
  std::ratio<-1>>>, units::unit<std::ratio<1>, units::base_unit<std::ratio<1>>>, double>' requested here
 1989 |                 nls(units::convert<UnitsRhs, Units, T>(rhs.m_value), std::true_type() /*store linear value*/)
      |                            ^
note: in instantiation of function template specialization 'units::unit_t<units::unit<std::ratio<1>,
  units::base_unit<std::ratio<1>>>>::unit_t<units::unit<std::ratio<5, 18>, units::base_unit<std::ratio<1>, std::ratio<0>, std::ratio<-1>>>, double,
  units::linear_scale>' requested here
   16 |   auto duration = time_to_goal(speed, distance);  // Wrong order
      |
```

The evolution: what we learned

- **CLHEP** (pre-1998): fast, simple — but **no safety at all**
- **Walter Brown SI** (1998): first **type safety at zero cost** — limited scope, storage in base units
- **Boost.Units** (2008): pioneering **generic framework** — verbose errors, heavy syntax
- **nholthaus/units** (2015): **simpler composition** — still challenging errors

The evolution: what we learned

- **CLHEP** (pre-1998): fast, simple — but **no safety at all**
- **Walter Brown SI** (1998): first **type safety at zero cost** — limited scope, storage in base units
- **Boost.Units** (2008): pioneering **generic framework** — verbose errors, heavy syntax
- **nholthaus/units** (2015): **simpler composition** — still challenging errors

Since then, C++ gained:

- **Concepts** (C++20) — readable diagnostics
- **CTAD** (C++17) — less type spelling
- **NTTP** (C++20) — class object values as template parameters

The evolution: what we learned

- **CLHEP** (pre-1998): fast, simple — but **no safety at all**
- **Walter Brown SI** (1998): first **type safety at zero cost** — limited scope, storage in base units
- **Boost.Units** (2008): pioneering **generic framework** — verbose errors, heavy syntax
- **nholthaus/units** (2015): **simpler composition** — still challenging errors

Since then, C++ gained:

- **Concepts** (C++20) — readable diagnostics
- **CTAD** (C++17) — less type spelling
- **NTTP** (C++20) — class object values as template parameters

These features enable fundamentally better library designs with cleaner syntax and more helpful errors.

The main goal: generate errors (mp-units)

```
quantity<si::second> time_to_goal(quantity<si::metre> distance,
                                  quantity<si::kilo<si::metre> / si::hour> speed)
{
  return distance / speed;
}
```

```
quantity distance = 110. * m;
quantity speed = 90. * km / h;
quantity duration = time_to_goal(speed, distance);  // Wrong order
```

The main goal: generate errors (mp-units)

```
quantity<si::second> time_to_goal(quantity<si::metre> distance,
                                  quantity<si::kilo<si::metre> / si::hour> speed)
{
  return distance / speed;
}
```

```
quantity distance = 110. * m;
quantity speed = 90. * km / h;
quantity duration = time_to_goal(speed, distance);  // Wrong order
```

```
error: no matching function for call to 'time_to_goal'
   16 |   quantity duration = time_to_goal(speed, distance);
      |                       ^~~~~~~~~~~~
candidate function not viable: no known conversion from
  'quantity<mp_units::derived_unit<mp_units::si::kilo_<mp_units::si::metre>, per<mp_units::non_si::hour>>{}, [...]>'
  to 'quantity<struct metre{}, [...]>' for 1st argument
    6 | quantity<si::second> time_to_goal(quantity<si::metre> distance,
      |                      ^            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The main goal: generate errors (mp-units)

```cpp
quantity<si::second> time_to_goal(quantity<si::metre> distance,
                                  quantity<si::kilo<si::metre> / si::hour> speed)
{
  return distance / speed;
}
```

```cpp
quantity distance = 110. * m;
quantity speed = 90. * km / h;
quantity duration = time_to_goal(speed, distance);  // Wrong order
```

```
error: no matching function for call to 'time_to_goal'
   16 |   quantity duration = time_to_goal(speed, distance);
      |                       ^~~~~~~~~~~
candidate function not viable: no known conversion from
  'quantity<mp_units::derived_unit<mp_units::si::kilo_<mp_units::si::metre>, per<mp_units::non_si::hour>>{}, [...]>'
  to 'quantity<struct metre{}, [...]>' for 1st argument
    6 | quantity<si::second> time_to_goal(quantity<si::metre> distance,
```

Code written by the user can be easily mapped to a generated type.

Nearly 30 years later...

In **1998**, Walter Brown showed HEP how to escape the `double` trap. Nearly **30 years later**, safety-critical projects like Geant4, ROOT, and CLHEP — with millions of lines of C++ simulating particle physics, medical imaging, and radiation protection — **still pass every physical quantity as a raw double**.

Nearly 30 years later…

In **1998**, Walter Brown showed HEP how to escape the **double** trap. Nearly **30 years later**, safety-critical projects like Geant4, ROOT, and CLHEP — with millions of lines of C++ simulating particle physics, medical imaging, and radiation protection — **still pass every physical quantity as a raw double**.

- The libraries existed. The knowledge existed. The bugs kept happening.
- Third-party solutions were **not enough** to change an ecosystem.

Nearly 30 years later...

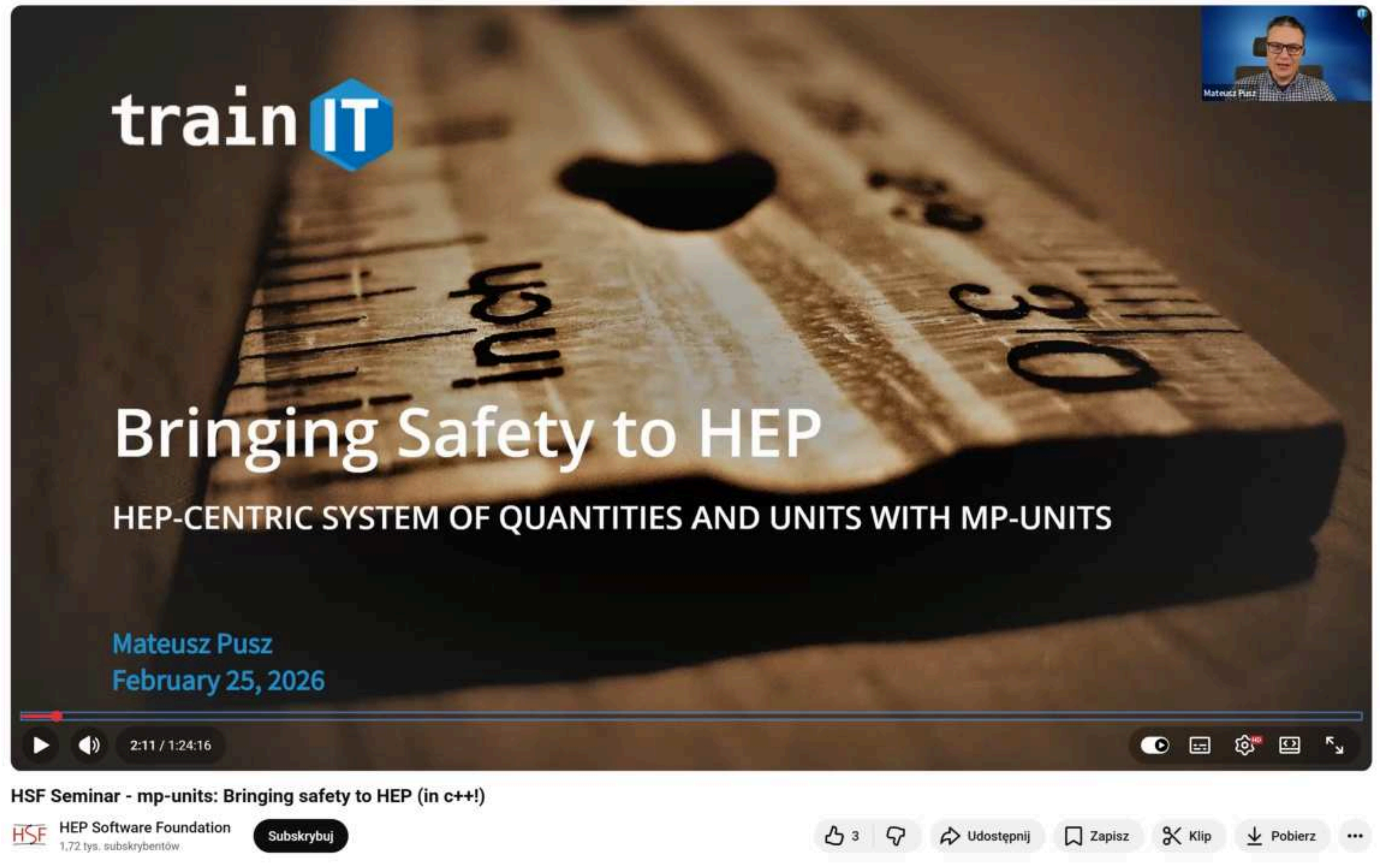
In **1998**, Walter Brown showed HEP how to escape the **`double`** trap. Nearly **30 years later**, safety-critical projects like Geant4, ROOT, and CLHEP — with millions of lines of C++ simulating particle physics, medical imaging, and radiation protection — **still pass every physical quantity as a raw `double`**.

- The libraries existed. The knowledge existed. The bugs kept happening.
- Third-party solutions were **not enough** to change an ecosystem.

Let's standardize it — so there is no excuse left.

Let's say "NO!" to raw **`double`** for units!

HEP considers moving forward



MP-UNITS & P3045

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

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

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

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

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

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

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

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

ISO C++ papers

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

Dependencies on other proposals

PRIORITY 1 (MANDATORY)

- P3094R6: `std::basic_fixed_string`

Dependencies on other proposals

PRIORITY 1 (MANDATORY)

- P3094R6: `std::basic_fixed_string`

PRIORITY 2 (FUNCTIONAL EXTENSIONS)

- P2822R2: User control of associated entities
- P0870R5, P2509R1: Preventing value truncation
- P1729R5: Text Parsing
- P3380R1: Extending NTTPs
- P3679R0: SFINAEable constexpr exceptions
- P2989R2: A Simple Approach to Universal Template Parameters
- P3003R0: Number concepts

`quantity` class template

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity;
```

quantity class template

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity;
```

Quantity is a strongly-typed numerical wrapper that annotates the user's representation type (**Rep**) with a **Reference** (a unit and quantity-related metadata).

`quantity` class template

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity;
```

- `Reference` might be either
  - `si::watt`, `si::metre / si::second`
  - `isq::kinetic_energy[J]`

`quantity` class template

```
template<Reference auto R,
         RepresentationOf<get_quantity_spec(R)> Rep = double>
class quantity;
```

- **`Reference`** might be either
  - **`si::watt`**, **`si::metre / si::second`**
  - **`isq::kinetic_energy[J]`**
- **`RepresentationOf`** ensures that the provided type *satisfies quantity character* requirements
  - real scalar
  - complex scalar
  - vector
  - tensor

Syntax motivation from ISO standards

> The value of the quantity is the **product of the number and the unit**. **The space** between the number and the unit **is regarded as a multiplication sign** (just as a space between units implies multiplication).
>
> *-- SI Brochure*

Syntax motivation from ISO standards

> The value of the quantity is the **product of the number and the unit**. **The space** between the number and the unit **is regarded as a multiplication sign** (just as a space between units implies multiplication).
>
> *-- SI Brochure*

```
quantity q1 = 42 * m / s;
```

Alternative creation syntaxes

If multiply syntax is not preferred, there are alternatives.

Alternative creation syntaxes

If multiply syntax is not preferred, there are alternatives.

USING TWO-PARAMETER CONSTRUCTOR

```
quantity q2{42, m / s};
```

Alternative creation syntaxes

If multiply syntax is not preferred, there are alternatives.

USING TWO-PARAMETER CONSTRUCTOR

```
quantity q2{42, m / s};
```

USING THE DELTA HELPER

```
quantity q3 = delta<m / s>(42);
```

Alternative creation syntaxes

If multiply syntax is not preferred, there are alternatives.

USING TWO-PARAMETER CONSTRUCTOR

```
quantity q2{42, m / s};
```

USING THE DELTA HELPER

```
quantity q3 = delta<m / s>(42);
```

Type: `quantity<derived_unit<si::metre, per<si::second>>{}, int>`.

Why not user-defined literals (UDLs)?

A common question: "Why not use `42m_per_s` or `42mps`?"

Why not user-defined literals (UDLs)?

LIMITED APPLICABILITY

- UDLs work **only with literals** (compile-time values)
- Production code rarely has compile-time known quantities
- Most quantities come *from calculations, I/O, or function parameters*
- **We would need an alternative** for runtime arguments anyway

Why not user-defined literals (UDLs)?

LIMITED APPLICABILITY

- UDLs work **only with literals** (compile-time values)
- Production code rarely has compile-time known quantities
- Most quantities come *from calculations, I/O, or function parameters*
- **We would need an alternative** for runtime arguments anyway

NAMING COLLISIONS

- Many unit symbols conflict with **built-in types**
  - `F` (farad), `J` (joule), `W` (watt), `K` (kelvin), `d` (day), `l` or `L` (litre), `erg`

Why not user-defined literals (UDLs)?

TYPE BLOAT

- UDLs typically use **widest representation types**

```
using namespace std::chrono_literals;
auto d1 = 42s;    // std::int64_t
auto d2 = 42.s;   // long double
```

- *Long types propagate* through all arithmetic
- No way to control representation type at creation

Why not user-defined literals (UDLs)?

NAMESPACE AMBIGUITY

- UDLs with **same names can't be disambiguated**

```
namespace si  { inline constexpr auto operator""s(...); }
namespace cgs { inline constexpr auto operator""s(...); }

using namespace si;
using namespace cgs;
auto t = 42s;   // ❌ Ambiguous – which second?
```

Why not user-defined literals (UDLs)?

POOR COMPOSABILITY

- Would need UDLs for **every unit combination**

```
inline constexpr auto operator""kg_m2_per_s(unsigned long long);
inline constexpr auto operator""kg_m2_per_s(long double);

// Now do the same for every possible combination of scaled units:
// g_cm2_per_s, g_m2_per_ms, kg_km2_per_h, ...
```

Why not user-defined literals (UDLs)?

POOR COMPOSABILITY

- Would need UDLs for **every unit combination**

```
inline constexpr auto operator""kg_m2_per_s(unsigned long long);
inline constexpr auto operator""kg_m2_per_s(long double);

// Now do the same for every possible combination of scaled units:
// g_cm2_per_s, g_m2_per_ms, kg_km2_per_h, ...
```

This would require **thousands of UDL definitions** and still not cover all cases.

Unit symbols are opt-in

Unit symbols like `m`, `s`, `kg` introduce many short identifiers.

Unit symbols are opt-in

Unit symbols like `m`, `s`, `kg` introduce many short identifiers.

- They can cause **naming collisions** with existing code
- This is why they are **opt-in** via namespace imports

Unit symbols are opt-in

Unit symbols like **m**, **s**, **kg** introduce many short identifiers.

- They can cause **naming collisions** with existing code
- This is why they are **opt-in** via namespace imports

```
void foo(double speed_m_s)
{
  using namespace std::si::unit_symbols;  // Import symbols locally
  std::quantity speed = speed_m_s * m / s;
  // ...
}
```

Unit symbols are opt-in

Unit symbols like **`m`**, **`s`**, **`kg`** introduce many short identifiers.

- They can cause **naming collisions** with existing code
- This is why they are **opt-in** via namespace imports

```
void foo(double speed_m_s)
{
  using namespace std::si::unit_symbols;  // Import symbols locally
  std::quantity speed = speed_m_s * m / s;
  // ...
}
```

Import unit symbols only in the scope where they're needed.

API at a glance

```
import std;
using namespace std::si::unit_symbols;

quantity distance = 110. * km;
quantity time = 2. * h;
quantity speed = distance / time;

std::println("Speed: {}", speed);
std::println("Speed: {::N[.4]}", speed.in(m / s));
```

```
Speed: 55 km/h
Speed: 15.28 m/s
```

The main goal: generate errors

```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;
const quantity time = (distance * speed).in(s);
```

The main goal: generate errors

```cpp
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;
const quantity time = (distance * speed).in(s);
```

```
error: no matching member function for call to 'in'
   18 | const quantity time = (distance * speed).in(s);
      |                       ~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: couldn't infer template argument 'ToRep'
  359 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
note: candidate template ignored: constraints not satisfied [with ToU = struct second]
  345 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
note: because 'unit_of<struct second, quantity_spec>' evaluated to false
  343 |   template<unit_of<quantity_spec> ToU>
      |            ^
note: because 'std::implicitly_convertible(get_quantity_spec(std::si::second{}),
  std::invoke_result_t<std::multiplies<>, decltype(quantity<struct metre{}, double>::quantity_spec),
  decltype(quantity<std::derived_unit<std::yard_pound::mile, std::per<std::non_si::hour>>{}, double>::quantity_spec)>)'
  evaluated to false
   67 |   (std::detail::implicitly_convertible(get_quantity_spec(U{}), QS) ||
      |    ^
...
```

The main goal: generate errors (P3679)

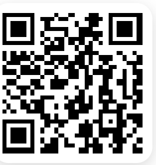
```
const quantity distance = 30. * m;
const quantity speed = 25. * mi / h;
const quantity time = (distance * speed).in(s);
```

```
error: no matching member function for call to 'in'
   18 | const quantity time = (distance * speed).in(s);
      |                       ~~~~~~~~~~~~~~~~~~~~^~
note: candidate template ignored: couldn't infer template argument 'ToRep'
  359 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
note: candidate template ignored: constraints not satisfied via exception
  "Unit 's' is associated with quantity of kind 'std::isq::duration' which is not convertible to
   the 'std::kind_of_<std::derived_quantity_spec<std::power<std::isq::length, 2>, std::per<std::isq::duration>>>' quantity"
  [with ToU = struct second]
  345 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
...
```

The main goal: generate errors (P3679)

```
quantity q1 = 42 * m;
quantity q2 = q1.in(km);
```

The main goal: generate errors (P3679)



```cpp
quantity q1 = 42 * m;
quantity q2 = q1.in(km);
```

```
error: no matching member function for call to 'in'
    9 |   quantity q2 = q1.in(km);
      |                 ~~~^~
note: candidate template ignored: couldn't infer template argument 'ToRep'
  359 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
note: candidate template ignored: constraints not satisfied via exception
  "Scaling from 'm' to 'km' is truncating for 'int' representation type"
  [with ToU = kilo_<decltype(metre{{{}}})>]
  345 |   [[nodiscard]] constexpr quantity_of<quantity_spec> auto in(ToU) const
      |                                                           ^
...
```

As fast as double

NO UNITS LIBRARY

```
double ttg_s(double d_m, double v_mps)
{
  return d_m / v_mps;
}
```

MP-UNITS

```
quantity<s> ttg(quantity<m> d, quantity<m/s> v)
{
  return d / v;
}
```

As fast as double

NO UNITS LIBRARY

```
double ttg_s(double d_m, double v_mps)
{
  return d_m / v_mps;
}
```

```
ttg_s(double, double):
  divsd xmm0, xmm1
  ret
```

MP-UNITS

```
quantity<s> ttg(quantity<m> d, quantity<m/s> v)
{
  return d / v;
}
```

```
ttg(quantity<m>, quantity<m/s>):
  divsd xmm0, xmm1
  ret
```

As fast as double

NO UNITS LIBRARY

```
double ttg_s(double d_m, double v_mps)
{
  return d_m / v_mps;
}
```

```
ttg_s(double, double):
  divsd xmm0, xmm1
  ret
```

MP-UNITS

```
quantity<s> ttg(quantity<m> d, quantity<m/s> v)
{
  return d / v;
}
```

```
ttg(quantity<m>, quantity<m/s>):
  divsd xmm0, xmm1
  ret
```

Identical assembly. Units are a compile-time-only concept.

As fast as double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::mm;
  return length / CLHEP::cm;
}
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * mm;
  return length.numerical_value_in(cm);
}
```

As fast as double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::mm;
  return length / CLHEP::cm;
}
```

```
.LCPI0_0:
  .quad    0x4024000000000000
foo(double):
  divsd    xmm0, qword ptr [rip + .LCPI0_0]
  ret
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * mm;
  return length.numerical_value_in(cm);
}
```

```
.LCPI0_0:
  .quad    0x4024000000000000
foo(double):
  divsd    xmm0, qword ptr [rip + .LCPI0_0]
  ret
```

As fast as double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::mm;
  return length / CLHEP::mm;
}
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * mm;
  return length.numerical_value_in(mm);
}
```

As fast as double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::mm;
  return length / CLHEP::mm;
}
```

```
foo(double):
  ret
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * mm;
  return length.numerical_value_in(mm);
}
```

```
foo(double):
  ret
```

Faster than double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::cm;
  return length / CLHEP::cm;
}
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * cm;
  return length.numerical_value_in(cm);
}
```

Faster than double

NO UNITS LIBRARY

```
auto foo(double val)
{
  double length = val * CLHEP::cm;
  return length / CLHEP::cm;
}
```

```
.LCPI0_0:
  .quad    0x4024000000000000
foo(double):
  movsd    xmm1, qword ptr [rip + .LCPI0_0]
  mulsd    xmm0, xmm1
  divsd    xmm0, xmm1
  ret
```

MP-UNITS

```
auto foo(double val)
{
  quantity length = val * cm;
  return length.numerical_value_in(cm);
}
```

```
foo(double):
  ret
```

Generic programming

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

Generic programming

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

```
// Works with any compatible units — no conversions needed at call site
quantity d1 = time_to_goal(400. * ft, 40. * mi / h);      // 10 h ft/mi
quantity d2 = time_to_goal(1512. * km, 95. * km / h);     // 15.9 h
```

Generic programming

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

```cpp
// Works with any compatible units — no conversions needed at call site
quantity d1 = time_to_goal(400. * ft, 40. * mi / h);      // 10 h ft/mi
quantity d2 = time_to_goal(1512. * km, 95. * km / h);     // 15.9 h
```

Generic interfaces backed by concepts and CTAD perform the most efficient calculations without unnecessary unit rescaling — while still being fully type-safe.

Metrology is an inherently complex problem

Metrology is an inherently complex problem

What we have shown so far is just the surface — dimension safety with unit conversions. This is what `std::chrono::duration` already provides for the time dimension.

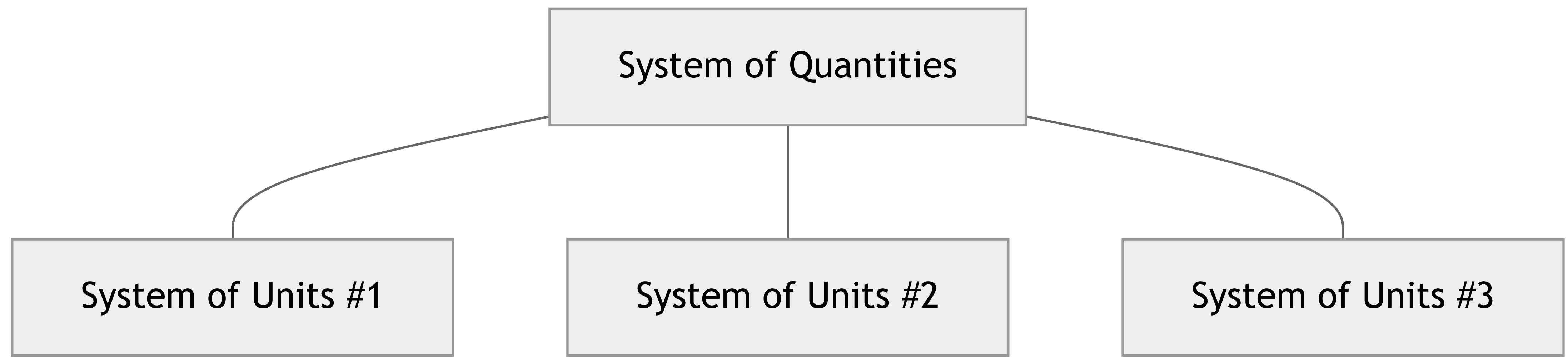
Metrology is an inherently complex problem

What we have shown so far is just the surface — dimension safety with unit conversions. This is what `std::chrono::duration` already provides for the time dimension.

- The real complexity lies **deeper** in the domain model
- We will introduce key concepts **where they are needed**
  - directly before each safety level that relies on them

QUICK DOMAIN OVERVIEW

Quick domain introduction



Quick domain introduction

SYSTEM OF QUANTITIES

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

INTERNATIONAL SYSTEM OF QUANTITIES (ISQ)

- Provided by ISO (ISO 80000)
- System of quantities *based on the seven base quantities*

Quick domain introduction

SYSTEM OF QUANTITIES

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

SYSTEM OF UNITS

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

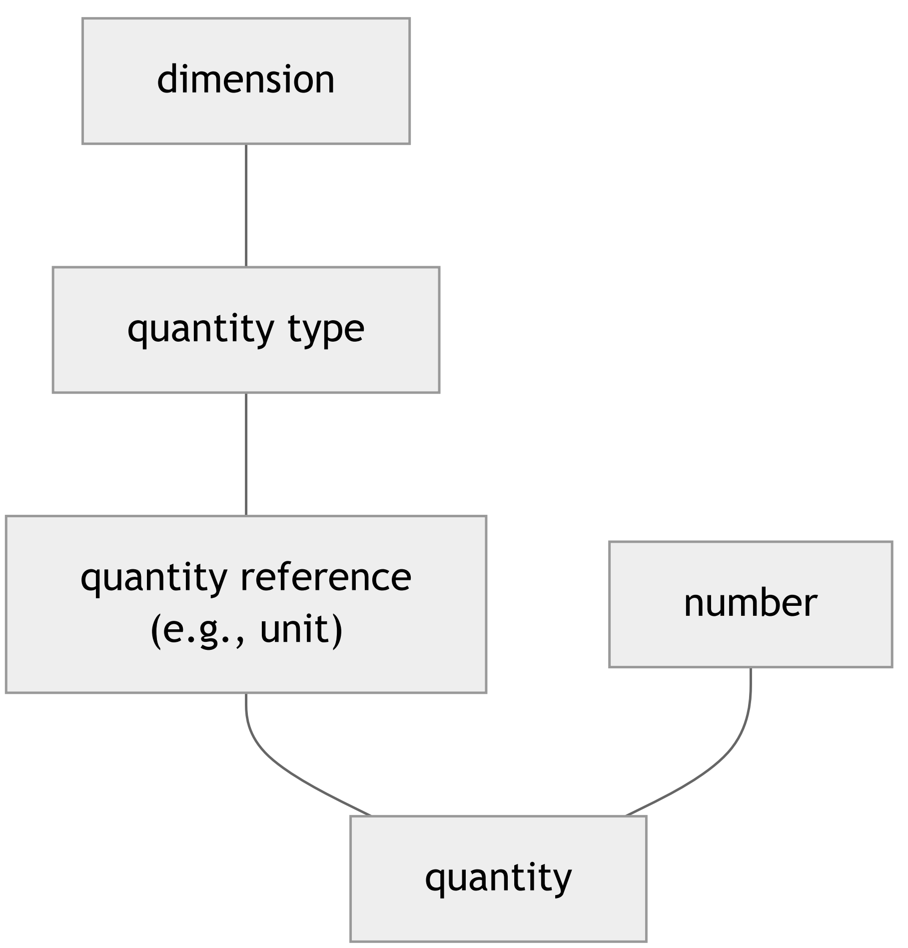
INTERNATIONAL SYSTEM OF QUANTITIES (ISQ)

- Provided by ISO (ISO 80000)
- System of quantities *based on the seven base quantities*

INTERNATIONAL SYSTEM OF UNITS (SI)

- Provided by CGPM
- *Based on the ISQ*

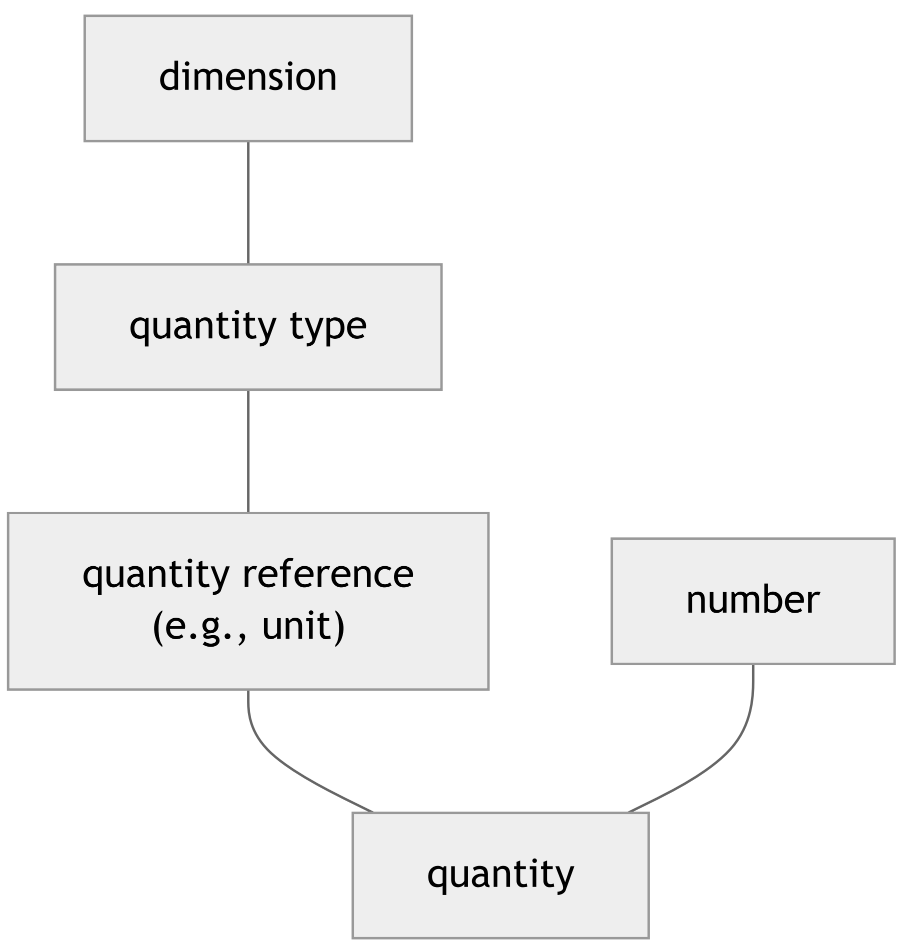
Quick domain introduction



DIMENSIONS

- Base dimensions in the ISQ:
  - *length* (L),
  - *mass* (M),
  - *time* (T),
  - *electric current* (I),
  - *thermodynamic temperature* (Θ),
  - *amount of substance* (N),
  - *luminous intensity* (J)
- Derived dimensions:
  - *force* is LMT^{-2}

Quick domain introduction

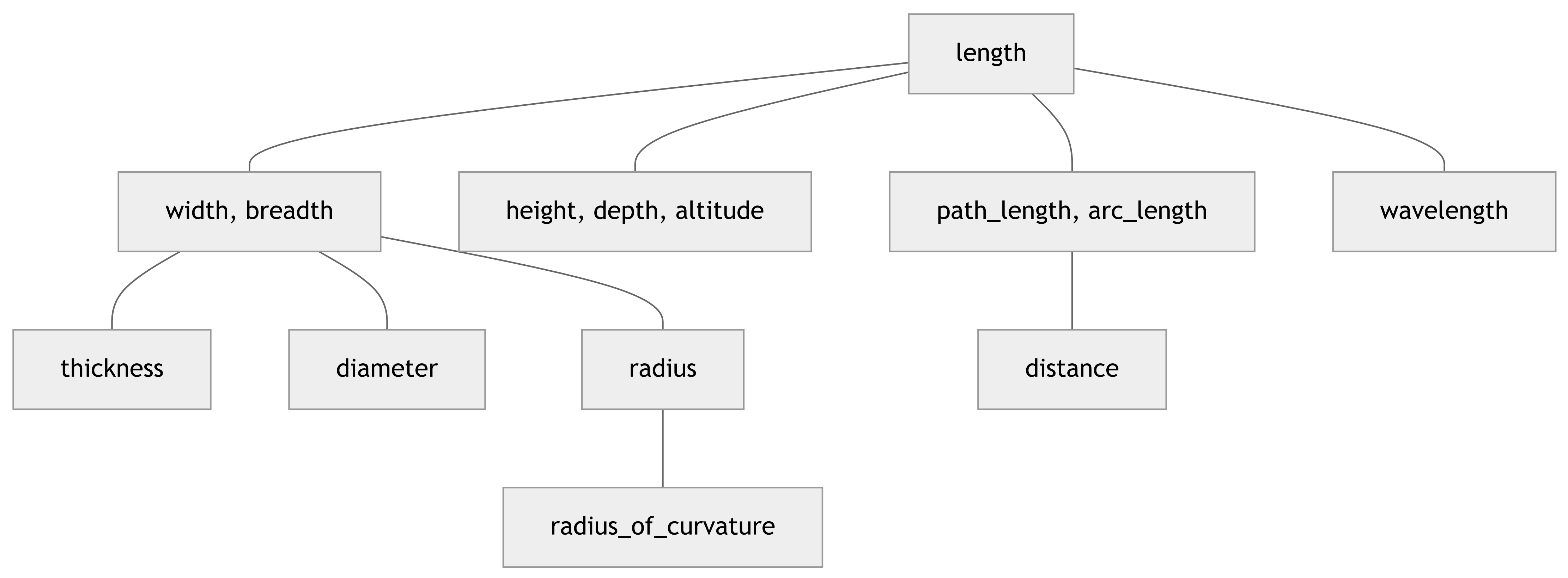


QUANTITY TYPE

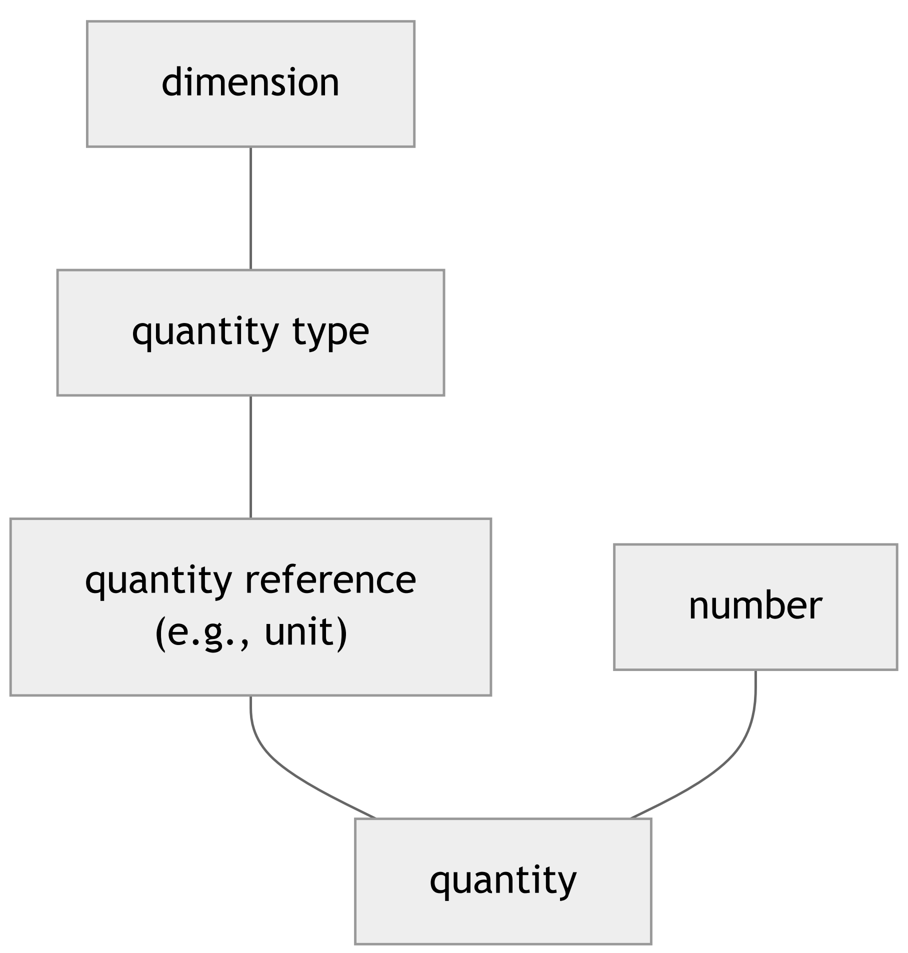
- Dimension is **not enough** to describe a quantity
- Same dimension, different kinds:
  - *work* vs. *torque*
  - *absorbed dose* vs. *dose equivalent*
  - *plane angle* vs. *solid angle*
  - *frequency* vs. *activity*
  - *fluid head* vs. *water head*
- Same dimension and kind but still distinct:
  - *radius* vs. *width* vs. *height*
  - *potential energy* vs. *kinetic energy*

Quick domain introduction

QUANTITIES FORM HIERARCHY TREES



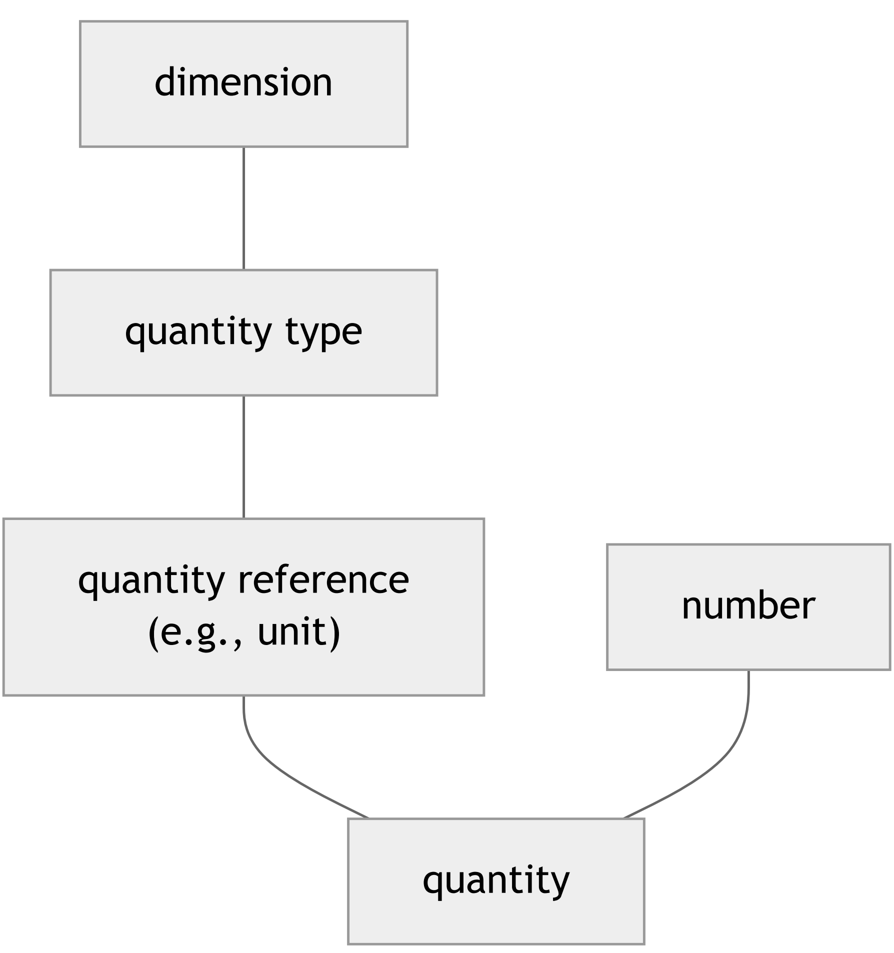
Quick domain introduction



UNIT

- Designated by conventionally assigned *name and symbol*
- J/K is a unit of both *heat capacity* and *entropy*
- $1/s$ is called *hertz* (Hz) when used for *frequencies* and *becquerel* (Bq) when used for *activities*

Quick domain introduction



QUANTITY

- Number and reference (unit)
- Scalars, vectors, and tensors

SIX LEVELS OF SAFETY

Six levels of safety

1. **Dimension Safety** — prevents mixing incompatible dimensions
2. **Unit Safety** — automatic conversions, safe value extraction
3. **Representation Safety** — protects against truncation and precision loss
4. **Quantity Kind Safety** — Sv $\neq$ Gy, rad $\neq$ sr
5. **Quantity Safety** — enforces quantity hierarchies (height $\neq$ width)
6. **Affine Space Safety** — distinguishes points from displacement vectors

Six levels of safety

1. **Dimension Safety** — prevents mixing incompatible dimensions
2. **Unit Safety** — automatic conversions, safe value extraction
3. **Representation Safety** — protects against truncation and precision loss
4. **Quantity Kind Safety** — Sv ≠ Gy, rad ≠ sr
5. **Quantity Safety** — enforces quantity hierarchies (height ≠ width)
6. **Affine Space Safety** — distinguishes points from displacement vectors

Each level adds another layer of compile-time protection, encoding domain knowledge directly into the type system — all with **zero runtime cost**.

LEVEL 1: DIMENSION SAFETY

Level 1: Dimension safety

Prevents mixing quantities of incompatible dimensions.

```
using namespace std::si::unit_symbols;

quantity speed = 100 * km / h;
quantity time = 2 * h;
// quantity<si::kilo<si::metre>> distance = 2 * h;          // ❌ Compile-time error!
// quantity<si::kilo<si::metre>> distance = speed / time;   // ❌ Compile-time error!
quantity<si::kilo<si::metre>> distance = speed * time;      // ✅ OK

// quantity result = distance + time;                       // ❌ Compile-time error!
// Error: cannot add length and time (incompatible dimensions)
```

LEVEL 2: UNIT SAFETY

Who sees the issue?

```
double avg_speed(double distance_m, double duration_s)
{
  return distance_m / duration_s;
}
```

```
double distance_m = 220'000.0;
double distance_km = distance_m / 1000.0;
double duration_h = 2.0;
double speed_mps = avg_speed(distance_km, duration_h);
double speed_kmph = speed_mps / 3.6;
```

Who sees the issue?

```cpp
double avg_speed(double distance_m, double duration_s)
{
  return distance_m / duration_s;
}
```

```cpp
double distance_m = 220'000.0;
double distance_km = distance_m / 1000.0;
double duration_h = 2.0;
double speed_mps = avg_speed(distance_km, duration_h);
double speed_kmph = speed_mps / 3.6;
```

Do you see both issues?

Often hard to spot in code reviews

```cpp
double avg_speed(double distance_m, double duration_s)
{
  return distance_m / duration_s;
}
```

```cpp
double distance_m = 220'000.0;                         // 220 km in meters
double distance_km = distance_m / 1000.0;              // converted to km (220.0)
double duration_h = 2.0;                               // 2 hours
double speed_mps = avg_speed(distance_km, duration_h); // Bug! km and h passed, m and s expected
double speed_kmph = speed_mps / 3.6;                   // Bug! should be × 3.6 (m/s → km/h)
```

Level 2: Safe unit conversions

```
quantity distance = 42 * km;
quantity d_in_m = distance.in(m);     // ✅ 42000 m
assert(distance == d_in_m);           // ✅ Same quantity, different units
```

Level 2: Safe unit conversions

```
quantity distance = 42 * km;
quantity d_in_m = distance.in(m);     // ✅ 42000 m
assert(distance == d_in_m);           // ✅ Same quantity, different units
```

```
quantity speed = 120. * km / h;
quantity s_mps = speed.in(m / s);     // ✅ 33.333... m/s — automatic conversion
```

How we define units?

BOOST.UNITS: MULTI-STEP CRTP + SPECIALIZATIONS

```cpp
// Step 1: Define base dimension tag (CRTP + friend functions for registration)
struct length_base_dimension :
  base_dimension<length_base_dimension, 1> {};

// Step 2: Define dimension typedef
typedef length_base_dimension::dimension_type length_dimension;

// Step 3: Define base unit
struct meter_base_unit : base_unit<meter_base_unit, length_dimension, 1> {};

// Step 4: Define unit typedef
typedef meter_base_unit::unit_type meter;

// Step 5: Add unit symbol BOOST_UNITS_STATIC_CONSTANT
BOOST_UNITS_STATIC_CONSTANT(meters, meter);
```

How we define units?

NHOLTHAUS/UNITS: MACRO-DRIVEN GENERATION

```
// Generates: singular typedef, plural typedef, abbreviation typedef
UNIT_ADD_UNIT_TAGS(length, meter, meters, m,
                   unit<std::ratio<1>, units::category::length_unit>)

// Generates: meter_t container type
UNIT_ADD_UNIT_DEFINITION(length, meter)

// Generates: ostream operator<< that prints value + "m"
UNIT_ADD_IO(length, meter, m)

// Requires 3 macro invocations per unit
```

Different macros for tags, container types, and I/O — easy to forget one or get inconsistent.

A key design principle

In mp-units, **a single line of code** defines a complete entity (dimension, unit, quantity_spec, origin) — with **no type traits, no multiple definitions, no macros**.

A key design principle

MP-UNITS: SINGLE-LINE COMPLETE DEFINITION

```cpp
// Everything in one line: symbol, magnitude, kind, and usage
inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;

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

inline constexpr struct hertz : named_unit<"Hz", one / second, kind_of<isq::frequency>> {} hertz;

inline constexpr struct watt : named_unit<"W", joule / second, kind_of<isq::power>> {} watt;
```

A key design principle

MP-UNITS: SINGLE-LINE COMPLETE DEFINITION

```
// Everything in one line: symbol, magnitude, kind, and usage
inline constexpr struct metre : named_unit<"m", kind_of<isq::length>> {} metre;

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

inline constexpr struct hertz : named_unit<"Hz", one / second, kind_of<isq::frequency>> {} hertz;

inline constexpr struct watt : named_unit<"W", joule / second, kind_of<isq::power>> {} watt;
```

One line. Complete definition. Symbol, text representation, quantity kind, type identity, and runtime operations — all integrated. No hidden steps. No macros. No separate registrations.

What is under the hood?

How are scaled units like `km` defined? What about imperial units or irrational magnitudes?

What is under the hood?

How are scaled units like **km** defined? What about imperial units or irrational magnitudes?

MAGNITUDE HELPERS

```
template<auto V>
constexpr UnitMagnitude auto mag;                  // mag<10> – integer magnitude

template<std::intmax_t N, std::intmax_t D>
constexpr UnitMagnitude auto mag_ratio;            // mag_ratio<1, 1000> – rational (1/1000)

template<auto Base, int Num, int Den = 1>
constexpr UnitMagnitude auto mag_power;            // mag_power<10, 3> – 10³

template<symbol_text Symbol, auto Value>
struct mag_constant;                               // symbolic constants like π
```

What is under the hood?

DEFINING KILOMETRE WITH PREFIX

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

template<PrefixableUnit U>
struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};

template<PrefixableUnit auto U>
constexpr kilo_<decltype(U)> kilo;

inline constexpr auto kilometre = kilo<metre>;  // 10³ m
```

What is under the hood?

DEFINING KILOMETRE WITH PREFIX

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

template<PrefixableUnit U>
struct kilo_ : prefixed_unit<"k", mag_power<10, 3>, U{}> {};

template<PrefixableUnit auto U>
constexpr kilo_<decltype(U)> kilo;

inline constexpr auto kilometre = kilo<metre>;  // 10³ m
```

The **`mag_power<10, 3>`** carries the conversion factor (1000) at **compile time** — enabling exact compile-time arithmetic and zero-cost conversions.

What is under the hood?

DEFINING YARD/POUND UNITS WITH RATIONAL MAGNITUDES

```
// Yard: 0.9144 m = 9144/10000 m (exact rational value)
inline constexpr struct yard :
  named_unit<"yd", mag_ratio<9'144, 10'000> * si::metre> {} yard;

// Pound: 0.45359237 kg (exact rational value)
inline constexpr struct pound :
  named_unit<"lb", mag_ratio<45'359'237, 100'000'000> * si::kilogram> {} pound;

// Foot: 1/3 of a yard (exact)
inline constexpr struct foot :
  named_unit<"ft", mag_ratio<1, 3> * yard> {} foot;
```

What is under the hood?

DEFINING YARD/POUND UNITS WITH RATIONAL MAGNITUDES

```
// Yard: 0.9144 m = 9144/10000 m (exact rational value)
inline constexpr struct yard :
  named_unit<"yd", mag_ratio<9'144, 10'000> * si::metre> {} yard;

// Pound: 0.45359237 kg (exact rational value)
inline constexpr struct pound :
  named_unit<"lb", mag_ratio<45'359'237, 100'000'000> * si::kilogram> {} pound;

// Foot: 1/3 of a yard (exact)
inline constexpr struct foot :
  named_unit<"ft", mag_ratio<1, 3> * yard> {} foot;
```

`mag_ratio` uses exact rational arithmetic — **no floating-point rounding errors**.

What is under the hood?

IRRATIONAL MAGNITUDES WITH π

```
// Degree: 1° = π/180 rad (exact symbolic representation)
inline constexpr struct degree :
  named_unit<"°", mag_ratio<1, 180> * π * si::radian> {} degree;

// Magnetic constant: μ₀ = 4π × 10⁻⁷ H/m
inline constexpr struct magnetic_constant :
  named_constant<"μ₀", mag<4> * mag_power<10, -7> * π * henry / metre> {}
  magnetic_constant;
```

What is under the hood?

IRRATIONAL MAGNITUDES WITH π

```
// Degree: 1° = π/180 rad (exact symbolic representation)
inline constexpr struct degree :
  named_unit<"°", mag_ratio<1, 180> * π * si::radian> {} degree;

// Magnetic constant: μ₀ = 4π × 10⁻⁷ H/m
inline constexpr struct magnetic_constant :
  named_constant<"μ₀", mag<4> * mag_power<10, -7> * π * henry / metre> {}
  magnetic_constant;
```

π is represented **symbolically** in the type system, enabling exact angle conversions without floating-point approximation errors.

What is under the hood?

BINARY PREFIXES: POWERS OF 2 (IEC 80000-13)

`std::ratio` requires unwieldy large integers for binary prefixes

```
std::ratio<1024>                        // kibi = 2¹⁰
std::ratio<1048576>                     // mebi = 2²⁰
std::ratio<1073741824>                  // gibi = 2³⁰
std::ratio<1099511627776>               // tebi = 2⁴⁰
std::ratio<1125899906842624>            // pebi = 2⁵⁰
std::ratio<1152921504606846976>         // exbi = 2⁶⁰
// std::ratio<...>                      // zebi = 2⁷⁰ – OVERFLOW on 64-bit std::intmax_t!
// std::ratio<...>                      // yobi = 2⁸⁰ – OVERFLOW on 64-bit std::intmax_t!
```

What is under the hood?

BINARY PREFIXES: POWERS OF 2 (IEC 80000-13)

`mag_power` makes binary prefixes clear and correct by construction

```
template<PrefixableUnit U> struct kibi_ : prefixed_unit<"Ki", mag_power<2, 10>, U{}> {};
template<PrefixableUnit U> struct mebi_ : prefixed_unit<"Mi", mag_power<2, 20>, U{}> {};
// ...up to yobi (2⁸⁰)

inline constexpr struct bit : named_unit<"bit", one> {} bit;
inline constexpr auto kibibit = kibi<bit>;  // 2¹⁰ bit = 1024 bit
```

What is under the hood?

BINARY PREFIXES: POWERS OF 2 (IEC 80000-13)

`mag_power` makes binary prefixes clear and correct by construction

```
template<PrefixableUnit U> struct kibi_ : prefixed_unit<"Ki", mag_power<2, 10>, U{}> {};
template<PrefixableUnit U> struct mebi_ : prefixed_unit<"Mi", mag_power<2, 20>, U{}> {};
// ...up to yobi (2⁸⁰)

inline constexpr struct bit : named_unit<"bit", one> {} bit;
inline constexpr auto kibibit = kibi<bit>;  // 2¹⁰ bit = 1024 bit
```

The same `mag_power` mechanism works for **any base** — decimal (10^n), binary (2^n), or custom.

Level 2: Safe numerical value extraction

```
double val1 = distance.numerical_value_in(m);    // ✅ explicit unit required
double val2 = distance.numerical_value_in(mm);   // ✅ different unit, auto-converted
double val3 = distance.numerical_value_in(s);    // ❌ Compile error
```

Level 2: Safe numerical value extraction

```
double val1 = distance.numerical_value_in(m);     // ✅ explicit unit required
double val2 = distance.numerical_value_in(mm);    // ✅ different unit, auto-converted
double val3 = distance.numerical_value_in(s);     // ❌ Compile error
```

COMPARISON WITH `std::chrono`

`std::chrono::duration`

```
auto d = 42s;
d.count();  // 42 – but 42 what?
```

`std::quantity`

```
quantity d = 42 * s;
d.numerical_value_in(s);    // explicit
d.numerical_value_in(ms);   // 42000
```

Level 2: Safe numerical value extraction

```
double val1 = distance.numerical_value_in(m);    // ✅ explicit unit required
double val2 = distance.numerical_value_in(mm);   // ✅ different unit, auto-converted
double val3 = distance.numerical_value_in(s);    // ❌ Compile error
```

COMPARISON WITH **std::chrono**

std::chrono::duration

```
auto d = 42s;
d.count();  // 42 — but 42 what?
```

std::quantity

```
quantity d = 42 * s;
d.numerical_value_in(s);   // explicit
d.numerical_value_in(ms);  // 42000
```

numerical_value_in() always requires a unit argument, making the extraction unambiguous. This is safer than **count()**.

Try it yourself! 🧪

Refactor the legacy function, and arguments passed to it from main() to use strongly typed quantities (e.g., quantity of mass in kilograms, quantity of speed in metre per second). Print the result in J and kJ.

LEVEL 3: REPRESENTATION SAFETY

Who sees the issue?

```
int total_mm = 5;
int total_m = total_mm / 1000;
```

A real mistake pattern

```
int total_mm = 5;
int total_m = total_mm / 1000;        // 0 — silently wrong!
```

A real mistake pattern

```cpp
int total_mm = 5;
int total_m = total_mm / 1000;        // 0 — silently wrong!
```

```cpp
quantity total = 5 * mm;
// quantity<m, int> bad = total;      // ❌ Compile error
quantity<m, double> safe = total;     // ✅ 0.005 m
```

Level 3: Preventing silent truncation

The library **follows `std::chrono::duration` logic**

```
quantity distance = 1500 * m;
quantity d_km = distance.in(km);      // ❌ Compile error: int truncation
quantity<km> d2 = distance;           // ❌ Compile error: same reason
```

Level 3: Preventing silent truncation

The library **follows `std::chrono::duration` logic**

```
quantity distance = 1500 * m;
quantity d_km = distance.in(km);      // ❌ Compile error: int truncation
quantity<km> d2 = distance;           // ❌ Compile error: same reason
```

```
// Floating-point is considered value-preserving:
quantity distance = 1500. * m;
quantity d_km = distance.in(km);      // ✅ 1.5 km
```

Level 3: Forcing truncation

When you know what you're doing

```
quantity distance = 1500 * m;
quantity d_km = distance.force_in(km);                    // ✅ 1 km (truncated)
auto km_count = distance.force_numerical_value_in(km);   // ✅ 1 (truncated)
quantity d_km2 = distance.in<double>(km);                 // ✅ 1.5 km
```

Level 3: Forcing truncation

When you know what you're doing

```
quantity distance = 1500 * m;
quantity d_km = distance.force_in(km);                   // ✅ 1 km (truncated)
auto km_count = distance.force_numerical_value_in(km);   // ✅ 1 (truncated)
quantity d_km2 = distance.in<double>(km);                // ✅ 1.5 km
```

Also handles representation type truncation

```
quantity q = 2.5 * m;
quantity<m, int> q2 = q;                 // ❌ Compile error: double → int
quantity<m, int> q3 = q.force_in<int>(); // ✅ 2 m (explicit truncation)
```

`explicit` is not explicit enough

TODAY

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

TOMORROW

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

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

`explicit` is not explicit enough

TODAY

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

TOMORROW

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

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

The code compiles in both cases, but calculations are now off by **1000×**.

`explicit` is not explicit enough

TODAY

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

TOMORROW

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

```
X x;
x.vec.emplace_back(42);         // ❌ Compile error – always requires a unit
x.vec.emplace_back(42 * ms);    // ✅ OK
x.vec.emplace_back(42, ms);     // ✅ OK
```

`explicit` is not explicit enough

TODAY

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

TOMORROW

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

```cpp
X x;
x.vec.emplace_back(42);          // ❌ Compile error – always requires a unit
x.vec.emplace_back(42 * ms);     // ✅ OK
x.vec.emplace_back(42, ms);      // ✅ OK
```

Quantity construction **always requires both a number and a unit**.

LEVEL 4: QUANTITY KIND SAFETY

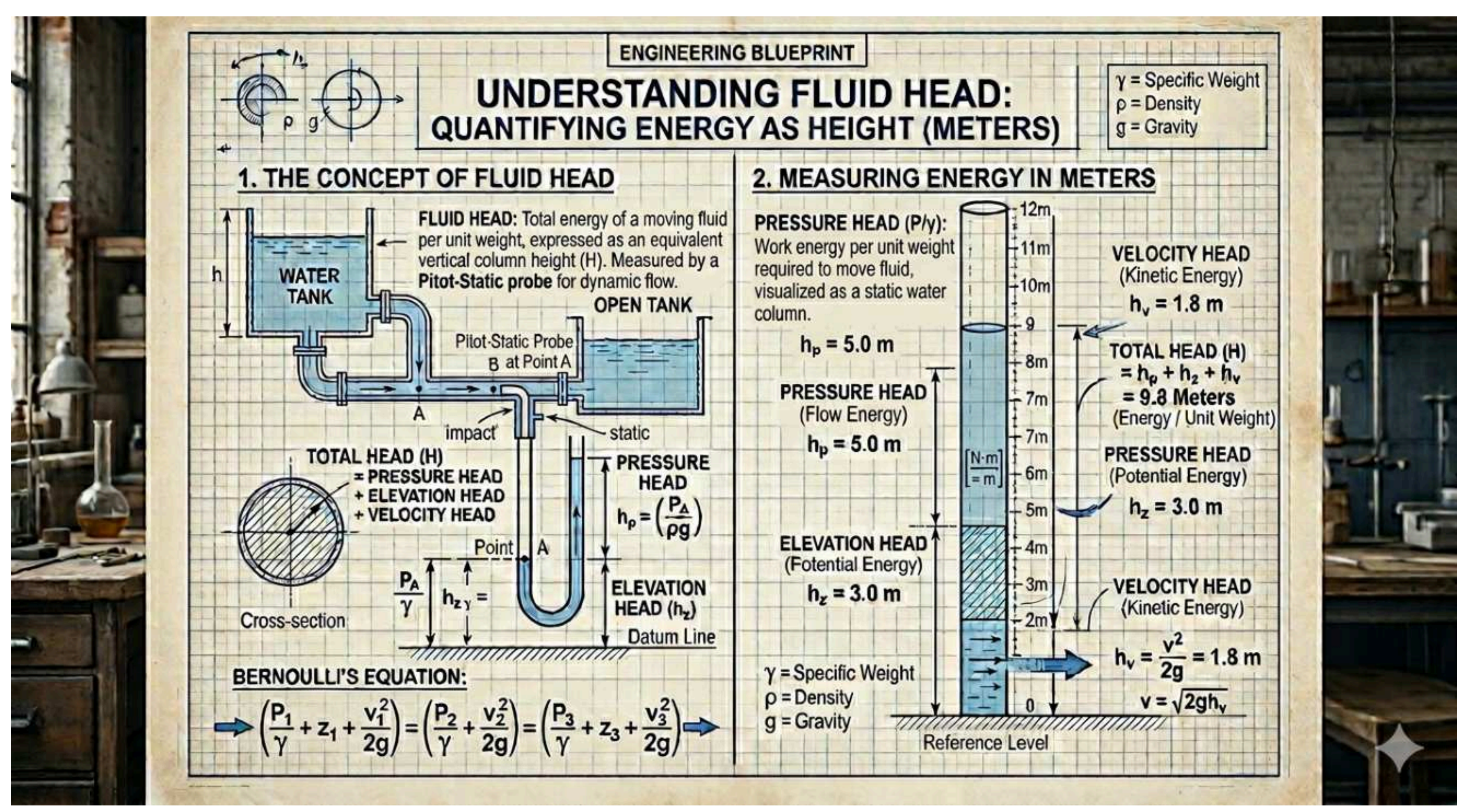
A real user problem #718

oZed opened on Sep 2, 2025

I am trying to use mp-units for some hydraulic calculations and so wanted to implement the hydraulic head as a new quantity. It is somewhat common to calculate the head in terms of either the fluid in question (sometimes called the 'fluid' head) or normalised against the specific gravity to give the head in terms of water (sometimes called the 'water' head). I want to make these two quantity types distinct to prevent them being mixed up, however since the fluid head really acts like a tuple of quantities (a head and a specific gravity value) it isn't really modelled by the ISQ character types. I thought about trying to use a quantity_point with the specific gravity being somewhat like an origin, but currently the library requires the origin and delta to have the same quantity type. Am I better off just implementing a custom type in such a case? Please let me know if I've missed something and this can be implemented with the current library types!

Create sub-issue

Fluid Head



Water head vs. fluid head

In hydraulic engineering, head can be expressed in two distinct ways — both with dimension of **length**:

- **Fluid head**: energy normalized to the *actual fluid's* density
- **Water head**: energy normalized to *water's* density

Water head vs. fluid head

In hydraulic engineering, head can be expressed in two distinct ways — both with dimension of **length**:

- **Fluid head**: energy normalized to the *actual fluid's* density
- **Water head**: energy normalized to *water's* density

```
// Both are just 'length' in Boost.Units — it cannot distinguish fluid from water head:
bu::quantity<bu::si::length> fluid_head = 2.0 * bu::si::meters;
bu::quantity<bu::si::length> water_head = 4.0 * bu::si::meters;

auto wrong_sum = fluid_head + water_head;  // Oops! Compiles! 6 m, but physically meaningless
if (fluid_head < water_head) { /* ... */ } // Oops! Compares magnitudes, not water equivalents

// No way to enforce that conversion via specific gravity must happen first:
use_water_head(fluid_head);                // Oops! Passes fluid head where water head expected
```

Water head vs. fluid head

In hydraulic engineering, head can be expressed in two distinct ways — both with dimension of **length**:

- **Fluid head**: energy normalized to the *actual fluid's* density
- **Water head**: energy normalized to *water's* density

```
// Both are just 'length' in Boost.Units – it cannot distinguish fluid from water head:
bu::quantity<bu::si::length> fluid_head = 2.0 * bu::si::meters;
bu::quantity<bu::si::length> water_head = 4.0 * bu::si::meters;

auto wrong_sum = fluid_head + water_head;  // Oops! Compiles! 6 m, but physically meaningless
if (fluid_head < water_head) { /* ... */ } // Oops! Compares magnitudes, not water equivalents

// No way to enforce that conversion via specific gravity must happen first:
use_water_head(fluid_head);                // Oops! Passes fluid head where water head expected
```

2 m of mercury ≈ **27.2 m** of water head (SG = 13.6) — **not** 2 m!

Distinguishing doses: Gy vs Sv

- **Absorbed dose** (Gy = J/kg) measures raw energy deposited in tissue
- **Dose equivalent** (Sv = J/kg) weights that energy by **biological effectiveness**
- 1 Gy of neutrons ≈ **20 Sv** of dose equivalent
- Treating them as interchangeable has **life-safety consequences**

Distinguishing doses: Gy vs Sv

- **Absorbed dose** (Gy = J/kg) measures raw energy deposited in tissue
- **Dose equivalent** (Sv = J/kg) weights that energy by **biological effectiveness**
- 1 Gy of neutrons ≈ **20 Sv** of dose equivalent
- Treating them as interchangeable has **life-safety consequences**

```
// Boost.Units — silently mixes them up:
bu::quantity<bu::si::absorbed_dose> absorbed = 2.5 * bu::si::gray;
bu::quantity<bu::si::dose_equivalent> equivalent = absorbed;     // Oops! Compiles!
auto equal = (absorbed == 2.5 * bu::si::sievert);                // Oops! Compiles!
```

Distinguishing doses: Gy vs Sv

- **Absorbed dose** (Gy = J/kg) measures raw energy deposited in tissue
- **Dose equivalent** (Sv = J/kg) weights that energy by **biological effectiveness**
- 1 Gy of neutrons ≈ **20 Sv** of dose equivalent
- Treating them as interchangeable has **life-safety consequences**

```
// Boost.Units — silently mixes them up:
bu::quantity<bu::si::absorbed_dose> absorbed = 2.5 * bu::si::gray;
bu::quantity<bu::si::dose_equivalent> equivalent = absorbed;     // Oops! Compiles!
auto equal = (absorbed == 2.5 * bu::si::sievert);                // Oops! Compiles!
```

The risk is so serious that the Au library omitted Sievert entirely rather than risk users mixing them up.

Same dimension, different meaning

QUANTITY	DIMENSION	UNIT	SI SYMBOL
frequency	T^{-1}	hertz	Hz
activity	T^{-1}	becquerel	Bq
modulation rate	T^{-1}	baud	Bd

Same dimension, different meaning

QUANTITY	DIMENSION	UNIT	SI SYMBOL
frequency	T^{-1}	hertz	Hz
activity	T^{-1}	becquerel	Bq
modulation rate	T^{-1}	baud	Bd

All three have the **same dimension** but are **completely different physical concepts**.

Same dimension, different meaning

QUANTITY	DIMENSION	UNIT	SI SYMBOL
frequency	T^{-1}	hertz	Hz
activity	T^{-1}	becquerel	Bq
modulation rate	T^{-1}	baud	Bd

All three have the **same dimension** but are **completely different physical concepts**.

Dimension is not enough to describe a quantity! Most libraries get this wrong.

What should be the result?

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

What should be the result?

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

OTHER LIBRARIES

LIBRARY	`1 * Hz + 1 * Bq`	`1 * Bq + 1 * Hz`
Boost.Units	2 Hz ❌	2 Hz ❌
nholthaus	2 s^{-1} ❌	2 s^{-1} ❌
Pint (Python)	2 Hz ❌	2 Bq ❌
JSR 385 (Java)	Compile error ✅	Compile error ✅
mp-units	Compile error ✅	Compile error ✅

ISQ quantities of dimension T^{-1}

angular_velocity
(angular_displacement / duration)
[rad/s, 1/s]
{vector}

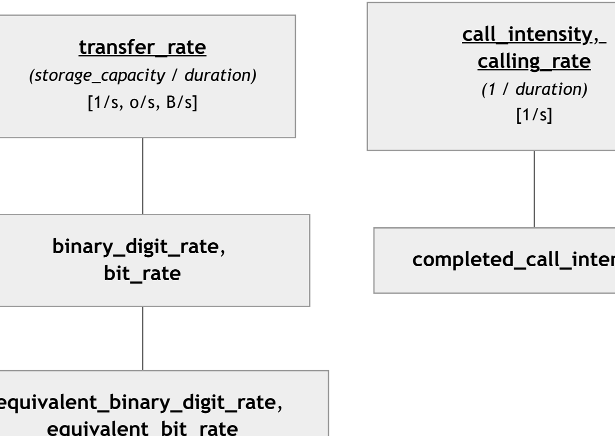
frequency
(1 / period_duration)
[Hz, 1/s]

rotational_frequency
(rotation / duration)
[1/s]

angular_frequency
(phase_angle / duration)
[rad/s, 1/s]

damping_coefficient
(1 / time_constant)
[1/s]

activity
(neutron_number / time)
[Bq, 1/s]



modulation_rate, line_digit_rate
(1 / duration)
[Bd, 1/s]

Concept: quantity kinds (ISO 80000)

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

Concept: quantity kinds (ISO 80000)

- Quantities may be grouped into **categories of mutually comparable quantities**
- 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 kind

Concept: quantity kinds (ISO 80000)

- Quantities may be grouped into **categories of mutually comparable quantities**
- 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 kind
- Quantities of the same kind **have the same dimension**
- Quantities of the same dimension are **not necessarily of the same kind**

kind_of<QS> modifier

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

```
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*

```
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*

```
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 encode quantity kind

```
namespace std::si {

inline constexpr struct second final :    named_unit<"s", kind_of<isq::time>> {} second;
inline constexpr struct metre final :     named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct hertz final :     named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel final : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton final :    named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct joule final :     named_unit<"J", newton * metre> {} joule;

}
```

SI units encode quantity kind

```
namespace std::si {

inline constexpr struct second final :    named_unit<"s", kind_of<isq::time>> {} second;
inline constexpr struct metre final :     named_unit<"m", kind_of<isq::length>> {} metre;
inline constexpr struct hertz final :     named_unit<"Hz", inverse(second), kind_of<isq::frequency>> {} hertz;
inline constexpr struct becquerel final : named_unit<"Bq", inverse(second), kind_of<isq::activity>> {} becquerel;
inline constexpr struct newton final :    named_unit<"N", kilogram * metre / square(second)> {} newton;
inline constexpr struct joule final :     named_unit<"J", newton * metre> {} joule;

}
```

`hertz` and **`becquerel`** have the same definition (**`1/s`**) but carry different quantity kind information. This is how **`Hz + Bq`** is rejected at compile time.

Water head vs. fluid head

```
// Both modelled as sub-kinds of isq::height — conversion requires SG
inline constexpr struct fluid_head final : quantity_spec<isq::height, is_kind> {} fluid_head;
inline constexpr struct water_head final : quantity_spec<isq::height, is_kind> {} water_head;
inline constexpr struct specific_gravity final : quantity_spec<dimensionless> {} specific_gravity;
```

Water head vs. fluid head

```
// Both modelled as sub-kinds of isq::height — conversion requires SG
inline constexpr struct fluid_head final : quantity_spec<isq::height, is_kind> {} fluid_head;
inline constexpr struct water_head final : quantity_spec<isq::height, is_kind> {} water_head;
inline constexpr struct specific_gravity final : quantity_spec<dimensionless> {} specific_gravity;
```

```
constexpr quantity_of<water_head> auto to_water_head(quantity_of<fluid_head> auto h_fluid,
                                                     quantity_of<specific_gravity> auto sg)
{ return water_head(isq::height(h_fluid) * sg); }
```

Water head vs. fluid head

```
// Both modelled as sub-kinds of isq::height – conversion requires SG
inline constexpr struct fluid_head final : quantity_spec<isq::height, is_kind> {} fluid_head;
inline constexpr struct water_head final : quantity_spec<isq::height, is_kind> {} water_head;
inline constexpr struct specific_gravity final : quantity_spec<dimensionless> {} specific_gravity;
```

```
constexpr quantity_of<water_head> auto to_water_head(quantity_of<fluid_head> auto h_fluid,
                                                      quantity_of<specific_gravity> auto sg)
{ return water_head(isq::height(h_fluid) * sg); }
```

```
quantity fluid = fluid_head(2 * m);
quantity water = water_head(4 * m);
use_water_head(water);            // ✅ OK

// use_water_head(fluid);         // ❌ Compile-time error!
// quantity q = fluid + water;    // ❌ Compile-time error!
// if (fluid == water) {}         // ❌ Compile-time error!

use_height(isq::height(fluid));   // ✅ OK: explicit upcast to generic length
use_height(isq::height(water));   // ✅ OK: explicit upcast to generic length

// Convert mercury fluid head to equivalent water head
quantity sg_mercury = specific_gravity(13.6);
quantity mercury_as_water = to_water_head(fluid, sg_mercury);
```

Strong-type numerical wrappers

```
inline constexpr struct pixel_x final :      quantity_spec<dimensionless, is_kind> {} pixel_x;
inline constexpr struct pixel_y final :      quantity_spec<dimensionless, is_kind> {} pixel_y;
inline constexpr struct sprite_count final : quantity_spec<dimensionless, is_kind> {} sprite_count;
inline constexpr struct frame_count final :  quantity_spec<dimensionless, is_kind> {} frame_count;
inline constexpr struct buffer_index final : quantity_spec<dimensionless, is_kind> {} buffer_index;

inline constexpr struct resolution_width final :  quantity_spec<pixel_x> {} resolution_width;
inline constexpr struct resolution_height final : quantity_spec<pixel_y> {} resolution_height;

inline constexpr struct render_rate final : quantity_spec<sprite_count / isq::time> {} render_rate;
inline constexpr struct frame_rate final :  quantity_spec<frame_count / isq::time> {} frame_rate;

[[nodiscard]] constexpr bool is_within_bounds(quantity_of<pixel_x> auto x,
                                              quantity_of<pixel_y> auto y,
                                              quantity_of<resolution_width> auto width,
                                              quantity_of<resolution_height> auto height);

[[nodiscard]] constexpr quantity_of<render_rate> auto render_rate_calc(quantity_of<sprite_count> auto count,
                                                                       quantity_of<isq::time> auto duration);

[[nodiscard]] constexpr quantity_of<buffer_index> auto grid_index(quantity_of<pixel_x> auto x,
                                                                  quantity_of<pixel_y> auto y,
                                                                  quantity_of<resolution_width> auto width);
```

Quantity kind safety at CERN

```
// HEP: proper_time and coordinate_time are different subkinds
inline constexpr struct proper_time final : quantity_spec<duration, is_kind> {} proper_time;
inline constexpr struct coordinate_time final : quantity_spec<duration, is_kind> {} coordinate_time;
```

Quantity kind safety at CERN

```
// HEP: proper_time and coordinate_time are different subkinds
inline constexpr struct proper_time final : quantity_spec<duration, is_kind> {} proper_time;
inline constexpr struct coordinate_time final : quantity_spec<duration, is_kind> {} coordinate_time;
```

```
quantity tau   = hep::proper_time(0.385 * ns);      // B⁰ rest-frame lifetime
quantity t_lab = hep::coordinate_time(4.2 * ns);    // lab time of flight

// ✅ Division across subkinds → dimensionless Lorentz factor
quantity gamma = hep::lorentz_factor(t_lab / tau);  // γ ≈ 10.9

// ❌ Can't accidentally add proper time and coordinate time!
// auto bad = tau + t_lab;  // Compile error
```

Quantity kind safety at CERN

```
// HEP: proper_time and coordinate_time are different subkinds
inline constexpr struct proper_time final : quantity_spec<duration, is_kind> {} proper_time;
inline constexpr struct coordinate_time final : quantity_spec<duration, is_kind> {} coordinate_time;
```

```
quantity tau   = hep::proper_time(0.385 * ns);      // B⁰ rest-frame lifetime
quantity t_lab = hep::coordinate_time(4.2 * ns);    // lab time of flight

// ✅ Division across subkinds → dimensionless Lorentz factor
quantity gamma = hep::lorentz_factor(t_lab / tau);  // γ ≈ 10.9

// ❌ Can't accidentally add proper time and coordinate time!
// auto bad = tau + t_lab;  // Compile error
```

Subkinds encode **physics intent** directly into the type system.

Try it yourself! 🧪

Implement a type-safe hydraulic head calculation system using mp-units that prevents mixing fluid head and water head without explicit conversion.

Metrology is an inherently complex problem

Metrology is an inherently complex problem

We just saw that **dimensionally identical** quantities (Hz vs Bq vs Bd; Gy vs Sv) can be **physically incompatible**. The type system must go deeper than dimensions alone.

LEVEL 5: QUANTITY SAFETY

The "Argument Soup" Problem

GEANT4: `G4Trap` CONSTRUCTOR

```
G4Trap(const G4String& name,
       G4double pDz, G4double pTheta, G4double pPhi,
       G4double pDy1, G4double pDx1, G4double pDx2, G4double pAlp1,
       G4double pDy2, G4double pDx3, G4double pDx4, G4double pAlp2);
```

The "Argument Soup" Problem

GEANT4: `G4Trap` CONSTRUCTOR

```
G4Trap(const G4String& name,
       G4double pDz, G4double pTheta, G4double pPhi,
       G4double pDy1, G4double pDx1, G4double pDx2, G4double pAlp1,
       G4double pDy2, G4double pDx3, G4double pDx4, G4double pAlp2);
```

- **12 double parameters** - which are lengths? which are angles?
- Accidentally swapping **`pDx2`** and **`pAlp1`** compiles fine

The "Argument Soup" Problem

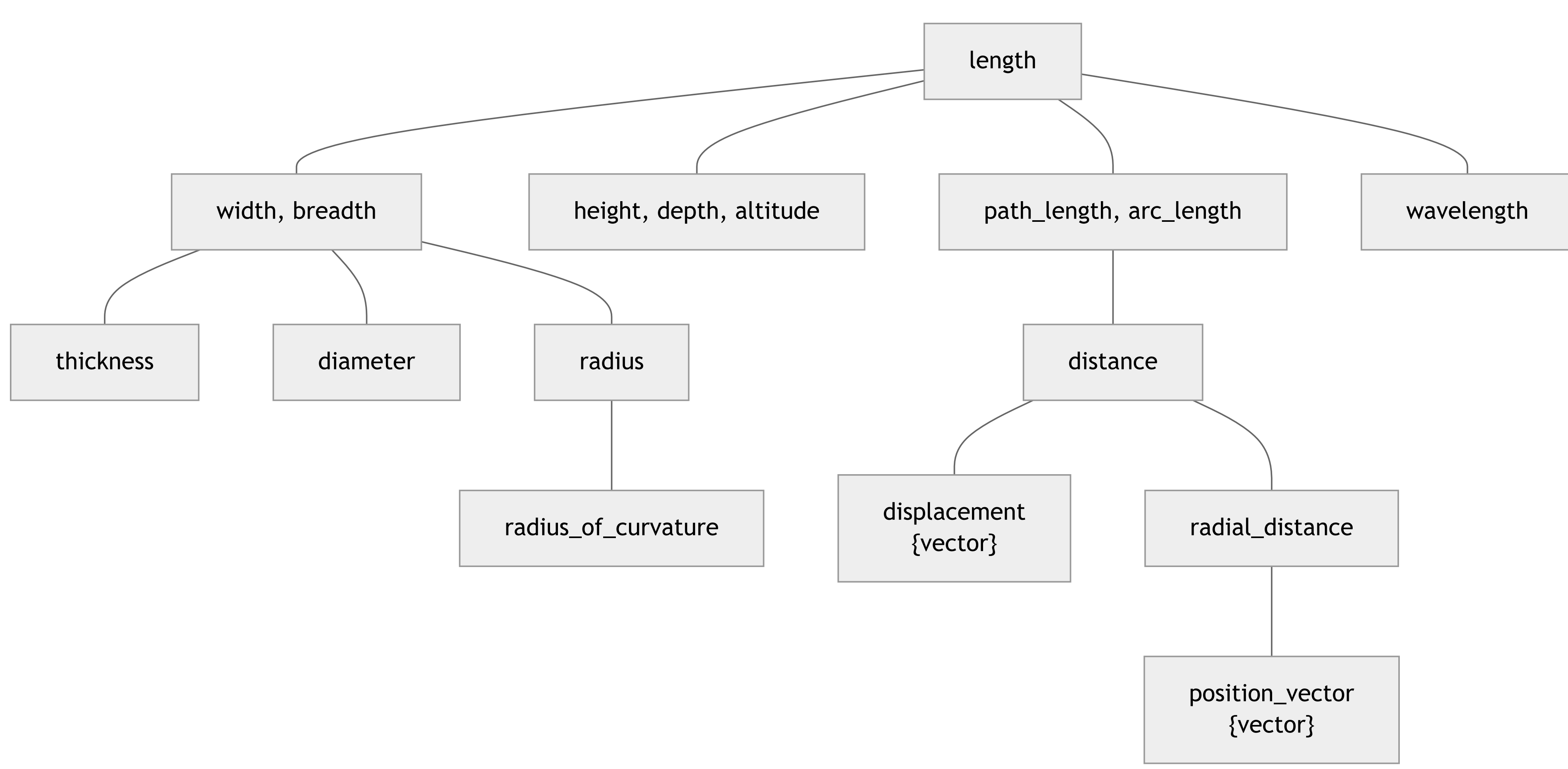
GEANT4: `G4Trap` CONSTRUCTOR

```
G4Trap(const G4String& name,
       G4double pDz, G4double pTheta, G4double pPhi,
       G4double pDy1, G4double pDx1, G4double pDx2, G4double pAlp1,
       G4double pDy2, G4double pDx3, G4double pDx4, G4double pAlp2);
```

- **12 double parameters** - which are lengths? which are angles?
- Accidentally swapping **`pDx2`** and **`pAlp1`** compiles fine

Result: Invalid detector geometry, caught only at runtime (maybe)

Discriminating between quantities of the same kind



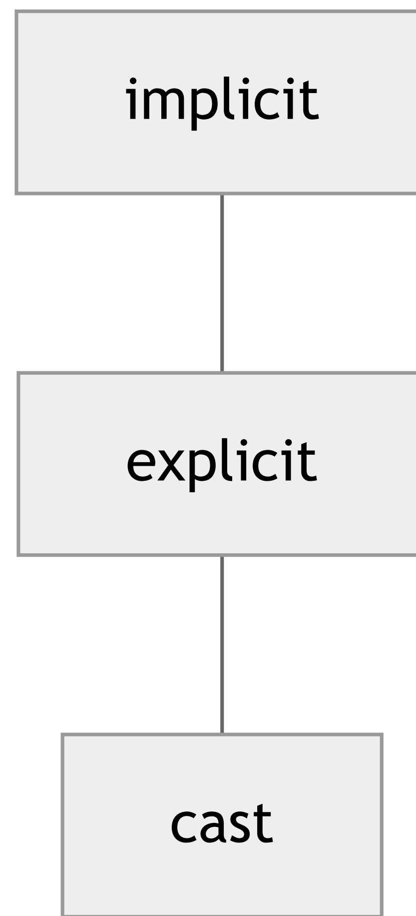
Defining quantities (ISQ model in code)

```
inline constexpr struct length final : quantity_spec<dim_length> {} length;
inline constexpr struct width final : quantity_spec<length> {} width;
inline constexpr auto breadth = width;
```

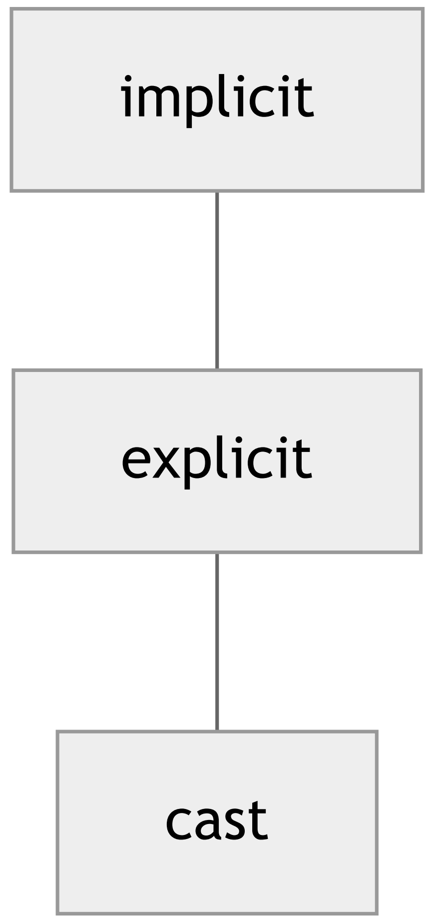
Defining quantities (ISQ model in code)

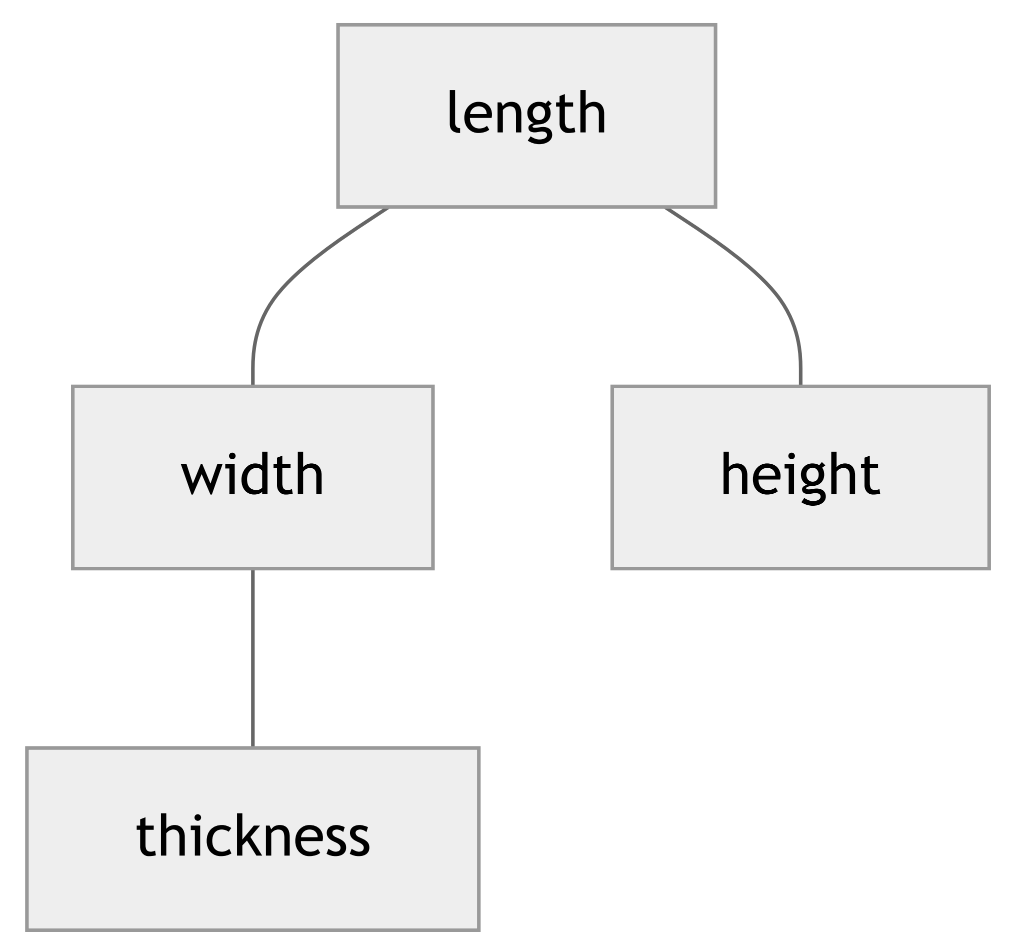
```
inline constexpr struct length final : quantity_spec<dim_length> {} length;
inline constexpr struct width final : quantity_spec<length> {} width;
inline constexpr auto breadth = width;
inline constexpr struct thickness final : quantity_spec<width> {} thickness;
inline constexpr struct diameter final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

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: implicit (child → parent)
static_assert(implicitly_convertible(isq::width, isq::length));

// length → width: explicit (parent → child)
static_assert(!implicitly_convertible(isq::length, isq::width));
static_assert(explicitly_convertible(isq::length, isq::width));

// height → width: cast (sibling → sibling)
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: impossible
static_assert(!castable(isq::time, isq::length));
```

Typed quantities: the Box example

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

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

Typed quantities: the Box example

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

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

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

Typed quantities: the Box example

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

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

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

```
// Arguments swapped: height passed where length expected
Box bad(isq::height(1 * m), horizontal_length(2 * m), isq::width(3 * m));
```

```
error: no known conversion for argument 1 from 'quantity<isq::height[m]>'
       to 'quantity<horizontal_length[m]>'
```

Typed quantities enable safer equations

Some quantities require specific inputs, not just dimensionally correct ones:

```
inline constexpr struct gravitational_potential_energy final :
    quantity_spec<isq::potential_energy, isq::mass * isq::acceleration_of_free_fall * isq::height> {}
    gravitational_potential_energy;
```

Typed quantities enable safer equations

Some quantities require specific inputs, not just dimensionally correct ones:

```
inline constexpr struct gravitational_potential_energy final :
    quantity_spec<isq::potential_energy, isq::mass * isq::acceleration_of_free_fall * isq::height> {}
    gravitational_potential_energy;
```

```
constexpr quantity g0 = isq::acceleration_of_free_fall(1 * si::standard_gravity);
quantity<isq::mass[kg]> mass = 10 * kg;
quantity<isq::height[m]> height = 5 * m;

// Gravitational potential energy requires height specifically
quantity<gravitational_potential_energy[J]> Ep = mass * g0 * height;        // ✅ OK

// Cannot substitute generic length for height
quantity<isq::length[m]> width = 2 * m;
// quantity<gravitational_potential_energy[J]> wrong = mass * g0 * width;  // ❌ Error: generic length
```

Quantity Safety in Action

*Turning the 11-**`double`** soup into a self-documenting API*

```
// Each geometric axis and angle role gets its own sub-kind of hep::length / hep::angle
inline constexpr struct trap_half_z      final : quantity_spec<hep::length> {} trap_half_z;
inline constexpr struct trap_half_y      final : quantity_spec<hep::length> {} trap_half_y;
inline constexpr struct trap_half_x      final : quantity_spec<hep::length> {} trap_half_x;
inline constexpr struct trap_tilt_angle  final : quantity_spec<hep::angle>  {} trap_tilt_angle;
inline constexpr struct trap_shear_angle final : quantity_spec<hep::angle>  {} trap_shear_angle;
```

Quantity Safety in Action

*Turning the 11-**`double`** soup into a self-documenting API*

```
// Each geometric axis and angle role gets its own sub-kind of hep::length / hep::angle
inline constexpr struct trap_half_z      final : quantity_spec<hep::length> {} trap_half_z;
inline constexpr struct trap_half_y      final : quantity_spec<hep::length> {} trap_half_y;
inline constexpr struct trap_half_x      final : quantity_spec<hep::length> {} trap_half_x;
inline constexpr struct trap_tilt_angle  final : quantity_spec<hep::angle>  {} trap_tilt_angle;
inline constexpr struct trap_shear_angle final : quantity_spec<hep::angle>  {} trap_shear_angle;
```

```
void MakeG4Trap(quantity<trap_half_z[mm]>       pDz,
                quantity<trap_tilt_angle[rad]>  pTheta,
                quantity<trap_tilt_angle[rad]>  pPhi,
                quantity<trap_half_y[mm]>       pDy1,
                quantity<trap_half_x[mm]>       pDx1,
                quantity<trap_half_x[mm]>       pDx2,
                quantity<trap_shear_angle[rad]> pAlp1,
                quantity<trap_half_y[mm]>       pDy2,
                quantity<trap_half_x[mm]>       pDx3,
                quantity<trap_half_x[mm]>       pDx4,
                quantity<trap_shear_angle[rad]> pAlp2);
```

Quantity Safety in Action

*Turning the 11-**`double`** soup into a self-documenting API*

```
// Each geometric axis and angle role gets its own sub-kind of hep::length / hep::angle
inline constexpr struct trap_half_z      final : quantity_spec<hep::length> {} trap_half_z;
inline constexpr struct trap_half_y      final : quantity_spec<hep::length> {} trap_half_y;
inline constexpr struct trap_half_x      final : quantity_spec<hep::length> {} trap_half_x;
inline constexpr struct trap_tilt_angle  final : quantity_spec<hep::angle>  {} trap_tilt_angle;
inline constexpr struct trap_shear_angle final : quantity_spec<hep::angle>  {} trap_shear_angle;
```

```
void MakeG4Trap(quantity<trap_half_z[mm]>        pDz,
                quantity<trap_tilt_angle[rad]>  pTheta,
                quantity<trap_tilt_angle[rad]>  pPhi,
                quantity<trap_half_y[mm]>        pDy1,
                quantity<trap_half_x[mm]>        pDx1,
                quantity<trap_half_x[mm]>        pDx2,
                quantity<trap_shear_angle[rad]> pAlp1,
                quantity<trap_half_y[mm]>        pDy2,
                quantity<trap_half_x[mm]>        pDx3,
                quantity<trap_half_x[mm]>        pDx4,
                quantity<trap_shear_angle[rad]> pAlp2);
```

```
quantity pDz   = trap_half_z(150. * mm);
quantity pDy1  = trap_half_y(25. * mm);
quantity theta = trap_tilt_angle(0. * rad);

// ❌ Level 1: mm where rad expected (pDy1)!
// MakeG4Trap(pDz, pDy1, ...);
// ❌ Level 5: pDy passed where pDz expected (both mm)!
// MakeG4Trap(pDy1, theta, ...);

// ✅ Lengths of different axes promote to
// common hep::length — safe for clearance calcs
quantity half_diag = pDz + pDy1;
static_assert(half_diag.quantity_spec == hep::length);
```

Speed vs. velocity: character matters

Velocity is a *vector* quantity — magnitude and direction.
Speed is its *scalar* magnitude.

Speed vs. velocity: character matters

Velocity is a *vector* quantity — magnitude and direction.
Speed is its *scalar* magnitude.

```
quantity velocity = isq::velocity(vec3{3, 4, 0} * m / s);
quantity<isq::speed[m/s]> speed = magnitude(velocity);        // ✅ 5 m/s

// quantity<isq::speed[m/s]> s = velocity;                    // ❌ Compile error!
// quantity<isq::velocity[m/s]> v = speed;                    // ❌ Compile error!

// quantity<isq::speed[m/s]> s2 = magnitude(speed);           // ❌ Compile error!
```

Speed vs. velocity: character matters

Velocity is a *vector* quantity — magnitude and direction.
Speed is its *scalar* magnitude.

```
quantity velocity = isq::velocity(vec3{3, 4, 0} * m / s);
quantity<isq::speed[m/s]> speed = magnitude(velocity);       // ✅ 5 m/s

// quantity<isq::speed[m/s]> s = velocity;                   // ❌ Compile error!
// quantity<isq::velocity[m/s]> v = speed;                   // ❌ Compile error!

// quantity<isq::speed[m/s]> s2 = magnitude(speed);          // ❌ Compile error!
```

Same dimension (LT^{-1}), same unit (m/s) — **fundamentally different**.

Who sees the issue?

```
quantity force = get_force();
quantity length = get_length();
quantity moment = force * length;
```

Who sees the issue?

```
quantity force = get_force();
quantity length = get_length();
quantity moment = force * length;
```

```
quantity work = force * length;
```

What does it mean to multiply or divide two vectors?

Who sees the issue?

```
quantity force = isq::force(vec3{10., 0., 0.} * N);
quantity displacement = isq::displacement(vec3{5., 0., 0.} * m);
quantity position = isq::position_vector(vec3{1., 2., 0.} * m);

quantity<isq::work[J]> work = scalar_product(force, displacement);
quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(force, position);
```

Who sees the issue?

```
quantity force = isq::force(vec3{10., 0., 0.} * N);
quantity displacement = isq::displacement(vec3{5., 0., 0.} * m);
quantity position = isq::position_vector(vec3{1., 2., 0.} * m);

quantity<isq::work[J]> work = scalar_product(force, displacement);
quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(force, position);
```

Vector product is anti-commutative.

Work vs. moment of force vs. torque

All three share the same dimension (N·m = J), yet are physically and mathematically **distinct**:

QUANTITY	OPERATION	RESULT	CHARACTER
Work	$\vec{F} \cdot \vec{d}$ (scalar product)	J	scalar
Moment of force	$\vec{r} \times \vec{F}$ (vector product)	N·m	vector
Torque	$\|\vec{r} \times \vec{F}\|$ (magnitude)	N·m	scalar

Work vs. moment of force vs. torque

All three share the same dimension (N·m = J), yet are physically and mathematically **distinct**:

QUANTITY	OPERATION	RESULT	CHARACTER
Work	$\vec{F} \cdot \vec{d}$ (scalar product)	J	scalar
Moment of force	$\vec{r} \times \vec{F}$ (vector product)	N·m	vector
Torque	$\|\vec{r} \times \vec{F}\|$ (magnitude)	N·m	scalar

`operator*` on two scalar quantities **cannot represent either** correctly — the angle and the distinction between $\vec{d}$ and $\vec{r}$ would be lost.

Vector operations

```
quantity force = isq::force(vec3{10., 0., 0.} * N);
quantity displacement = isq::displacement(vec3{5., 0., 0.} * m);
quantity position = isq::position_vector(vec3{1., 2., 0.} * m);

// quantity wrong = force * displacement;                   // ❌ Compile error!
```

Vector operations

```
quantity force = isq::force(vec3{10., 0., 0.} * N);
quantity displacement = isq::displacement(vec3{5., 0., 0.} * m);
quantity position = isq::position_vector(vec3{1., 2., 0.} * m);

// quantity wrong = force * displacement;                   // ❌ Compile error!
```

```
quantity<isq::work[J]> work = scalar_product(force, displacement);                      // ✅
quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(position, force);    // ✅
// quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(force, position); // ❌
```

Vector operations

```
quantity force = isq::force(vec3{10., 0., 0.} * N);
quantity displacement = isq::displacement(vec3{5., 0., 0.} * m);
quantity position = isq::position_vector(vec3{1., 2., 0.} * m);

// quantity wrong = force * displacement;                    // ❌ Compile error!
```

```
quantity<isq::work[J]> work = scalar_product(force, displacement);                       // ✅
quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(position, force);    // ✅
// quantity<isq::moment_of_force[N * m], vec3> moment = vector_product(force, position); // ❌
```

```
// Torque is the scalar magnitude of moment of force
quantity<isq::torque[N * m]> torque = magnitude(moment);     // ✅

// Work and torque are distinct – cannot assign one to the other:
// quantity<isq::torque[N * m]> wrong = work;                // ❌ Compile error!
// quantity<isq::work[J]> wrong = torque;                    // ❌ Compile error!
```

AC power: four distinct quantities

```
// Boost.Units — all collapse to the same type:
bu::quantity<bu::si::power> P = 800. * bu::si::watts;
bu::quantity<bu::si::power> Q = 600. * bu::si::watts;    // Oops! Should be var
auto nonsense = P + Q;                                   // Oops! Compiles!
```

AC power: four distinct quantities

```
// Boost.Units — all collapse to the same type:
bu::quantity<bu::si::power> P = 800. * bu::si::watts;
bu::quantity<bu::si::power> Q = 600. * bu::si::watts;     // Oops! Should be var
auto nonsense = P + Q;                                     // Oops! Compiles!
```

W, var, and VA are all dimensionally equivalent to J/s — but **not interchangeable**.

AC power: four distinct quantities

QUANTITY	UNIT	CHARACTER	PHYSICAL MEANING
Active power	W	real scalar	energy actually consumed
Reactive power	var	real scalar	energy oscillating
Complex power	VA	complex scalar	full phasor $S = P + jQ$
Apparent power	VA	real scalar	$\|S\| = \sqrt{P^2 + Q^2}$

AC power: four distinct quantities

QUANTITY	UNIT	CHARACTER	PHYSICAL MEANING
Active power	W	real scalar	energy actually consumed
Reactive power	var	real scalar	energy oscillating
Complex power	VA	complex scalar	full phasor $S = P + jQ$
Apparent power	VA	real scalar	$\|S\| = \sqrt{P^2 + Q^2}$

At CppCon, a power systems engineer stated that any units library would be **useless in production** unless it correctly prevents these mistakes.

AC power done right

```
quantity P = isq::active_power(800. * W);
// quantity Q = isq::reactive_power(600 * W);          // ❌ Error: wrong unit!
quantity Q = isq::reactive_power(600. * var);          // ✅ OK
```

AC power done right

```
quantity P = isq::active_power(800. * W);
// quantity Q = isq::reactive_power(600 * W);                 // ❌ Error: wrong unit!
quantity Q = isq::reactive_power(600. * var);                 // ✅ OK
```

```
// Apparent power from proper equation
quantity<isq::apparent_power[VA]> S = hypot<VA>(P, Q);       // ✅ 1000 VA
// auto wrong = P + Q;                                        // ❌ Error: invalid operation
```

AC power done right

```cpp
quantity P = isq::active_power(800. * W);
// quantity Q = isq::reactive_power(600 * W);                   // ❌ Error: wrong unit!
quantity Q = isq::reactive_power(600. * var);                   // ✅ OK
```

```cpp
// Apparent power from proper equation
quantity<isq::apparent_power[VA]> S = hypot<VA>(P, Q);          // ✅ 1000 VA
// auto wrong = P + Q;                                          // ❌ Error: invalid operation
```

```cpp
// Complex power with correct argument order
quantity<isq::complex_power[VA], std::complex<double>> complex =
    make_complex_quantity(P, Q);                                // ✅ S = P + jQ
// quantity<isq::complex_power[VA], std::complex<double>> complex =
// make_complex_quantity(Q, P);                                 // ❌ Error: wrong order!
```

AC power done right

```
quantity P = isq::active_power(800. * W);
// quantity Q = isq::reactive_power(600 * W);                 // ❌ Error: wrong unit!
quantity Q = isq::reactive_power(600. * var);                  // ✅ OK
```

```
// Apparent power from proper equation
quantity<isq::apparent_power[VA]> S = hypot<VA>(P, Q);        // ✅ 1000 VA
// auto wrong = P + Q;                                         // ❌ Error: invalid operation
```

```
// Complex power with correct argument order
quantity<isq::complex_power[VA], std::complex<double>> complex =
    make_complex_quantity(P, Q);                               // ✅ S = P + jQ
// quantity<isq::complex_power[VA], std::complex<double>> complex =
// make_complex_quantity(Q, P);                                // ❌ Error: wrong order!
```

```
// Extract components
quantity<isq::active_power[W]> P2 = real(complex);            // ✅ 800 W
quantity<isq::reactive_power[var]> Q2 = imag(complex);        // ✅ 600 var
quantity<isq::apparent_power[VA]> S2 = modulus(complex);      // ✅ 1000 VA
// quantity<isq::active_power[W]> wrong = imag(complex);      // ❌ Error!
```

Try it yourself! 🧪

- Refactor the **`find_route`** function and its usage to use semantic wrappers (typed quantities) for each argument, so that accidental swaps are caught at compile time. Make sure to update the return type as well.
- You do not need to introduce a typed quantity for every physical value—use them where confusion is possible (e.g., **`length`**), and keep others simple (e.g., **`duration`**).
- Check how this approach prevents mistakes and clarifies the API.
- Try to pass **`turn_radius`** argument to width function parameter and vice-versa. Analyze the difference and propose a solution to prevent this issue.

LEVEL 6: AFFINE SPACE SAFETY

Issue #457

QuanityKind and Altitudes #457

Answered by mpusz

jasonbeach asked this question in Q&A

jasonbeach on May 9, 2023

I've just started really looking through the documentation on mp-units and really got my curiosity piqued when I read about quantity point kinds and altitudes, since tracking the datum of an altitude tends to be a challenge. The glide computer example illustrates this well--the datum is hard-coded assumed to be MSL, but not all altitudes are MSL. Is there a way to track and ideally automatically convert between different datums? Common datums in UAV autonomy (my use-case) are MSL (Mean Sea Level), HAE (Height above WGS84 Ellipsoid), HAL (height above launch), AGL (above ground level or height above terrain). Inadvertently mixing altitudes with different datums is a fruitful source of bugs. One challenge to converting between them is the offsets to convert between them are not fixed (can't be done at compile time) and in some cases it changes depending on your geographic location (Geoid undulation which is offset between HAE and MSL varies depending on your location.) I guess at a minimum it would be at least be nice to be able to have a compile time error if you mix altitudes with two different datums. Is that possible?

The affine space

The affine space has two types of entities:

- **point** — a position specified with coordinate values (e.g., temperature, timestamp, altitude)
- **displacement vector** — the difference between two points (e.g., temperature difference, duration)

The affine space operations

EXPRESSION	RESULT
point − point	vector
point + vector	point
point − vector	point
vector + vector	vector
point + point	❌
vector - point	❌
point × scalar	❌
point × point	❌

Points are everywhere

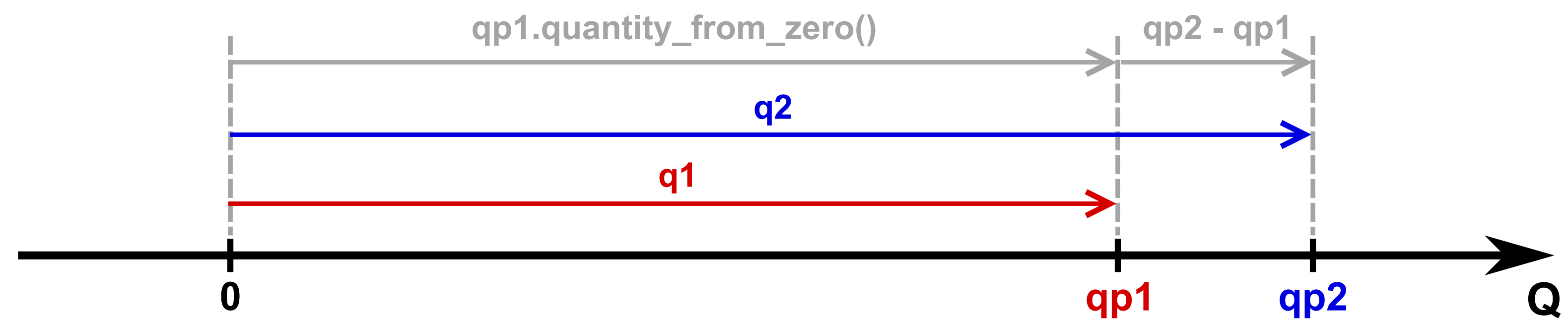
- *Temperature* points (absolute temperatures)
- *Timestamps*
- Daily *mass readouts* from the scale
- *Altitudes* of mountain peaks on a map
- Current path length measured by the car's *odometer*
- *Today's price* of instruments on the market
- *GPS coordinates*
- *Voltage levels* in a circuit
- ...

Points are everywhere

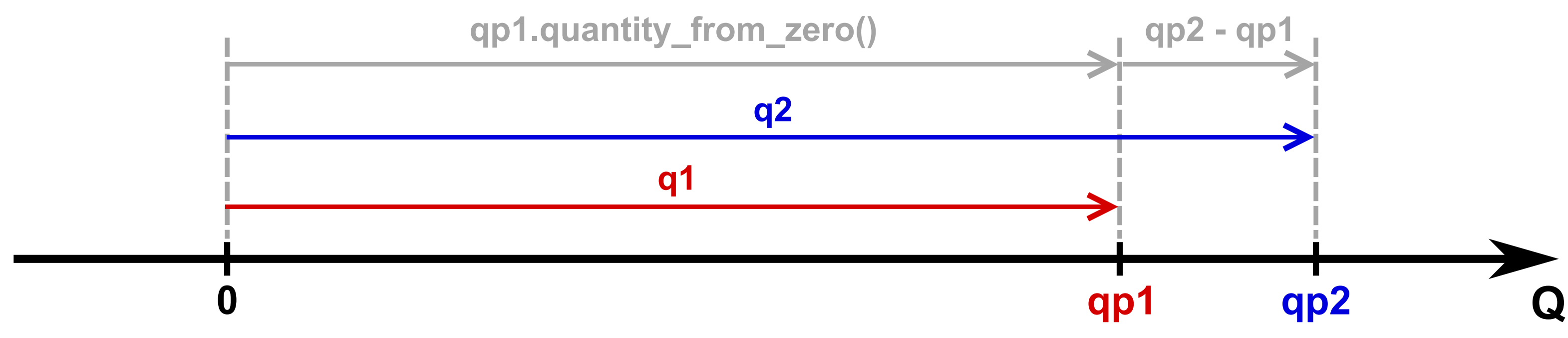
- *Temperature* points (absolute temperatures)
- *Timestamps*
- Daily *mass readouts* from the scale
- *Altitudes* of mountain peaks on a map
- Current path length measured by the car's *odometer*
- *Today's price* of instruments on the market
- *GPS coordinates*
- *Voltage levels* in a circuit

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

zeroth_point_origin — the default

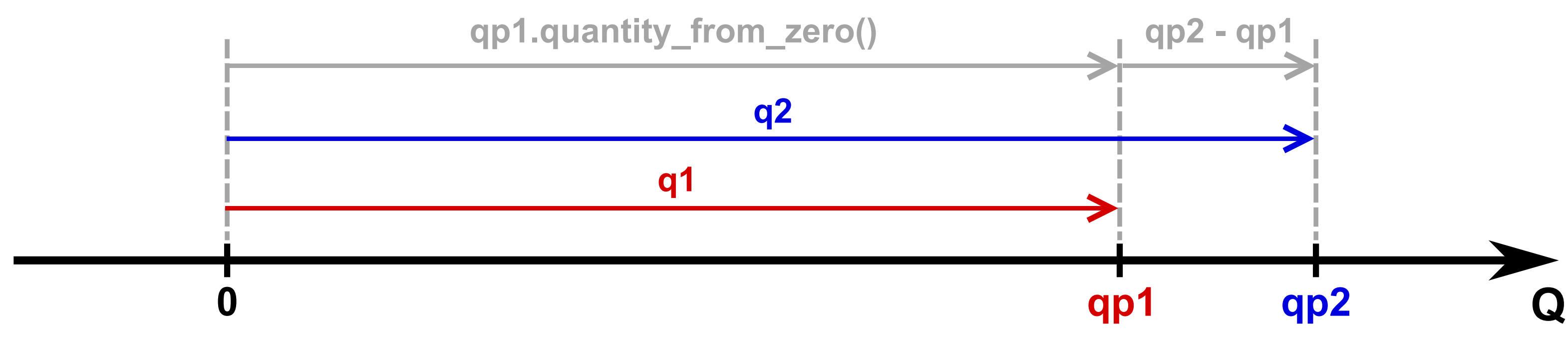


zeroth_point_origin — the default




```
quantity_point qp1 = point<isq::distance[m]>(100);
quantity_point qp2 = point<isq::distance[m]>(120);

assert(qp2 - qp1 == 20 * m);
// auto bad = qp1 + qp2;    // ❌ Compile-time error
```

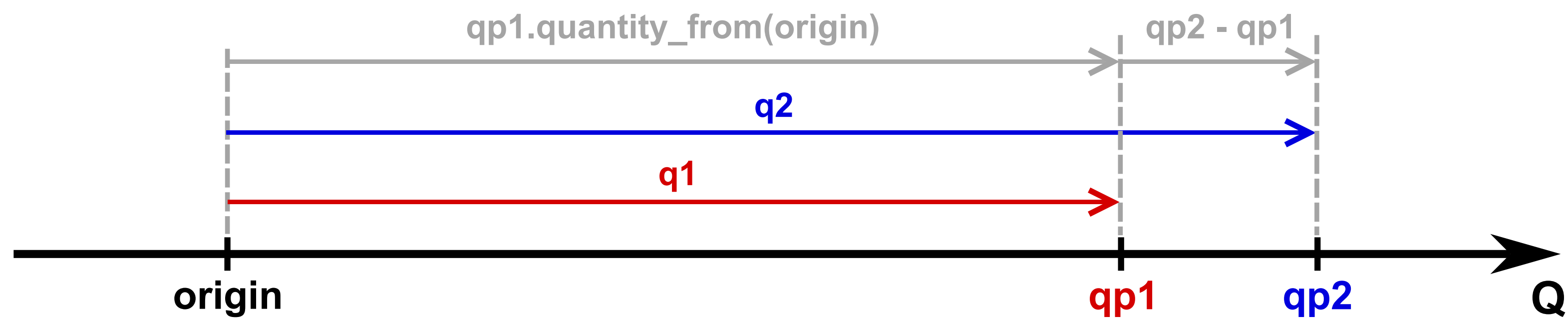
zeroth_point_origin — the default



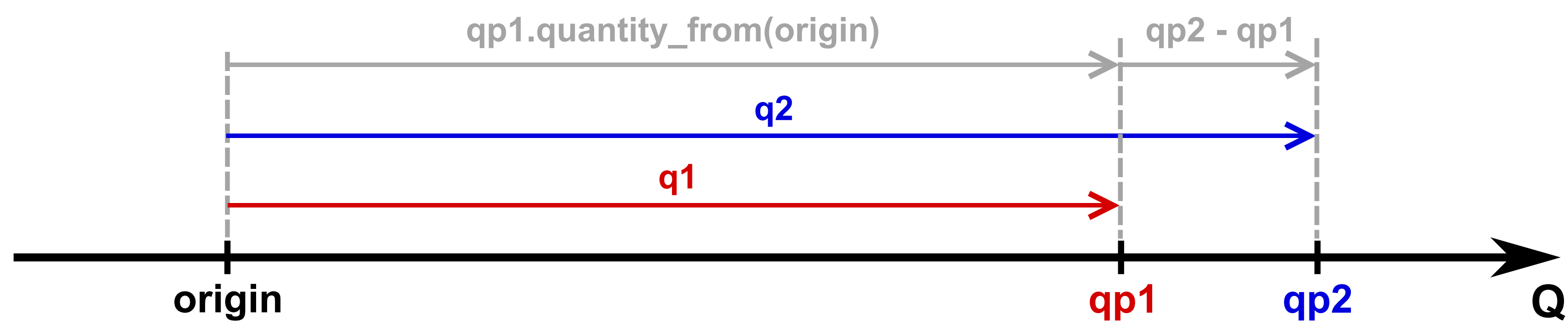

```
quantity_point qp1 = point<isq::distance[m]>(100);
quantity_point qp2 = point<isq::distance[m]>(120);

assert(qp2 - qp1 == 20 * m);
// auto bad = qp1 + qp2;    // ❌ Compile-time error
```

The default origin is at the zeroth point of the measurement scale. All **quantity_point** types with compatible quantity specs share the same origin.

Absolute point origin

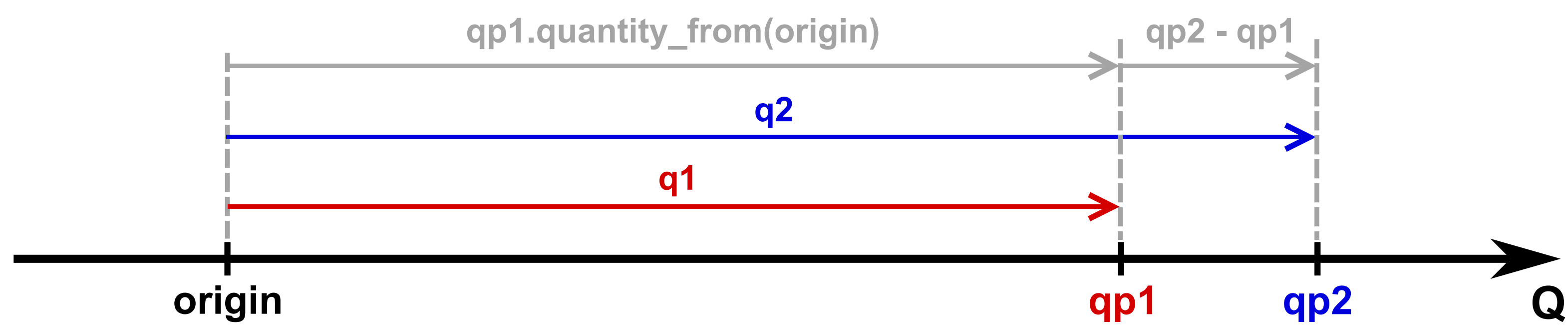


Absolute point origin




```
inline constexpr struct origin final : absolute_point_origin<isq::distance> {} origin;

quantity_point qp1 = origin + 100 * m;
quantity_point qp2 = origin + 120 * m;

assert(qp1.quantity_from(origin) == 100 * m);
assert(qp2 - qp1 == 20 * m);
```

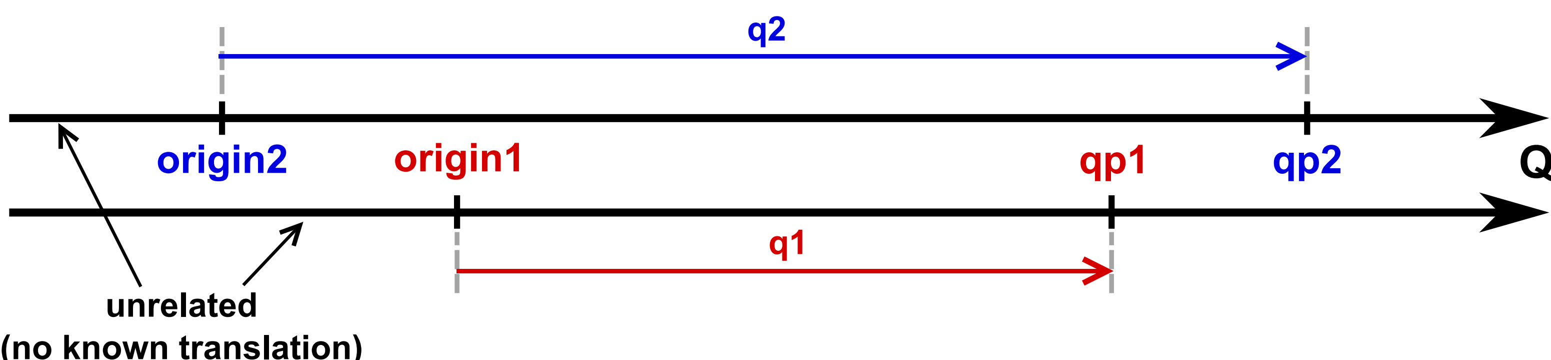
Absolute point origin



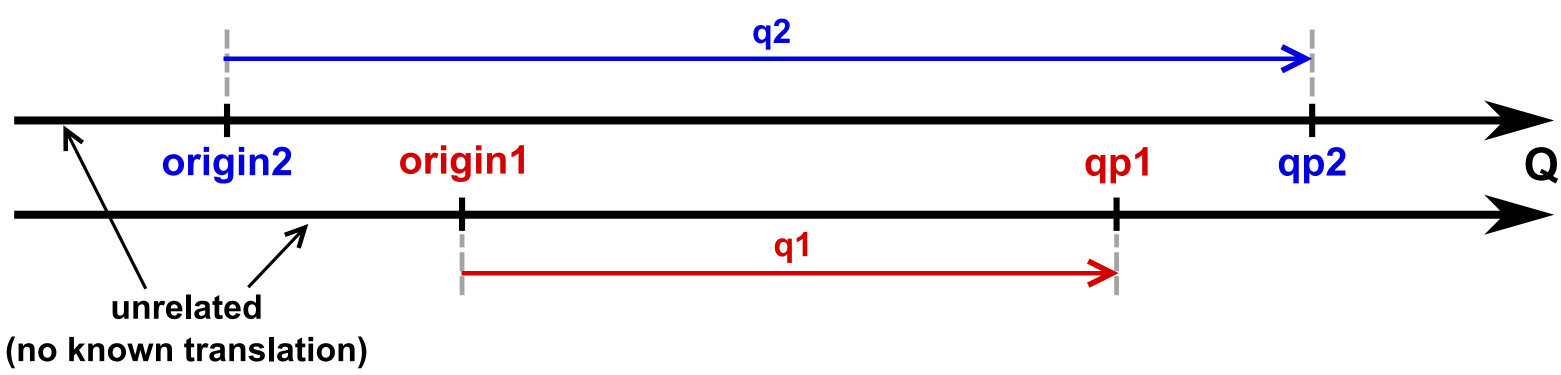

```
inline constexpr struct origin final : absolute_point_origin<isq::distance> {} origin;

quantity_point qp1 = origin + 100 * m;
quantity_point qp2 = origin + 120 * m;

assert(qp1.quantity_from(origin) == 100 * m);
assert(qp2 - qp1 == 20 * m);
```

With a named origin, **`quantity_from_zero()`** is not available — you must always specify which origin you measure from.

Independent absolute origins

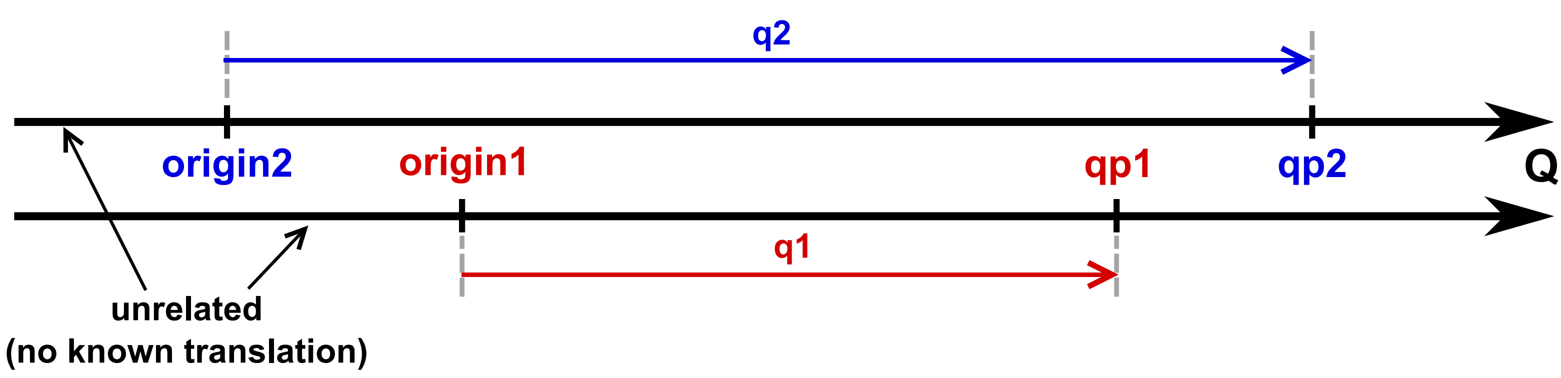


Independent absolute origins




```cpp
inline constexpr struct origin1 final : absolute_point_origin<isq::distance> {} origin1;
inline constexpr struct origin2 final : absolute_point_origin<isq::distance> {} origin2;

quantity_point qp1 = origin1 + 100 * m;
quantity_point qp2 = origin2 + 120 * m;
// auto d = qp2 - qp1;   // ❌ Compile-time error (different origins)
```

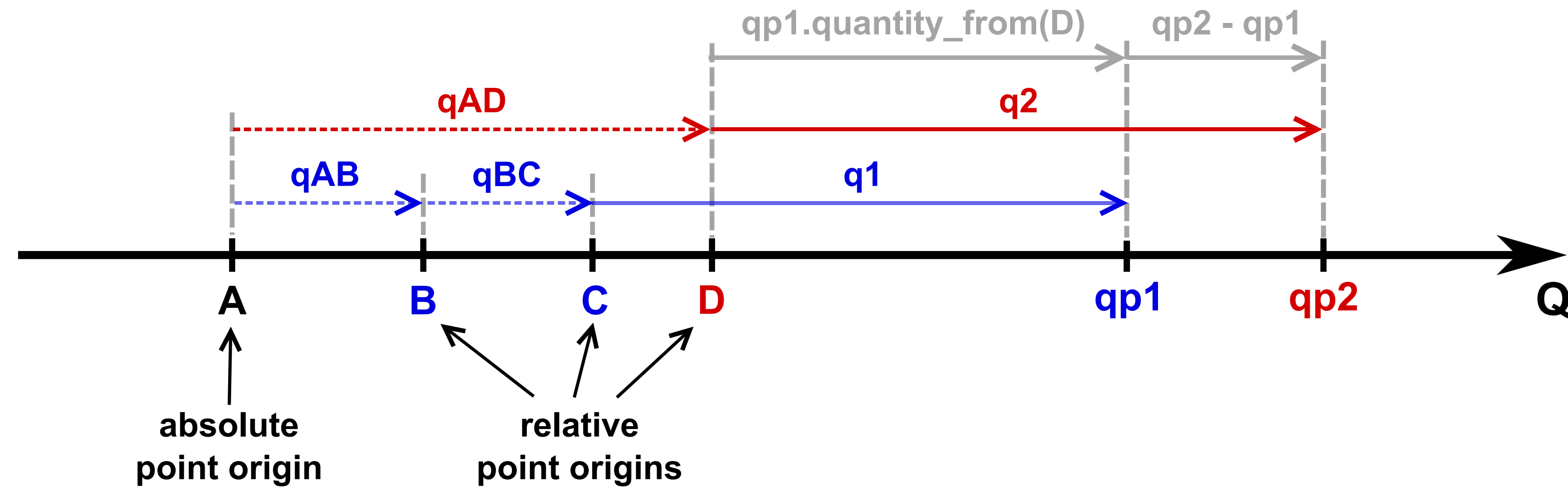
Independent absolute origins



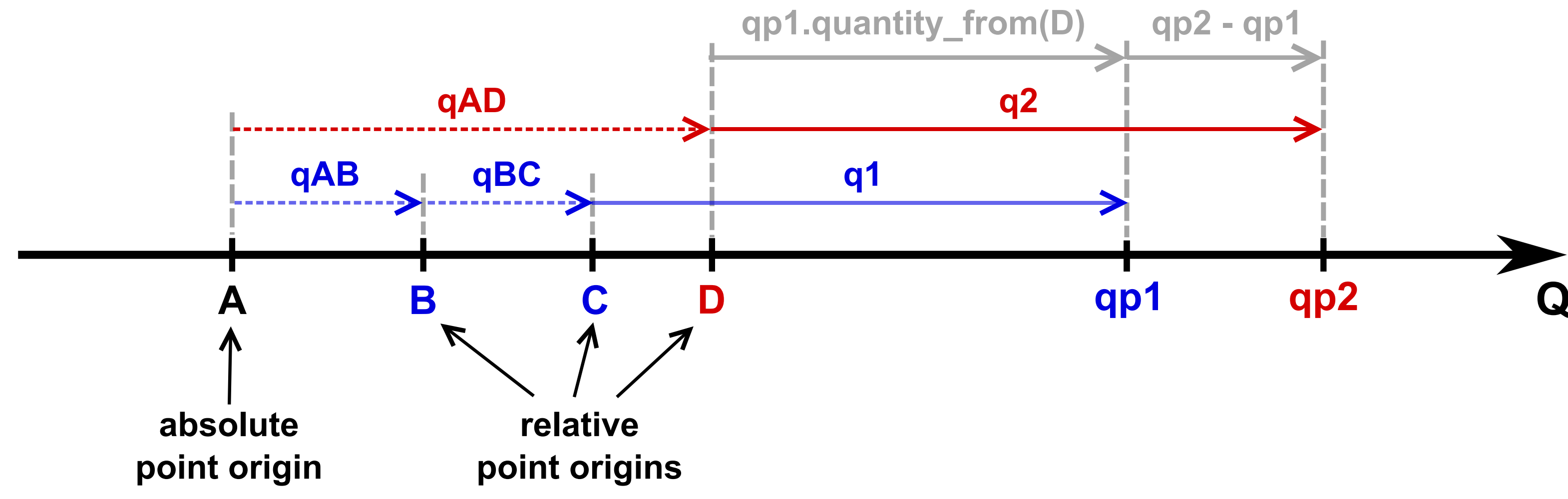

```
inline constexpr struct origin1 final : absolute_point_origin<isq::distance> {} origin1;
inline constexpr struct origin2 final : absolute_point_origin<isq::distance> {} origin2;

quantity_point qp1 = origin1 + 100 * m;
quantity_point qp2 = origin2 + 120 * m;
// auto d = qp2 - qp1;   // ❌ Compile-time error (different origins)
```

Absolute point origins create **isolated spaces**. Even with the same quantity type and unit, points from different spaces cannot be mixed.

Relative point origins

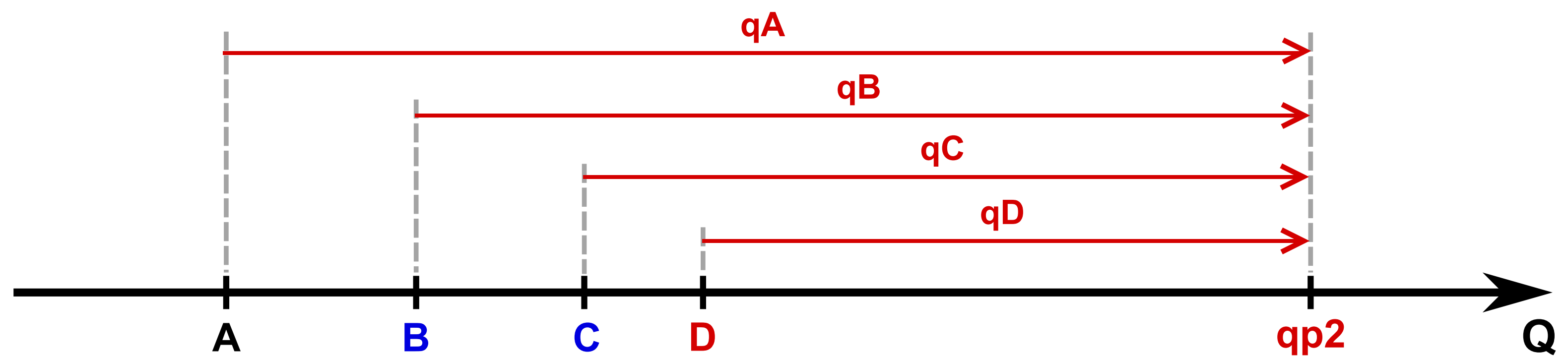


Relative point origins

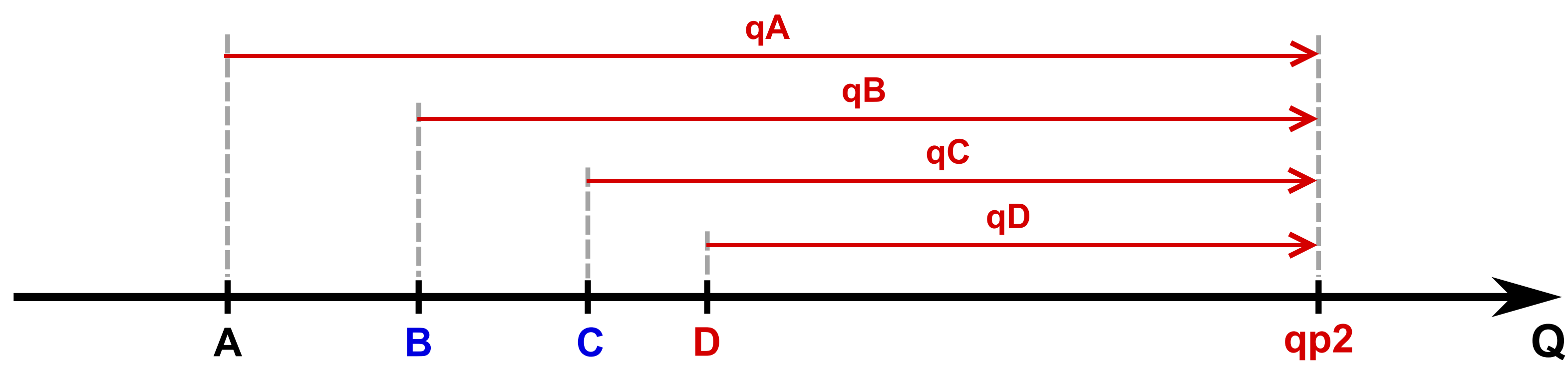



```
inline constexpr struct A final : absolute_point_origin<isq::distance> {} A;
inline constexpr struct B final : relative_point_origin<A + 10 * m> {} B;
inline constexpr struct C final : relative_point_origin<B + 10 * m> {} C;
inline constexpr struct D final : relative_point_origin<A + 30 * m> {} D;

quantity_point qp1 = C + 100 * m;
quantity_point qp2 = D + 120 * m;

assert(qp2 - qp1 == 30 * m);    // ✅ automatic offset handling
assert(qp1.quantity_from(D) == 90 * m);
assert(D - C == 10 * m);
```

Converting between representations

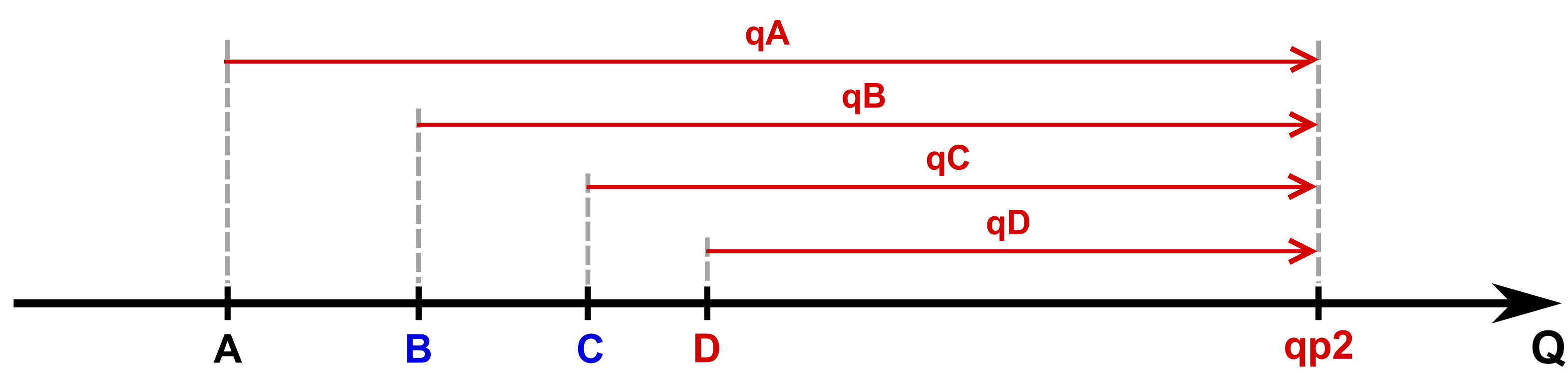


Converting between representations



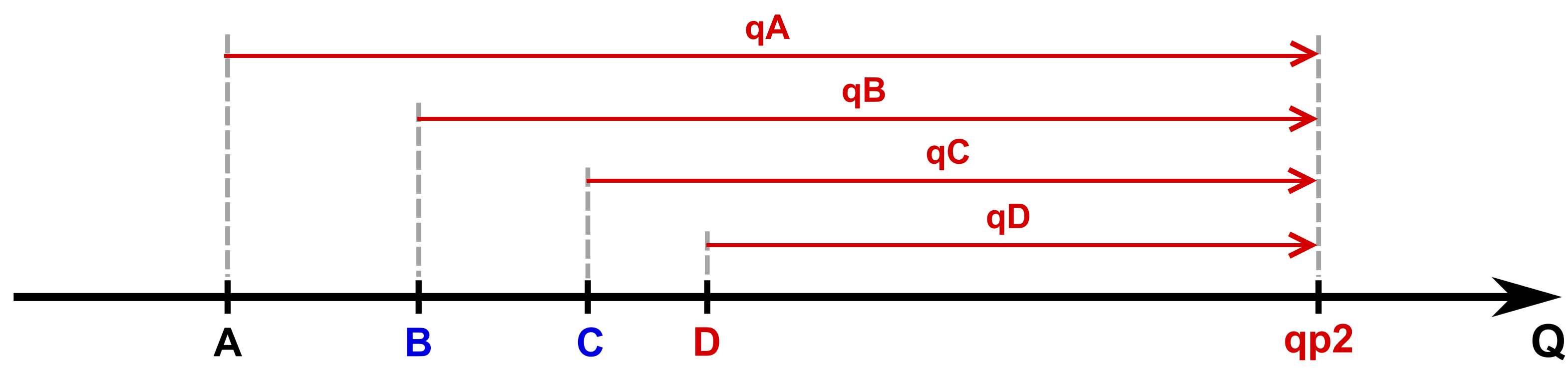
All representations describe **the same point** — only the displacement vector from each origin differs. Conversions are only allowed within origins that share the same **`absolute_point_origin`**.

Converting between representations



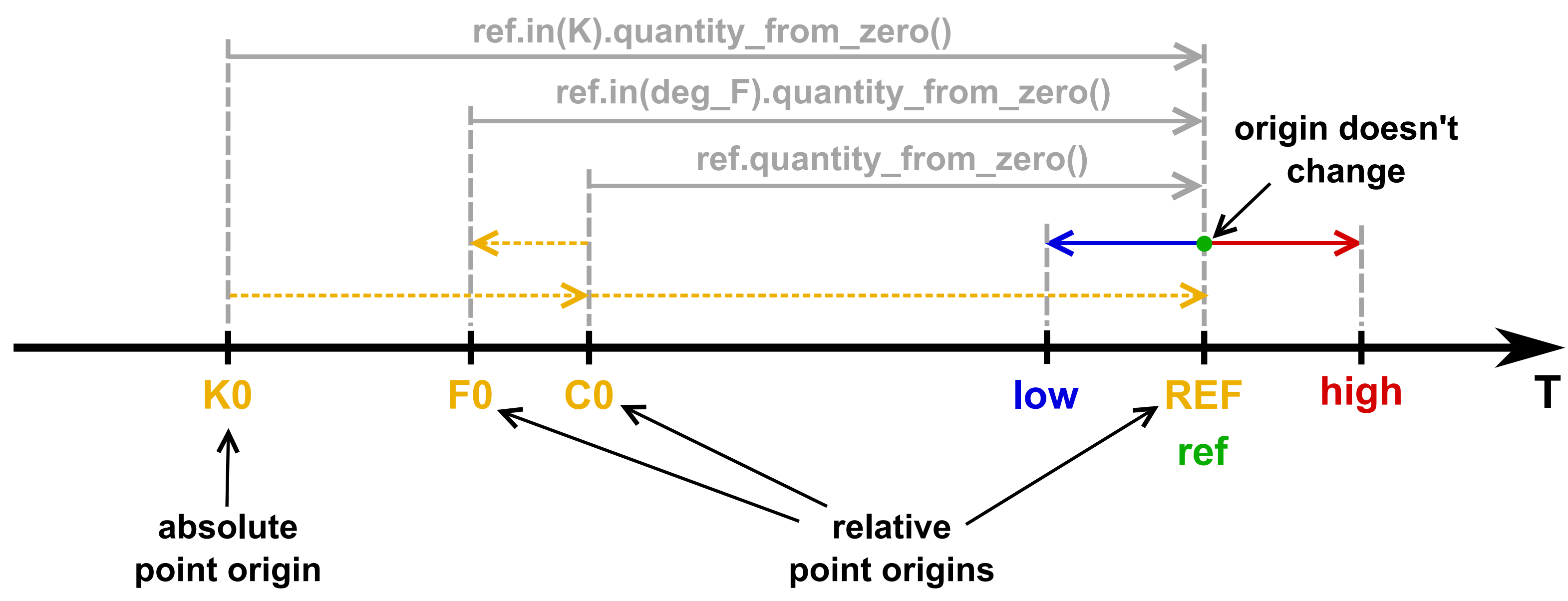

```
quantity_point qp2 = D + 120 * m;

quantity_point qp2C = qp2.point_for(C);
quantity_point qp2A = qp2.point_for(A);

assert(qp2C.quantity_ref_from(C) == 130 * m);
assert(qp2A.quantity_ref_from(A) == 150 * m);
assert(qp2 == qp2C);    // ✅ same point
assert(qp2 == qp2A);    // ✅ same point
```

Converting between representations

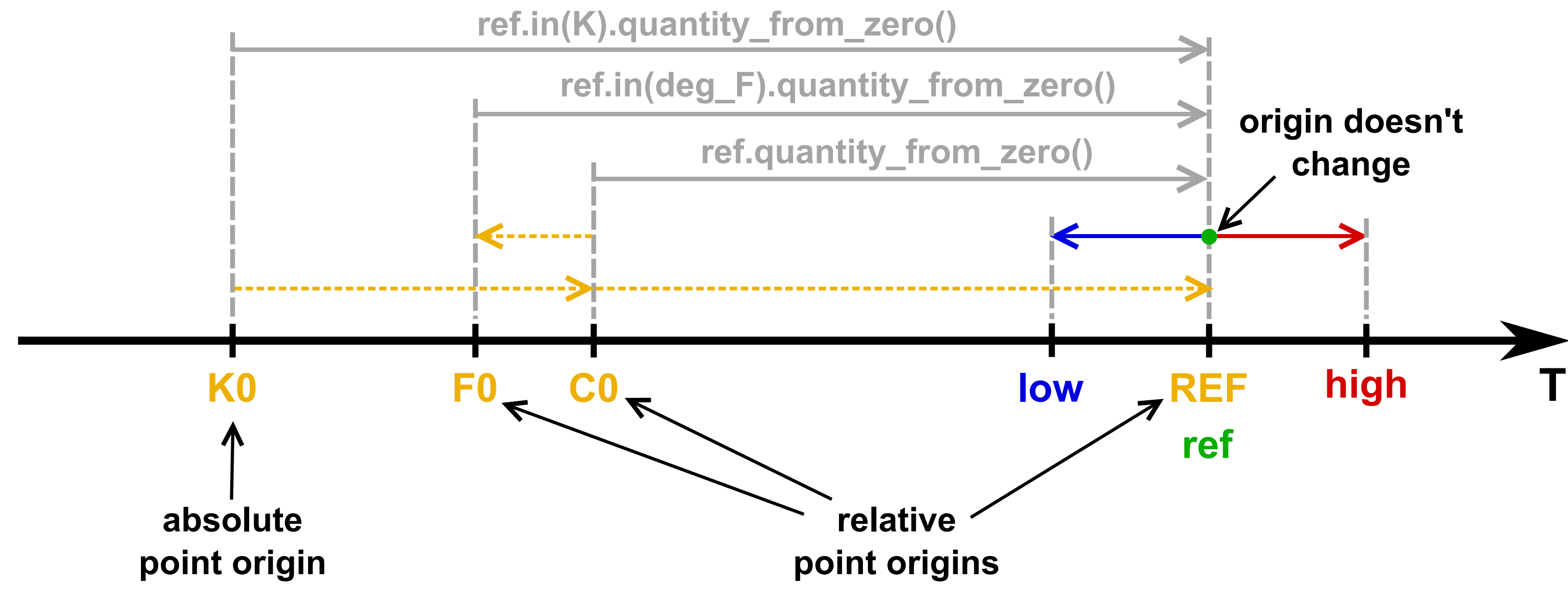


This is the major reason why we do not provide text output for quantity points by default (at least for now). Various representation of the same point should yield the same output. This knowledge is often domain specific.

Temperature point origins

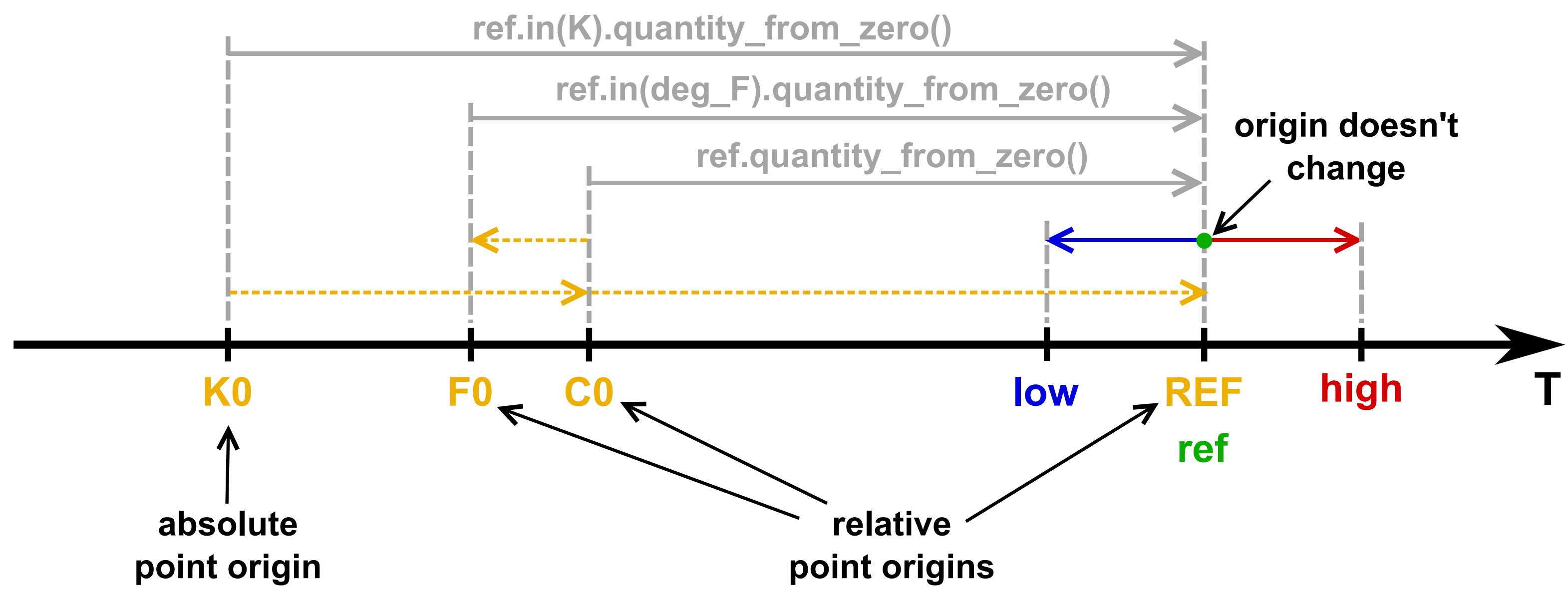


Temperature point origins



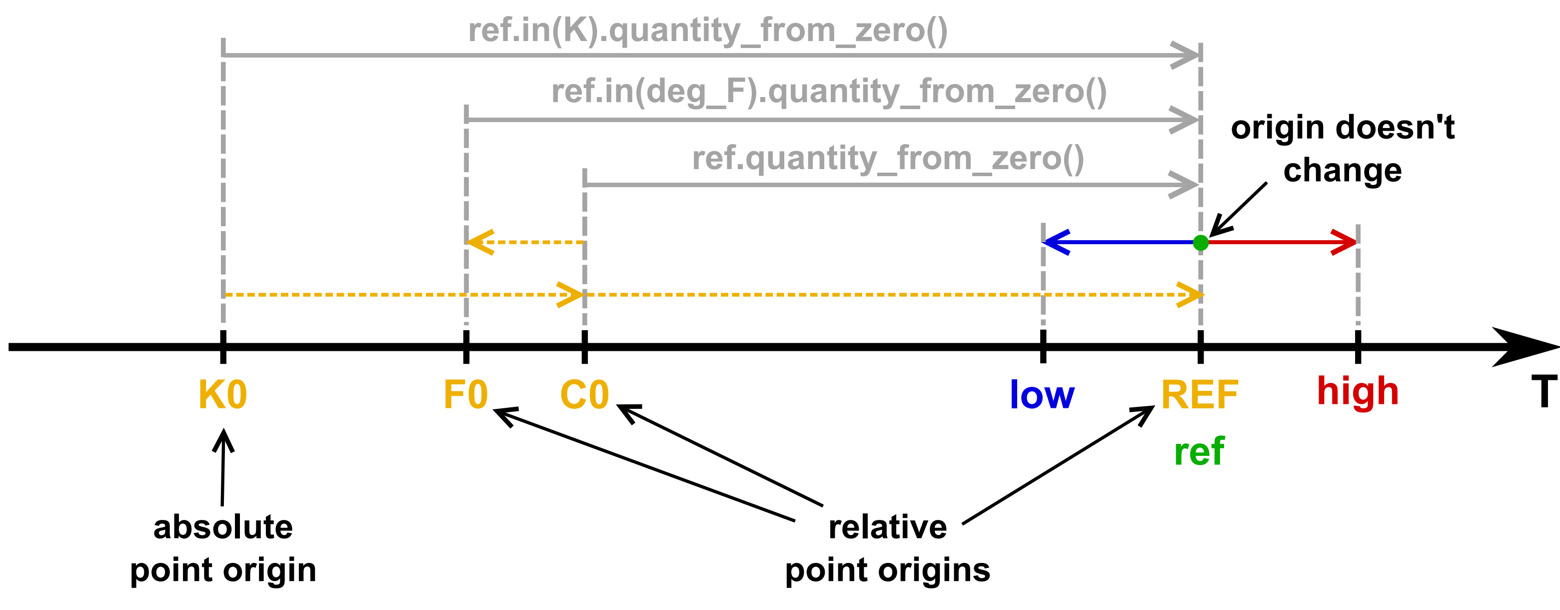
Point origins can be **stacked**: `zeroth_degree_Fahrenheit` is relative to `ice_point`, which is relative to `absolute_zero`. They may also use different units and representation types.

Temperature point origins




```
inline constexpr struct absolute_zero final : absolute_point_origin<isq::thermodynamic_temperature> {} absolute_zero;
inline constexpr struct ice_point final : relative_point_origin<point<milli<kelvin>>(273'150)> {} ice_point;
inline constexpr struct zeroth_degree_Fahrenheit final :
    relative_point_origin<point<mag_ratio<5, 9> * si::degree_Celsius>(-32)> {} zeroth_degree_Fahrenheit;

constexpr struct room_reference_temp final : relative_point_origin<point<deg_C>(21)> {} room_reference_temp;

using room_temp = quantity_point<deg_C, room_reference_temp>;
```

Temperature point origins




```
room_temp room_ref{};
room_temp room_low = room_ref - delta<deg_C>(4);
room_temp room_high = room_ref + delta<deg_C>(4);

std::println("Room reference temperature: {} ({}, {::N[.2f]})\n",
             room_ref.quantity_from_zero(),
             room_ref.in(usc::degree_Fahrenheit).quantity_from_zero(),
             room_ref.in(si::kelvin).quantity_from_zero());
```

Why 42 * deg_C does not compile?

```
quantity q1 = 42 * m;                   // ✅ 42 m
// quantity q2 = 42 * deg_C;            // ❌ Compile-time error!
```

Why 42 * deg_C does not compile?

```
quantity q1 = 42 * m;                    // ✅ 42 m
// quantity q2 = 42 * deg_C;             // ❌ Compile-time error!
```

Is 42 °C

- a **temperature point** (a reading on a thermometer), or
- a **temperature difference** (how much warmer)?

Why `42 * deg_C` does not compile?

For units with a point origin in their definition (°C, °F, K), the multiply syntax is **disabled** to prevent ambiguity. Use **`delta<U>`** for differences and **`point<U>`** for absolute readings.

Why 42 * deg_C does not compile?

For units with a point origin in their definition (°C, °F, K), the multiply syntax is **disabled** to prevent ambiguity. Use **delta<U>** for differences and **point<U>** for absolute readings.

```
// Displacement vector (temperature difference)
quantity delta_t = delta<deg_C>(42);          // 42 °C difference

// Point (absolute temperature)
quantity_point temp = point<deg_C>(42);       // 42 °C on the Celsius scale
```

The temperature conversion trap

```
// Temperature POINT: 20 °C → 68 °F  (correct: × 9/5 + 32)
double fahrenheit = celsius * 9.0 / 5.0 + 32.0;   // ✅ for points

// Temperature DIFFERENCE: 10 °C → ?
double diff_f = diff_c * 9.0 / 5.0 + 32.0;        // ❌ 50 °F (Should be 18 °F)
```

The temperature conversion trap

```
// Temperature POINT: 20 °C → 68 °F  (correct: × 9/5 + 32)
double fahrenheit = celsius * 9.0 / 5.0 + 32.0;   // ✅ for points

// Temperature DIFFERENCE: 10 °C → ?
double diff_f = diff_c * 9.0 / 5.0 + 32.0;        // ❌ 50 °F (Should be 18 °F)
```

```
quantity_point t = point<deg_C>(20.);
quantity delta = 10. * delta<deg_C>;
std::println("{}", t.in(deg_F).quantity_from_zero());   // 68 °F ✅
std::println("{}", delta.in(deg_F));                    // 18 °F ✅
```

The temperature conversion trap

```
// Temperature POINT: 20 °C → 68 °F  (correct: × 9/5 + 32)
double fahrenheit = celsius * 9.0 / 5.0 + 32.0;    // ✅ for points

// Temperature DIFFERENCE: 10 °C → ?
double diff_f = diff_c * 9.0 / 5.0 + 32.0;         // ❌ 50 °F (Should be 18 °F)
```

```
quantity_point t = point<deg_C>(20.);
quantity delta = 10. * delta<deg_C>;
std::println("{}", t.in(deg_F).quantity_from_zero());   // 68 °F ✅
std::println("{}", delta.in(deg_F));                    // 18 °F ✅
```

The library **automatically distinguishes** between point and delta conversions. The Guardian article got this wrong: "temperatures nearly 68 °F above average" instead of 32 °F.

Different reference origins

```
inline constexpr struct amsterdam_sea_level final :
    absolute_point_origin<isq::altitude> {} amsterdam_sea_level;
inline constexpr struct mediterranean_sea_level final :
    relative_point_origin<amsterdam_sea_level - 27 * cm> {} mediterranean_sea_level;
```

Different reference origins

```
inline constexpr struct amsterdam_sea_level final :
    absolute_point_origin<isq::altitude> {} amsterdam_sea_level;
inline constexpr struct mediterranean_sea_level final :
    relative_point_origin<amsterdam_sea_level - 27 * cm> {} mediterranean_sea_level;
```

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

altitude_DE de_side(330 * m);
altitude_CH ch_side(300 * m);

// The 27 cm offset between origins is handled automatically
quantity diff = de_side - ch_side;
```

Different reference origins

```cpp
inline constexpr struct amsterdam_sea_level final :
    absolute_point_origin<isq::altitude> {} amsterdam_sea_level;
inline constexpr struct mediterranean_sea_level final :
    relative_point_origin<amsterdam_sea_level - 27 * cm> {} mediterranean_sea_level;
```

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

altitude_DE de_side(330 * m);
altitude_CH ch_side(300 * m);

// The 27 cm offset between origins is handled automatically
quantity diff = de_side - ch_side;
```

Mixing **`quantity_points`** with **unrelated origins** is a compile-time error.

Mixing points with **related origins** applies the offset automatically.

Detector vertex positions (HEP)

```
// Track vertex in the ATLAS detector coordinate system
quantity_point z_vertex = atlas_origin + 150 * mm;

// Decay vertex: offset from production vertex
quantity_point z_decay = z_vertex + decay_length * cos(theta);
```

Detector vertex positions (HEP)

```
// Track vertex in the ATLAS detector coordinate system
quantity_point z_vertex = atlas_origin + 150 * mm;

// Decay vertex: offset from production vertex
quantity_point z_decay = z_vertex + decay_length * cos(theta);
```

```
// ❌ Cannot add two position points
// auto bad = z_vertex + z_decay;

// ✅ Can compute distance between points (vector)
quantity flight_path = z_decay - z_vertex;
```

Detector vertex positions (HEP)

```
// Track vertex in the ATLAS detector coordinate system
quantity_point z_vertex = atlas_origin + 150 * mm;

// Decay vertex: offset from production vertex
quantity_point z_decay = z_vertex + decay_length * cos(theta);
```

```
// ❌ Cannot add two position points
// auto bad = z_vertex + z_decay;

// ✅ Can compute distance between points (vector)
quantity flight_path = z_decay - z_vertex;
```

Position points in a coordinate system are **not vectors** — you can find the distance between them, but you cannot add them.

Issue #457

```cpp
inline constexpr struct mean_sea_level final : absolute_point_origin<isq::altitude> {} mean_sea_level;
using msl_altitude = quantity_point<isq::altitude[si::metre], mean_sea_level>;
```

Issue #457

```
inline constexpr struct mean_sea_level final : absolute_point_origin<isq::altitude> {} mean_sea_level;
using msl_altitude = quantity_point<isq::altitude[si::metre], mean_sea_level>;

inline constexpr struct height_above_launch final : absolute_point_origin<isq::altitude> {} height_above_launch;
using hal_altitude = quantity_point<isq::altitude[si::metre], height_above_launch>;
```

Issue #457

```cpp
inline constexpr struct mean_sea_level final : absolute_point_origin<isq::altitude> {} mean_sea_level;
using msl_altitude = quantity_point<isq::altitude[si::metre], mean_sea_level>;

inline constexpr struct height_above_launch final : absolute_point_origin<isq::altitude> {} height_above_launch;
using hal_altitude = quantity_point<isq::altitude[si::metre], height_above_launch>;

enum class earth_gravity_model : std::int8_t { egm84_15, egm95_5, egm2008_1 };

template<earth_gravity_model M>
struct height_above_ellipsoid_t final : absolute_point_origin<isq::altitude> {
  static constexpr earth_gravity_model egm = M;
};
template<earth_gravity_model M>
constexpr height_above_ellipsoid_t<M> height_above_ellipsoid;

template<earth_gravity_model M>
using hae_altitude = quantity_point<isq::altitude[si::metre], height_above_ellipsoid<M>>;
```

Issue #457

```cpp
inline constexpr struct mean_sea_level final : absolute_point_origin<isq::altitude> {} mean_sea_level;
using msl_altitude = quantity_point<isq::altitude[si::metre], mean_sea_level>;

inline constexpr struct height_above_launch final : absolute_point_origin<isq::altitude> {} height_above_launch;
using hal_altitude = quantity_point<isq::altitude[si::metre], height_above_launch>;

enum class earth_gravity_model : std::int8_t { egm84_15, egm95_5, egm2008_1 };

template<earth_gravity_model M>
struct height_above_ellipsoid_t final : absolute_point_origin<isq::altitude> {
  static constexpr earth_gravity_model egm = M;
};
template<earth_gravity_model M>
constexpr height_above_ellipsoid_t<M> height_above_ellipsoid;

template<earth_gravity_model M>
using hae_altitude = quantity_point<isq::altitude[si::metre], height_above_ellipsoid<M>>;

template<earth_gravity_model M>
hae_altitude<M> to_hae(msl_altitude msl, position<double> pos)
{
  const auto geoid_undulation =
    isq::height(GeographicLibWhatsMyOffset(pos.lat.quantity_from_zero().numerical_value_in(si::degree),
                                           pos.lon.quantity_from_zero().numerical_value_in(si::degree)) * si::metre);
  return height_above_ellipsoid<M> + (msl - mean_sea_level - geoid_undulation);
}
```

Try it yourself! 🧪

- Define *absolute height points* for the bridge deck and river surface, each with their respective origins (NHN and Swiss reference system).
- Convert one measurement to the other's reference system, being careful to account for the 27 cm offset that caused the famous Hochrheinbrücke construction challenge.
- Compute the *height difference* (clearance) between the bridge and the river in meters ($h_{\text{clearance}} = alt_{\text{bridge}} - alt_{\text{river}}$).

Six levels — summary

LEVEL	PROTECTION	EXAMPLE
1. DIMENSION	No mixing m + s	`distance + time` → ❌
2. UNIT	Auto conversions, safe getters	`km` → `m`, `numerical_value_in(s)`
3. REPRESENTATION	No silent truncation	`int(500m)` → `int(km)` → ❌
4. QUANTITY KIND	Gy ≠ Sv	`Gy + Sv` → ❌
5. QUANTITY	height ≠ width at interfaces	`height` where `width` expected → ❌
6. AFFINE SPACE	points ≠ vectors	`temp1 + temp2` → ❌

Six levels — summary

LEVEL	PROTECTION	EXAMPLE
1. DIMENSION	No mixing m + s	`distance + time` → ❌
2. UNIT	Auto conversions, safe getters	`km` → `m`, `numerical_value_in(s)`
3. REPRESENTATION	No silent truncation	`int(500m)` → `int(km)` → ❌
4. QUANTITY KIND	Gy ≠ Sv	`Gy + Sv` → ❌
5. QUANTITY	height ≠ width at interfaces	`height` where `width` expected → ❌
6. AFFINE SPACE	points ≠ vectors	`temp1 + temp2` → ❌

All checks are compile-time with zero runtime cost. `std::chrono::duration` provides levels 1–3 for time. P3045 provides all six levels for all quantities.

Metrology is an inherently complex problem

Metrology is an inherently complex problem

Six levels of compile-time safety — not because the library is over-engineered, but because **the domain demands it**. Every level prevents a real class of bugs.

SCOPE OF THE CORE LIBRARY FRAMEWORK

Why all of these features are needed?

A common reaction is: "Why not just standardize a simple dimension/unit checker?"

Why all of these features are needed?

A common reaction is: "Why not just standardize a simple dimension/unit checker?"

- **Levels 1–3** (dimension, unit, representation) are the established minimum for all units libraries

Why all of these features are needed?

A common reaction is: "Why not just standardize a simple dimension/unit checker?"

- **Levels 1–3** (dimension, unit, representation) are the established minimum for all units libraries
- Without **Level 4** (quantity kinds): `quantity<Gy> q = 42 * Sv;` — very dangerous; `1 * rad + 1 * sr` — meaningless; different fluid heads are indistinguishable

Why all of these features are needed?

A common reaction is: "Why not just standardize a simple dimension/unit checker?"

- **Levels 1–3** (dimension, unit, representation) are the established minimum for all units libraries
- Without **Level 4** (quantity kinds): `quantity<Gy> q = 42 * Sv;` — very dangerous; `1 * rad + 1 * sr` — meaningless; different fluid heads are indistinguishable
- Without **Level 5** (typed quantities): `width` passes where `height` expected, `speed` assigns to `velocity`, `1 * W + 1 * var` compiles, plane and solid angles collapse to dimensionless, `Ep` can be constructed from width — wrong physics compiles

Why all of these features are needed?

A common reaction is: "Why not just standardize a simple dimension/unit checker?"

- **Levels 1–3** (dimension, unit, representation) are the established minimum for all units libraries
- Without **Level 4** (quantity kinds): `quantity<Gy> q = 42 * Sv;` — very dangerous; `1 * rad + 1 * sr` — meaningless; different fluid heads are indistinguishable
- Without **Level 5** (typed quantities): `width` passes where `height` expected, `speed` assigns to `velocity`, `1 * W + 1 * var` compiles, plane and solid angles collapse to dimensionless, `Ep` can be constructed from width — wrong physics compiles
- Without **Level 6** (affine space): temperature conversions break, timestamps can be multiplied — nonsensical operations compile

Why all of these features are needed?

Removing any of these levels would leave **known classes of real-world bugs undetectable**.

Why all of these features are needed?

Removing any of these levels would leave **known classes of real-world bugs undetectable**.

The Core Library Framework in P3045 defines:

- Framework for systems of quantities
- Framework for systems of units (SI will be defined in a separate paper)
- Quantity and unit arithmetic
- The affine space (`quantity_point`)
- Text output formatting (`std::format` support)
- Interoperability with `std::chrono`

TEACHABILITY

Five steps for beginners

1. Include and use namespaces

```
import std;
using namespace std::si::unit_symbols;
```

Five steps for beginners

1. Include and use namespaces

```
import std;
using namespace std::si::unit_symbols;
```

2. Create quantities

```
std::quantity distance = 42 * km;
std::quantity time = 1.5 * h;
```

Five steps for beginners

1. Include and use namespaces

```
import std;
using namespace std::si::unit_symbols;
```

2. Create quantities

```
std::quantity distance = 42 * km;
std::quantity time = 1.5 * h;
```

3. Do arithmetic

```
std::quantity speed = distance / time;
```

Five steps for beginners (continued)

4. Convert when needed

```
std::println("Speed: {}", speed.in(m / s));
```

Five steps for beginners (continued)

4. Convert when needed

```
std::println("Speed: {}", speed.in(m / s));
```

5. Extract values for legacy APIs

```
double val = speed.numerical_value_in(m / s);
```

Five steps for beginners (continued)

4. Convert when needed

```
std::println("Speed: {}", speed.in(m / s));
```

5. Extract values for legacy APIs

```
double val = speed.numerical_value_in(m / s);
```

This five-step introduction covers the vast majority of beginner use cases. The **advanced features** (typed quantities, affine space, custom systems) are available but **never forced** on the user.

A complete example: hello_units

```
import std;

constexpr std::quantity<std::si::metre / std::si::second> avg_speed(std::quantity<std::si::metre> d,
                                                                    std::quantity<std::si::second> t)
{ return d / t; }

int main()
{
  using namespace std::si::unit_symbols;
  using namespace std::yard_pound::unit_symbols;

  constexpr std::quantity v1 = 110 * km / h;
  constexpr std::quantity v2 = avg_speed(220 * km, 2 * h);
  constexpr std::quantity v3 = avg_speed(std::isq::distance(140 * mi), 2 * h);
  std::println("v1 = {}", v1);                          // v1 = 110 km/h
  std::println("v2 = {}", v2);                          // v2 = 30.555555555555557 m/s
  std::println("v3 = {}", v3);                          // v3 = 31.2928 m/s
  std::println("v3 = {::N[.4]}", v3.in(m / s));         // v3 = 31.29 m/s
}
```

A complete example: **hello_units**

```
import std;

constexpr std::quantity<std::si::metre / std::si::second> avg_speed(std::quantity<std::si::metre> d,
                                                                     std::quantity<std::si::second> t)
{ return d / t; }

int main()
{
  using namespace std::si::unit_symbols;
  using namespace std::yard_pound::unit_symbols;

  constexpr std::quantity v1 = 110 * km / h;
  constexpr std::quantity v2 = avg_speed(220 * km, 2 * h);
  constexpr std::quantity v3 = avg_speed(std::isq::distance(140 * mi), 2 * h);
  std::println("v1 = {}", v1);                            // v1 = 110 km/h
  std::println("v2 = {}", v2);                            // v2 = 30.555555555555557 m/s
  std::println("v3 = {}", v3);                            // v3 = 31.2928 m/s
  std::println("v3 = {::N[.4]}", v3.in(m / s));     // v3 = 31.29 m/s
}
```

No manual conversion factors; Types carry physical meaning; Output includes units automatically; **constexpr**-friendly.

Compilation errors readability

A lot of effort went into **error message quality**.

Compilation errors readability

A lot of effort went into error message quality.

```
error: no matching function for call to 'time_to_goal'
   16 |   quantity duration = time_to_goal(speed, distance);
      |                       ^~~~~~~~~~~~
candidate function not viable: no known conversion from
  'quantity<mp_units::derived_unit<mp_units::si::kilo_<mp_units::si::metre>, per<mp_units::non_si::hour>>{}, [...]>'
  to 'quantity<struct metre{}, [...]>' for 1st argument
    6 | quantity<si::second> time_to_goal(quantity<si::metre> distance,
      |                      ^            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Compilation errors readability

A lot of effort went into **error message quality**.

```
error: no matching function for call to 'time_to_goal'
   16 |   quantity duration = time_to_goal(speed, distance);
      |                       ^~~~~~~~~~~~
candidate function not viable: no known conversion from
  'quantity<mp_units::derived_unit<mp_units::si::kilo_<mp_units::si::metre>, per<mp_units::non_si::hour>>{}, [...]>'
  to 'quantity<struct metre{}, [...]>' for 1st argument
    6 | quantity<si::second> time_to_goal(quantity<si::metre> distance,
      |                      ^            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- The error **names derived quantities and units** in human-readable form
- Template noise is minimized through careful design

Compilation errors readability

A lot of effort went into error message quality.

```
error: no matching function for call to 'time_to_goal'
   16 |   quantity duration = time_to_goal(speed, distance);
      |                       ^~~~~~~~~~~~
candidate function not viable: no known conversion from
  'quantity<mp_units::derived_unit<mp_units::si::kilo_<mp_units::si::metre>, per<mp_units::non_si::hour>>{}, [...]>'
  to 'quantity<struct metre{}, [...]>' for 1st argument
    6 | quantity<si::second> time_to_goal(quantity<si::metre> distance,
      |                      ^            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- The error **names derived quantities and units** in human-readable form
- Template noise is minimized through careful design

Generated types are easy to understand by non-experts and students.

Interactive tutorials and workshops

TUTORIALS

- Your First Quantities
- Unit Conversions
- Safe and Unsafe Conversions
- Compile-Time Protection
- Dimensional Analysis
- Temperature Handling
- ...

Interactive tutorials and workshops

TUTORIALS

- Your First Quantities
- Unit Conversions
- Safe and Unsafe Conversions
- Compile-Time Protection
- Dimensional Analysis
- Temperature Handling
- ...

WORKSHOPS

- Refactor to Strong Types
- Generic Type-Safe Interfaces
- Working with Temperatures
- Extracting Numeric Values
- Typed Quantities of Same Kind
- Distinct Quantity Kinds
- ...

Metrology is an inherently complex problem

Metrology is an inherently complex problem

The library is teachable in five steps, yet powerful enough to model the full complexity of metrology. That balance is exactly what the C++ Standard should provide.

Migration to production

A common concern: *"We have millions of lines of code — how do we migrate?"*

Migration to production

A common concern: *"We have millions of lines of code — how do we migrate?"*

- Projects already using **strong types** (another units library or even simple `struct` wrappers) will find migration **trivial and painless** — the concepts map directly

Migration to production

A common concern: *"We have millions of lines of code — how do we migrate?"*

- Projects already using **strong types** (another units library or even simple `struct` wrappers) will find migration **trivial and painless** — the concepts map directly
- Projects using **raw `double`** (e.g., HEP) face a bigger challenge, but **incremental migration** is fully supported — you don't have to convert everything at once

Migration to production

A common concern: *"We have millions of lines of code — how do we migrate?"*

- Projects already using **strong types** (another units library or even simple `struct` wrappers) will find migration **trivial and painless** — the concepts map directly
- Projects using **raw `double`** (e.g., HEP) face a bigger challenge, but **incremental migration** is fully supported — you don't have to convert everything at once
- The library provides **interoperability layers** for legacy interfaces — wrap at the boundary, keep type safety inside

THE STANDARDIZATION ARGUMENT

Why this belongs in the Standard

1. Vocabulary types must be standardized

- `std::string`, `std::vector`, `std::optional` — all vocabulary
- `std::chrono::duration`, `std::chrono::time_point` — quantities of time
- `quantity`, `quantity_point` — the natural generalization

Why this belongs in the Standard

1. Vocabulary types must be standardized

- `std::string`, `std::vector`, `std::optional` — all vocabulary
- `std::chrono::duration`, `std::chrono::time_point` — quantities of time
- `quantity`, `quantity_point` — the natural generalization

2. Fragmentation is the alternative

- Boost.Units, nholthaus/units, mp-units, dozens more
- Each with incompatible `quantity` types
- Library A's `meter` ≠ Library B's `meter`

Why this belongs in the Standard

1. Vocabulary types must be standardized

- **`std::string`**, **`std::vector`**, **`std::optional`** — all vocabulary
- **`std::chrono::duration`**, **`std::chrono::time_point`** — quantities of time
- **`quantity`**, **`quantity_point`** — the natural generalization

2. Fragmentation is the alternative

- Boost.Units, nholthaus/units, mp-units, dozens more
- Each with incompatible **`quantity`** types
- Library A's **`meter`** ≠ Library B's **`meter`**

3. A standard type ends the fragmentation

- One **`quantity`** type that everyone can use at API boundaries
- Just like **`std::string`** ended the **`char*`** / **`CString`** / **`QString`** fragmentation (for APIs)
- HEP has used raw **`double`** for 30+ years — a standard type finally gives a reason to stop

Why this belongs in the Standard

4. Certification requirements demand it

- MISRA C++, AUTOSAR, DO-178C require dimensional analysis
- Standards compliance is easier when the tool is **in** the Standard

Why this belongs in the Standard

4. Certification requirements demand it

- MISRA C++, AUTOSAR, DO-178C require dimensional analysis
- Standards compliance is easier when the tool is **in** the Standard

5. chrono proves it works

- Before `<chrono>`: raw integers for time, manual conversion factors
- After `<chrono>`: compile-time unit safety, zero overhead, universal adoption
- P3045 does for **all physical quantities** what chrono did for time

Why this belongs in the Standard

4. Certification requirements demand it

- MISRA C++, AUTOSAR, DO-178C require dimensional analysis
- Standards compliance is easier when the tool is **in** the Standard

5. chrono proves it works

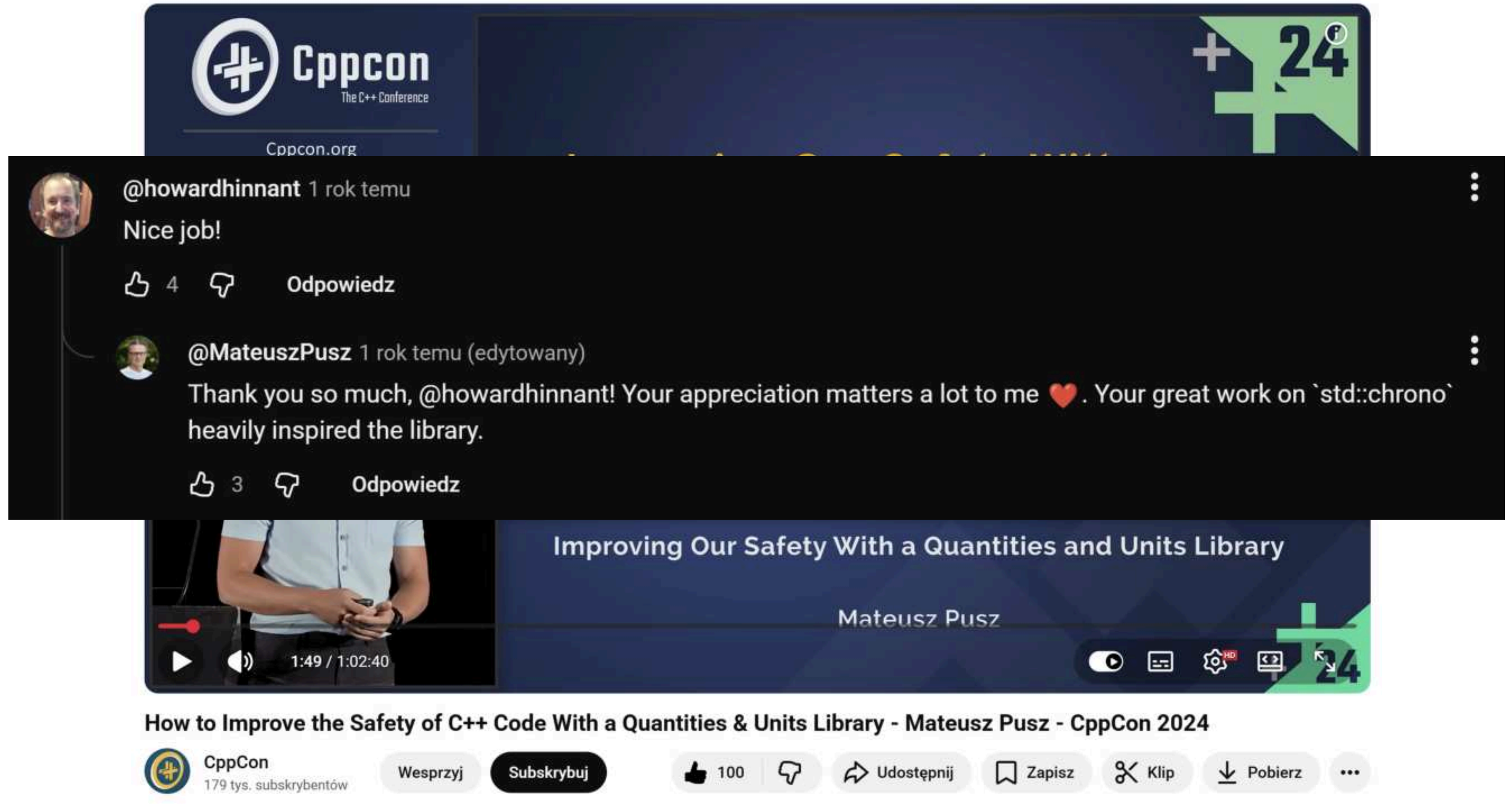
- Before `<chrono>`: raw integers for time, manual conversion factors
- After `<chrono>`: compile-time unit safety, zero overhead, universal adoption
- P3045 does for **all physical quantities** what chrono did for time

6. Metrology is inherently complex

- A proper solution requires handling dimension, unit, representation, quantity kind, quantity type, and affine space
- This is **not** something every project should reinvent

CppCon 2024

CppCon 2024



Tomorrow in LEWG

We are starting the review of this proposal tomorrow in LEWG.

Tomorrow in LEWG

We are starting the review of this proposal tomorrow in LEWG.

- In the Committee, **showing up is voting**
- Papers don't advance on merit alone — they *advance when the people who care are in the room*

Tomorrow in LEWG

We are starting the review of this proposal tomorrow in LEWG.

- In the Committee, **showing up is voting**
- Papers don't advance on merit alone — they *advance when the people who care are in the room*

If this talk convinced you that physical quantities belong in the Standard, please consider attending the LEWG session. Your presence matters.

How you can help?

This work is developed and maintained by an **independent C++ trainer** in his free time.

How you can help?

This work is developed and maintained by an **independent C++ trainer** in his free time.

- *Thousands of hours spent* on mp-units, the papers, and this standardization effort
- There is *still a lot of work ahead* to finalize the design and get this into the C++ Standard

How you can help?

If this project is valuable to you or your organization:

How you can help?

If this project is valuable to you or your organization:

- **GitHub Sponsors** — https://github.com/sponsors/mpusz

How you can help?

If this project is valuable to you or your organization:

- ⭐ **GitHub Sponsors** — https://github.com/sponsors/mpusz
- 💻 **Book a training** — C++ hands-on workshops by the author of mp-units (https://train-it.eu)

How you can help?

If this project is valuable to you or your organization:

- ⭐ **GitHub Sponsors** — https://github.com/sponsors/mpusz
- 💻 **Book a training** — C++ hands-on workshops by the author of mp-units (https://train-it.eu)
- 🛠️ **Contribute** — report issues, submit PRs, help with documentation or test cases

How you can help?

If this project is valuable to you or your organization:

- ⭐ **GitHub Sponsors** — https://github.com/sponsors/mpusz
- 💻 **Book a training** — C++ hands-on workshops by the author of mp-units (https://train-it.eu)
- 🛠️ **Contribute** — report issues, submit PRs, help with documentation or test cases
- 🗣️ **Spread the word** — try the library, give feedback, share with colleagues

How you can help?

If this project is valuable to you or your organization:

- ⭐ **GitHub Sponsors** — https://github.com/sponsors/mpusz
- 💻 **Book a training** — C++ hands-on workshops by the author of mp-units (https://train-it.eu)
- 🛠️ **Contribute** — report issues, submit PRs, help with documentation or test cases
- 🗣️ **Spread the word** — try the library, give feedback, share with colleagues

Every bit of support helps keep this effort moving forward. Thank you!

Resources

- P3045R7 — "Quantities and units library"
- **mp-units library documentation** (https://mpusz.github.io/mp-units)
- **Tutorials and workshops**
- **GitHub** (https://github.com/mpusz/mp-units)
- **Compiler Explorer support**

train IT

Questions?

CAUTION

Programming

is addictive

(and too much fun)